Alpine: A User-Level Infrastructure for Network Protocol Development

David Ely, Stefan Savage, and David Wetherall

Department of Computer Science and Engineering
University of Washington, Seattle WA

Abstract

In traditional operating systems, modifying the network protocol code is a tedious and error-prone task, largely because the networking stack resides in the kernel. For this reason, among others, many have proposed moving the networking stack to user-level. Unfortunately, implementations of this design have never entered widespread use due to the impractical requirements they place on the user: either the kernel or applications must be modified; or code cannot be moved seamlessly between the user-level and kernel stacks. In this paper, we present Alpine, a user-level networking infrastructure free from these drawbacks. Alpine supports a FreeBSD networking stack on top of a Unix operating system. It is freely available as source code. In this paper, we discuss the challenges we faced in virtualizing the FreeBSD networking stack without compromising on kernel, networking stack, and application compatibility. We then show how Alpine is effective at easing the burden of debugging and testing protocol modifications or new network protocols. In our experience, Alpine can reduce the overhead of modifying a protocol from hours to minutes.

1 Introduction

The Internet enables a wide range of applications and supports clients with a wide range of connectivity, from low bandwidth mobile clients to clients with Gb/sec links; and yet there two protocols, UDP and TCP, that govern most of this communication. For this reason, many have proposed modifications and specializations to UDP or TCP to better serve the needs of applications. A literature search for proposed modifications quickly returns many results [3, 4, 5, 6, 10, 11, 13, 16, 18, 20, 23, 25, 28, 30, 31].

This work was funded by generous grants from NSF (DGE-9616736) and DARPA (F30602-00-2-0565). Correspondence concerning this paper may be sent to ely@cs.washington.edu.

Because the networking stack is traditionally part of the operating system, most of these modifications were developed and tested on "live" kernels, which has many drawbacks. Moving the networking stack into a user-level library for development gains the following advantages over developing protocols in the kernel:

- *Shortened revise/test cycle.* Kernel development includes an additional step in the revise/test cycle: a system reboot. This inconvenience increases the turnaround between revisions from a few seconds to a few minutes.
- *Easier debugging.* User-level development allows for easier source-level debugging.
- *Improved stability*. When developing protocols in a user-level environment, an unstable stack affects only the application using it and does not cause a system crash.

To provide these advantages to protocol developers, we have created Alpine (Application Level Protocol Infrastructure for Network Experimentation), a publically available practical tool that moves network protocols into a user-level library to facilitate development. The key distinction between our system and other user-level networking infrastructures is that we have been unwilling to compromise on compatibility with either existing application or kernel code. Once modifications have been developed and tested using Alpine, they are easily moved back into the kernel because Alpine and the kernel use the exact same source files. Furthermore, Alpine works with unmodified application binaries and requires no kernel modifications.

In the process of developing Alpine, we identified three fundamental requirements of virtualizing kernel services to user-level:

- *Virtualization of hardware and kernel routines*. Routines normally available to the kernel but are not available in user-space must be simulated. Hardware must also be virtualized because the actual hardware devices cannot be accessed from user-level.

- *Synchronization with kernel services*. Resources that are shared between the virtualized service and the kernel must be kept consistent without changing the kernel.

- *Transparent integration with applications*. The original semantics of applications that use the virtualized services must be preserved without changing their source code.

Our implementation section provides a more detailed discussion of how Alpine satisfies each of these requirements. The remainder of the paper is organized as follows. We first discuss how our infrastructure relates to previous work and how it differs. Then we present our design and implementation of Alpine in more detail. We then evaluate Alpine's success based on several factors, including performance and ease-of-use. Finally, we conclude with some comments on our experience of building this infrastructure and give directions for future work.

2 Previous Work

Many have proposed moving some or all of the network stack's functionality to user-space. There are essentially three motivations behind this user-level networking:

1. *Improved performance over the kernel's stack.* Work has been done to design high-speed access to device hardware for low-latency cluster processing[8, 32]. In contrast, Alpine is focused on normal applications that use normal networking APIs.

2. *Per-application specialization.* Many have shown that special kernel modifications or downloadable kernel code make application specialization of the networking stack possible [7, 9, 14, 15, 21, 24, 27, 29, 33]. Alpine supports specialization, but this is not its focus. Since our design constraints include requiring no kernel or application modifications, Alpine cannot achieve the high-performance of many of these systems.

3. *Simplified development.* Work has also been done to make kernel development easier. For example, [12] redesigned the kernel from scratch to allow user and kernel modules to be interchanged. Likewise, [17] simplifies development by separating the operating system kernel into encapsulated components, which can be interchanged or reused. In contrast, we have developed Alpine for an unmodified legacy operating system with unmodified application binaries.

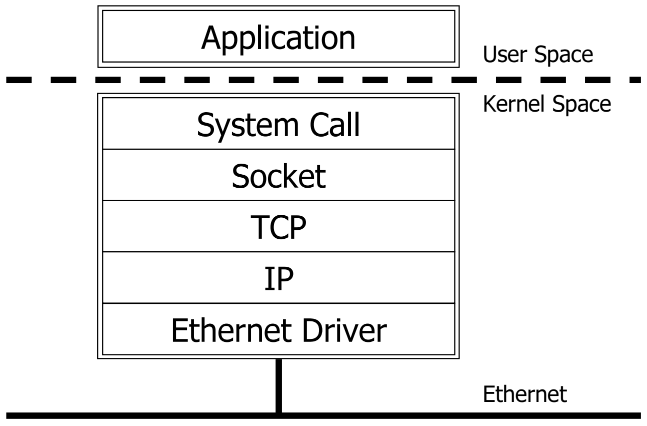


Figure 1: In a traditional network stack, applications interface with the network through a set of system calls (e.g., `socket`, `bind`, `send`), and all network processing is performed in the kernel.

While Alpine provides per-application specialization, its primary purpose is to simplify protocol development. Very little work has been done in this area. Alpine is the first user-level protocol development environment that runs on an unmodified legacy operating system and works with unmodified application binaries. We are focused on delivering an environment that makes modifying and developing networking protocols easier.

Some of our goals are shared by network simulators, which offer a convenient way to test an entire network on a single machine. Unlike Alpine, simulators are very rarely built using the kernel's networking stack. An exception to this is Entrapid [19], which is a simulator that allows a developer to simulate an entire network on a single machine using a modified FreeBSD networking stack. This system is intended more for developing higher level protocols because changes made in the modified stack might not be easily moved into the kernel. While a user can use our library to simulate multiple nodes in a network, Alpine is different from most network simulators [1, 22] because it is not confined to communicating only within a single machine or a single simulation environment.

3 Design

Alpine's primary goal is to be a practical platform for developing network protocols and protocol modifications. This goal lead us to four related design constraints:

- *No kernel changes.* Tools that require kernel modifications are difficult to deploy because of the general apprehension of installing unproven code in the kernel. Kernel modifications are often not portable to other versions of the same kernel, which limits the accessibility of tools that require them.

- *No application changes.* Requiring application changes more severely limits the usefulness of a development tool because *each* application must be modified. Also, the developer might be unfamiliar with the application source code or the application is available only in binary form.

- *No networking stack changes.* If the networking stack were altered, then moving protocol modifications back into the kernel would be difficult because the two stacks are built from separate source code.

- *No administrative oversight.* Administrative barriers, such as requiring an Alpine user to obtain a secondary IP address from their network provider, must be avoided.

In other words, we require Alpine to integrate transparently from the kernel, the application, and the programmer's perspective. This presents the dual challenge of virtualizing access to network and kernel resources while integrating this virtualized system into the native environment. These constraints limit how applications can interface with Alpine and how Alpine can interface with the operating system.

In the traditional Unix network design, a user application interfaces to the network through the socket interface. A socket is a unique communication channel between two hosts, which allow applications to connect to a remote computer, to send and receive data, and to listen for incoming connections. The socket API is a collection of system calls (e.g., `connect`, `sendmsg`, `recvmsg`, and `listen`). Figure 1 shows that the application interacts with the network exclusively through the socket API. The socket code calls into the transport layer (either UDP or TCP), which allows messages to be transmitted between hosts. After UDP or TCP has packaged the message, it sends it to the IP layer, which determines how to route the packet to the destination computer. After determining the appropriate route, IP sends the packet to the interface driver, which is responsible for actually putting the packet on the wire. Each layer includes additional information with the packet in the form of packet headers. In this architecture, the message crosses the dividing line between user-space and kernel-space very early (at the system call layer). This makes debugging the network stack difficult.

We propose to move this line much lower. In Alpine all of the packet processing and framing occurs inside of a user-level networking library. The fully-formed packet (including TCP/IP headers) is sent directly from user-space to the network interface. In our design, we have moved the barrier between user code and kernel code from the system call level to the interface driver. As Figure 2 shows, the unmodified socket layer and the TCP/IP layers have been moved into a library, which is responsible for sending and receiving packets and maintaining state about connections. Ideally, this library interacts with the interface (e.g., Ethernet card) directly, but for security reasons, the interface cannot be accessed without going through the kernel. For this reason, we wrote a software-only interface driver that does not control a hardware interface, but instead it sends packets using a raw socket and receives them using a packet capture library[1]. This system-independent device is termed a faux-ethernet driver because from IP's perspective, it is identical to a normal ethernet driver. To prevent the kernel's networking stack from reacting to packets destined for applications using Alpine, a firewall is installed that filters out Alpine's packets before they reach the kernel's stack.

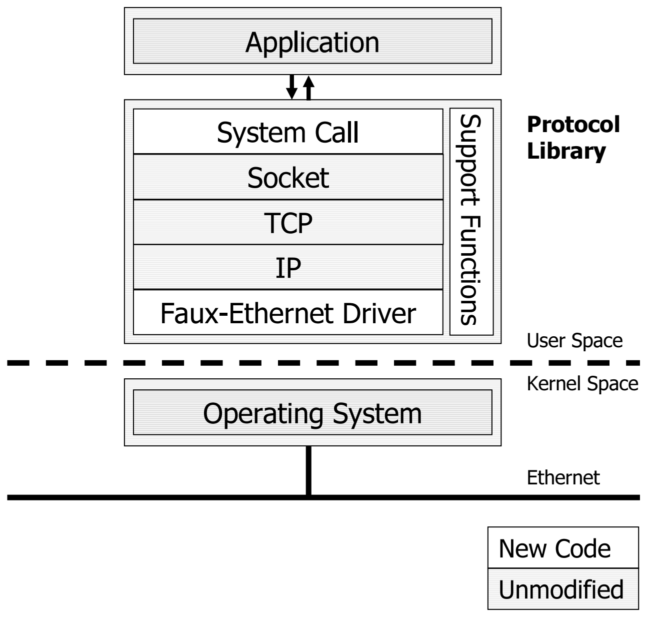


Figure 2: In Alpine the unmodified networking code is placed in a user-level library. An application is not modified because it interfaces with the library using the traditional socket system API.

At the application interface to the networking stack, we exported the same networking API as the kernel. This involved writing a pseudo-system call layer that replaced the traditional networking system calls with our own. We also provide some support code that emulates kernel functions upon which the networking stack relies. These mostly include synchronization functions and functions manipulating kernel data structures.

The following section discusses this design in more detail and attempts to classify the challenges we met when implementing Alpine.

[1]Using raw sockets and the packet capture library requires root privileges, but this is seen as only a minor inconvenience since modifying the kernel also requires root access. In section 6, we propose a solution to this shortcoming.

4 Implementation

We now discuss the implementation challenges of virtualizing network protocols with the design constraint that the kernel, the applications, and the networking stack remain unmodified. All of these challenges are either challenges in virtualization or challenges in integration. From the perspective of the networking stack we must *virtualize* kernel services; from the perspective of the application we must *virtualize* the system call interface to network protocols. We must also *integrate* this virtualized system into a native running environment. This involves managing shared resources such as a port space and a file descriptor table.

Achieving both service virtualization and seamless integration was the main technical challenge we overcame when building Alpine. The following are three fundamental requirements for virtualizing kernel services and integrating them into a running environment:

- Virtualization of hardware and kernel routines
- Synchronization with kernel services
- Transparent integration with applications

These three requirements are not unique to Alpine. They can, and should, be used as guidelines for virtualizing other kernel services. Organized according to these three requirements, our discussion of Alpine follows.

4.1 Virtualization of Hardware and Kernel Services

In Alpine an unmodified kernel networking stack runs inside a user-level library where it is not possible to supply all kernel facilities, such as direct access to hardware or fine grain timers. These facilities must be virtualized using only user-level services because the kernel cannot be modified to extend these services to user-level. In this section, we discuss the specific virtualization challenges that we met, including sending and receiving full-formed IP packets from user-level, simulating the boot process for the networking stack, and managing protocol timers.

Sending/receiving through the faux-ethernet interface

For security reasons, applications and user-level libraries cannot directly access a hardware interface but instead must go through the kernel. We use two different techniques for sending and receiving packets that are similar to accessing the interface directly but do not violate the security imposed by the kernel.

Sending. Sending a packet is straightforward. We chose to use a per-application raw socket that sends preformed IP packets to the operating system for transmission. A raw socket bypasses the transport and IP layer of processing and is sent directly to the hardware interface. The packets sent to a raw socket are identical to those sent to the interface, and there is no impact on the IP layer because it generates identical packets whether it is part of the kernel or part of Alpine.

Receiving. Receiving packets is more complicated. We use the libpcap packet capture library [2] to receive packets. This library enables a user application to receive copies of all packets that are received by a given interface. Other user-level protocol implementations have used this same approach to receive packets [24]. Although Alpine has access to all incoming and outgoing packets, it installs a Berkeley Packet Filter that discards packets destined for other applications. This limits the kernel resources needed to buffer packets that have been received.

Faux-Ethernet. We have encapsulated our methods of sending and receiving packets into a faux-ethernet device which presents itself to IP as any other interface. Although this interface has been named faux-ethernet, it is not specific to ethernets and can easily be adapted to other interfaces such as modems.[2] This faux-ethernet device is attached to the user-level stack during initialization. Because it is the only interface present in the user-level stack, all IP packets are sent through it.

Simulating interrupts

Packets can be sent synchronously, that is, when a user calls `send`, the packet can actually be placed on the wire before returning. But packets *arrive* asynchronously and cannot be processed in this fashion. The kernel receives an interrupt whenever a packet arrives, but this interrupt is not passed up to the application level. We could have modified the kernel to forward this interrupt to Alpine using signals, but this would have violated a design constraint. In order for Alpine to receive packets, it continually poll libpcap to check if a packet has arrived. This is done approximately once every millisecond by using SIGALRM to call an interrupt handler 1000 times per second. This could be done more often if we changed the granularity of the kernel's software timer, but polling for packets once every millisecond has been sufficient. Because we are polling for packets instead of receiving interrupts, there may be up to a 1ms delay between when a packet is received by the interface and when Alpine processes the packet.

Pseudo-kernel environment

Not surprisingly, the TCP/IP stack is not a completely separable part of the kernel. It relies on many features that are only available in the kernel, such as scheduling and certain

[2] It may be necessary to alter certain parameters, such as MTU, to match that of the actual machine interface.

memory allocation routines. The stack also relies on being properly initialized during system startup.

The FreeBSD kernel was modular enough to extract the networking stack without having to bring along a lot of additional code, but some kernel code not pertaining directly to the networking stack was imported for convenience. This includes code to manipulate certain system data structures, synchronization code, and code used for timeouts. The networking stack also relied on certain functions that could not be directly imported from the kernel.

Support Functions. As Figure 2 shows, we implemented a small set of support functions that emulate their counterparts in the kernel. These functions include several operations dealing with memory allocation.

Software Interrupts. The kernel's processing of incoming packets is asynchronous and driven by software interrupts. The interface driver and the protocol layer both use software interrupts to schedule packet processing routines. Like hardware interrupts, a priority level is assigned to each interrupt, and an interrupt service routine can only be interrupted by a higher priority interrupt. The kernel provides functions, such as `splnet` and `splhigh`, to raise the interrupt level and `splx` to restore a previous level. The networking stack often raises the interrupt level when executing critical regions of code to prevent shared data structures from being corrupted. Alpine includes implementations of these software interrupt functions. They are used primarily to prevent Alpine's SIGALRM handler from executing when the application is in a critical region of code. Beyond providing this support code, the second major issue was ensuring that all of the copies of system data structures are properly initialized.

Initialization. Alpine's initialization routine must be called before the user makes any calls into the library. Alpine supplies an `init` function that executes before any other library function is called. It initializes its own internal data structures as well as calling a modified version of the kernel's `main` function. The kernel's `main` function calls the various network initialization routines, such as `ip_init` and `tcp_init`. Finally, a set of dynamic ports is allocated to be used for sockets that are not explicitly bound to ports.

Timer management

An operating system performs many tasks. These include synchronous tasks such as flushing the file cache to disk or scheduling processes, and asynchronous tasks, such as handling user input or processing incoming packets. The networking stack uses `timeout` to handle synchronous events and `tsleep`/`wakeup` for asynchronous events. For Alpine's stack to function properly, we must correctly implement each of these functions and do so without affecting the semantics of the application.

Timeout. The kernel allows protocols such as IP and TCP to export two functions, `slowtimo` and `fasttimo`, which are called periodically. `Fasttimo` is called five times per second, while `slowtimo` is called only twice per second. TCP uses these functions to retransmit missing packets after a given interval and for delayed acknowledgements. Calling these functions five and two times per second is not difficult because Alpine is already using SIGALRM to poll for packets every millisecond. The `slowtimo` and `fasttimo` functions are instances of a more general problem. The kernel has a function, `timeout`, that allows an arbitrary function to be called after a specified number of clock ticks. This `timeout` function is used in multiple places in the TCP/IP stack, and we were able to import the kernel's `timeout` implementation with few modifications.

Tsleep and Wakeup. The function `tsleep` allows the caller to wait on a specific event until a timeout expires. This permits functions such as `recv` to wait until a packet arrives. The caller is restarted when the timeout expires or when `wakeup` is called on the appropriate event. For example, a socket can sleep on its incoming queue, and when a packet is appended to this queue, `wakeup` is called to restart the caller.

`Tsleep` and `wakeup` are not exposed to a user-level application but can only be used inside the kernel. Therefore, it was necessary to implement our own versions of `tsleep` and `wakeup` that preserve their original semantics. The kernel can suspend the caller process until the timeout expires or `wakeup` is called, but because the application and the user-level networking library run in the same process, doing so in Alpine would result in the process sleeping forever. This makes implementing `tsleep` and `wakeup` more difficult because Alpine must continue to run even if the application is blocked. In our simple implementation, `tsleep` busy waits on a global flag, which is set by `wakeup`. To reduce CPU utilization, `tsleep` sleeps for a few microseconds between checks of this global flag.

For an unmodified networking stack to run within Alpine, it was necessary to virtualize several kernel services using only user-level services. These virtualized services borrow heavily from the kernel implementations and are often more simple than the kernel version because the service is only used by a single process. Because the kernel and Alpine share certain state, including a port space and file descriptors, it is necessary to synchronize state with these kernel services. This is discussed in the next section.

4.2 Synchronization with Kernel Services

Alpine provides a service that is also provided by the kernel, and state shared with the kernel, such as ports and file descriptors, must be synchronized between the two stacks. The kernel assumes that it is the exclusive manager of this state, and due to our design constraints, Alpine is responsible for keeping this state consistent without violating the kernel's assumption. To minimize administrative burden,

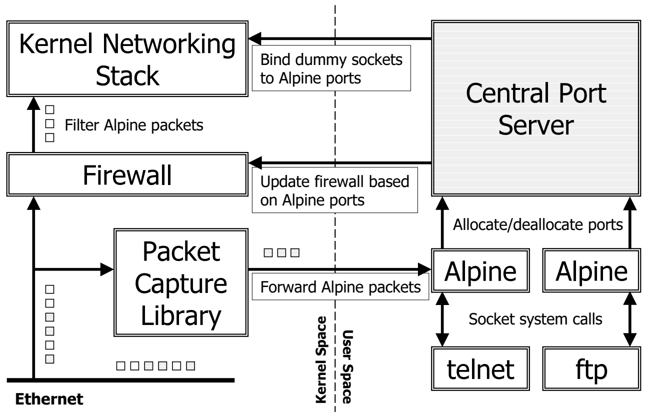


Figure 3: The role of Alpine's central port server is shown. The central server is responsible for allocating ports to each Alpine process. It uses a dummy socket to prevent the kernel from binding another socket to the port, and it uses a firewall to prevent the kernel's stack from reacting to packets sent to the bound port.

Alpine and the kernel's stack share an IP address. Thus, they share a port space that must be kept consistent. We found that having a central user-level process allocate ports to Alpine applications to be the best way to isolate faults and keep the two sets of ports synchronized. As section 3 mentions, Alpine interfaces with applications at the socket layer where Unix applications refer to sockets using file descriptors. Several system calls, including `open` and `close`, were overridden to synchronize Alpine's set of file descriptors (allocated to Alpine's sockets) with the kernel's file descriptors (allocated to files and pipes). Details of the methods we used to keep both the set of ports and file descriptors consistent follow.

Using a central server to mangage port allocation

Many conflicts between the kernel and Alpine can be avoided by allocating a separate IP address to Alpine. However, sometimes obtaining additional IP addresses is not feasible or desirable. In this case, the kernel and Alpine must share a single IP address and the port space that accompanies it. The user-level stack should not interfere with the port allocations of the kernel by re-allocating ports that the kernel has already allocated, and the kernel should not allocate ports that the user-library is using.

To solve both of these problems, each Alpine application allocates ports from a central user-level process. The central server's role in Alpine is shown in Figure 3 with its three specific duties listed below:

- A dummy socket is bound to each port that an Alpine application requests to prevent the kernel from allocating this port to another process.
- A firewall is installed that filters out packets destined for this port to prevent the kernel's stack from receiving and reacting to packets destined for an Alpine application. [3]
- After each Alpine process exits, the firewall is updated and the dummy socket closed for each port that the application was using, allowing other applications to bind to the port.

The primary reason for using a central server is to keep the firewall consistent with the set of ports that Alpine applications are using. Each Alpine application could update the firewall on its own, but this leads to two problems: 1) race conditions exist if multiple applications try to update the firewall concurrently and 2) if the process does not exit cleanly, the firewall may not be uninstalled properly, preventing other applications from using the blocked ports. Centralizing port allocation for Alpine processes is a reasonable way to ensure that the firewall remains consistent. Our approach is similar to that proposed in [29], which uses a "dedicated registry server" to handle connection setup and teardown.

Allocating Alpine file descriptors

For security reasons, applications in Unix cannot directly access a file on disk. Rather, they refer to open files using a file descriptor, which is merely an index into a per-process table of all open files. When a file is opened, a new file descriptor is created, and this file descriptor is passed to subsequent `read` and `write` calls to distinguish which file is being accessed. Because the socket API also uses file descriptors to distinguish between open sockets, we had to override all system calls that allocate or deallocate file descriptors.

Open and Close. A new file descriptor is created whenever `open` is called to open a file or pipe, or when `socket` is called to create a new socket. Because Alpine must know which file descriptors to allocate to its user-level sockets, it keeps track of the file descriptors being used by the kernel by overriding all system calls that create or delete file descriptors. This not only includes `open` and `socket`, which create a file descriptor, but also `close`, which deletes a file descriptor, and `dup`, which creates a copy of a file descriptor. As with the port space, the kernel and Alpine must share a set of file descriptors. Alpine cannot indiscriminately allocate file descriptors to the sockets that it creates because the kernel could allocate the same descriptor to a file in a future call to `open`. We solve this problem by opening a dummy file whenever a new user-level socket is created.

Whenever `socket` is called, Alpine assigns the same file descriptor to this socket as the kernel would, and then it opens the file "`/dev/null`," which prevents the kernel from allocating the chosen file descriptor to another file.

[3] Because the firewall runs on the same machine as the user-level stack, the packet capture library described previously is still able to receive all incoming packets.

This file is closed when the socket is closed. This approach allocates file descriptors identically to the kernel, preserving the original behavior of the application. A table is kept to distinguish our file descriptors from actual kernel file descriptors, enabling Alpine to correctly multiplex overloaded system calls such as `read`.

Although challenges of managing a shared port space and a shared file descriptor table may seem different, they essentially both involve keeping a shared namespace consistent. In fact, they are both solved using the same technique of attaching a false "name" (i.e. a dummy socket or a dummy open file) to a kernel resource to prevent the kernel from allocating it elsewhere.

4.3 Integration with Applications

For a development environment of Alpine's nature to be useful, it must work without modifying existing applications. For instance, having to rewrite application source code is unacceptable. Therefore, Alpine exports the traditional interface to network communication, the socket API. Furthermore, requiring recompiling or relinking of an application may seem acceptable, but this is sometimes inconvenient or impossible, which is why Alpine works with existing executable binaries. Exporting the socket API from Alpine requires manipulating the order in which the application is linked; by linking with the Alpine library before other libraries, Alpine's networking stack is used instead of the kernel's. To work with existing binaries, Alpine exploits dynamic linking; by loading Alpine's dynamic library before any other dynamic library, its networking stack is used instead of the kernel's. This technique cannot be used for applications that are statically linked. Fortunately, most applications are dynamically linked, especially those whose source code is unavailable.

There is an additional concern involved with preserving application semantics. We must ensure that none of the techniques Alpine uses to virtualize kernel services affects the semantics of the application. Two problems that could affect applications involve Alpine's `SIGALRM` handler which is used to perform periodic duties. First, we must allow the application to install a `SIGALRM` handler, and yet not allow the application to override Alpine's `SIGALRM` handler. Second, we must deal with the reentrancy issues introduced by having a signal handler that calls non-reentrant library routines.

Overriding socket system calls

Because we cannot modify the networking stack, we use a faux-ethernet device to send and receive packets. This is the interface that Alpine has with the operating system. A similar issue is at what level to interface with the user application. Applications interface with the network through the socket API, which is a set of system calls that allows an application to connect to another computer and send and receive data. As a design constraint we avoid application modifications, so having Alpine export a socket API is the only choice.

Send and Recv. System calls that are only used by sockets, such as `send`, `recv` and `connect`, were the simplest to implement. These system calls are replaced with our identically named functions, and as long as Alpine is loaded before libc, these socket system calls will be called instead of the kernel's.

Read and Write. In Unix, file descriptors are overloaded to refer to files, pipes, and sockets. With any of these types of "file", the user can call a certain set of overloaded functions including `read`, `write`, and `ioctl`. The operating system multiplexes calls to these functions into the appropriate file, pipe or socket function calls. For example, if read is called with a socket file descriptor then the system translates this into a call to `soreceive`. Therefore, we have to override these system calls and multiplex these calls between actual kernel files or Alpine sockets.

Select and Poll. Finally, parameters to functions such as `select` and `poll`, which determine if there is anything to "read" in a set of files or sockets, can include both file descriptors referring to files and to sockets. A timeout parameter associated with `select` and `poll` determines how long the operating system should wait for the "file" to become readable or writeable[4]. For instance, an application may block waiting either for a pipe to become readable or data to be received in a socket. `Select` will return when either the pipe or the socket becomes readable or when the timeout expires.

Alpine can determine locally if there is anything to read out of the socket buffers, but it must make a `select` call into the kernel to determine if there is anything to read out of the files. The timeout value cannot be passed through to the kernel because an incoming packet might cause a selecting socket to become readable. Thus, when an application is waiting on both a socket and a file, we poll (e.g., use a zero timeout) both the kernel file descriptors and the socket file descriptors until the timeout expires.

Transparent integration with existing binaries

Alpine can be used with unmodified application binaries by exploiting dynamic linking, which delays the binding of function calls until the application executes. The LD_PRELOAD environment variable allows the Alpine dynamic library to be loaded before any other library, which implies that Alpine's networking stack will be used instead of the kernel's. This enables Alpine to be used with any dynamically linked application.

[4]Libpcap's file descriptor cannot be passed directly to `select` because Alpine's SIGALRM handler would not run while the process was blocked, preventing packets from being retransmitted.

Application timers

Applications also use SIGALRM for timeouts and to perform periodic duties, but Unix only allows a single signal handler to be installed for each signal. We must not allow the application to replace Alpine's signal handler, however, we cannot prevent the application from using timers. To solve this problem, Alpine replaces many of the signal based system calls, such as `setsigaction` and `setitimer`, with its own implementations. Alpine records any SIGALRM handler that the application installs, but it does not change the actual handler for this signal. When the application schedules a SIGALRM to be delivered, the application signal handler is called from Alpine's signal handler after the application-specified delay. Because Alpine's signal handler is called at the highest possible frequency, it will always be able to call the application's signal handler at the correct time. However, if Alpine is executing a critical region of code, then this signal is delayed until the next clock tick. This is acceptable because the kernel can also delay delivery of signals for the same reason.

Non-reentrant library routines

Even though Alpine does not use threads, problems still arise with reentrancy because Alpine's SIGALRM handler can be called while the application is executing a non-reentrant library routine. For example, the signal handler should not call `malloc` if the application is updating a global data structure inside of `free`. To solve this problem, Alpine uses wrapper functions to place a lock around non-reentrant library routines, and its signal handler does not execute if the application is executing one of these routines. The application cannot call a non-reentrant library routine in an unsafe way because Alpine's signal handler always runs to completion.

To integrate with unmodified applications, Alpine is required to export the traditional socket interface to the network, and to ensure that the virtualization of kernel services has not altered the semantics of applications. We solved these two classes of challenges by exploiting properties of the linker, which allows Alpine to override any system call or library routine without modifying the application.

5 Experiences

Alpine has been fully implemented in the FreeBSD 3.3 operating system. However, very little of this code is specific to this version of FreeBSD, and most of it is portable to any Unix environment. It was successfully implemented without requiring modifications to the host operating system or applications using Alpine, and the same source files are used to build both Alpine and the kernel's networking stack. Alpine works with most applications, but it does not yet support applications that call `fork` because the fork produces

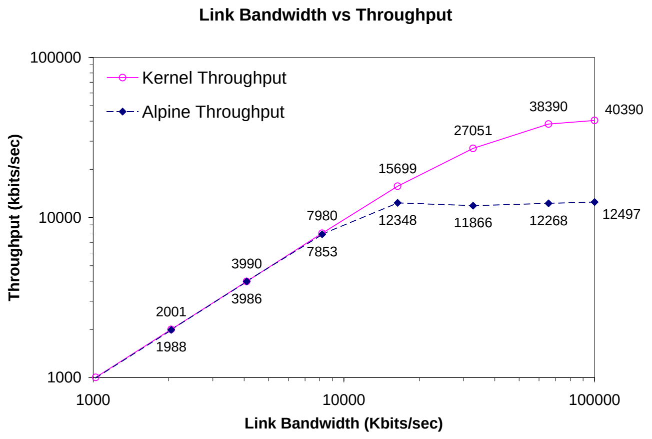


Figure 4: The throughput of both the kernel stack and Alpine are shown as the speed of the link increases. The extra copies, which are used by Alpine to maintain compatibility, limit it to a throughput of 12 Mbit/sec.

a second networking stack. Open connections can be shared between the parent and child processes, which leads to problems in Alpine. For example, because the parent and child have their own copies of the networking stack, each will send acknowledgements to incoming TCP packets. In section 6, we discuss a way to extend Alpine to handle fork.

In the remainder of this section, we first compare Alpine's performance with the performance of the kernel's stack, and then we show possible uses of Alpine. While demonstrating ease-of-use quantitatively is difficult, we believe this section will enable the reader to understand the improvements Alpine makes over kernel development. The examples presented in this section are all related to TCP, but Alpine is certainly not limited to being used with TCP. It can be used to modify or test any transport level protocol.

5.1 Performance

Because Alpine is a user-level *development* infrastructure and is not intended as a replacement for the kernel's stack, its success does not depend on outperforming the kernel. However, Alpine must have reasonable performance in order to be a useful tool. Although certainly slower than the kernel's stack, Alpine can satisfy almost every application's bandwidth and latency requirements. Alpine cannot compete with the kernel's stack on the highest bandwidth links, although for link speeds up to 10 Mb/sec the two achieve similar performance. Latency in the local area is only slightly worse for the user-level stack.

Figure 4 shows how the kernel and Alpine performed as the link speed was varied. The test machine sent data as fast as possible to a second machine. Each machine was configured to use a third machine as a gateway, which used Dummynet to limit the bandwidth of the link [26]. All experiments were run on 200MHz Pentium-Pro PCs running

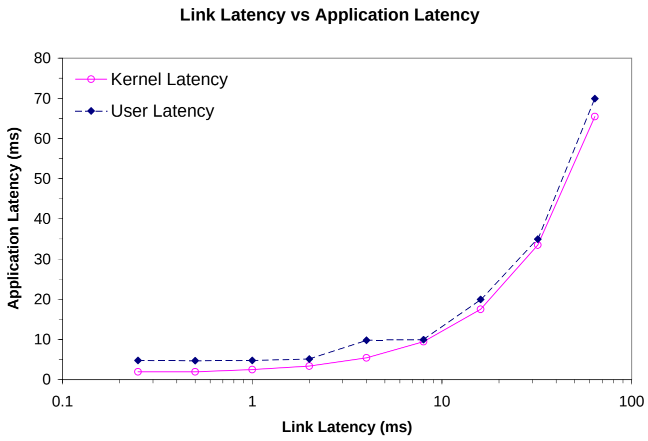


Figure 5: The application latency of the kernel stack and Alpine is shown. Due to additional overhead, the user-level stack is consistently 2.5 ms slower than the kernel network stack.

FreeBSD 3.3. The two stacks are comparable up to link speeds of 10 Mbits/sec, where they start to diverge. While the kernel can achieve up to 40 Mbits/sec, the user-level stack can obtain at most 12 Mbits/sec. Less modest machines could achieve even higher performance.

The extra data copies Alpine needs to integrate seamlessly with applications and the unmodified kernel stack are responsible for this slowdown. In Alpine approximately five copies are necessary between when the application calls `send` and the packet is actually placed on the wire. Comparing this to the two or three copies the kernel needs, it is not surprising that Alpine cannot compete at higher bandwidths[5].

Figure 5 depicts how these extra copies and other overhead affect latency. In this experiment, one byte of data was sent to a remote computer, which immediately echoed the data. The link latency was varied from having no artificial link latency at .25 ms to having a 64 ms link latency. (In the local-area, .25 ms latencies are common, while in the wide-area, latencies of 30-60 ms are typical.) Alpine's latency was consistently 2.5 ms larger than the kernel latency, which is negligible for wide-area applications and is acceptable for most local-area applications.

We hope to improve the throughput and latency of Alpine, but the current design will always be slower than the kernel's stack. This is only a minor drawback because once protocol modifications have been tested in Alpine, they are easily moved back into the kernel where they can achieve higher performance.

[5]The additional copies required by Alpine occur at the interface to the application and the interface to the raw socket. The user buffer is copied into the kernel mbuf data structure which is shared by each layer of the protocol stack. Finally, the faux-ethernet driver copies the fully-formed packet out of the mbuf into a buffer, which is passed to the raw socket send.

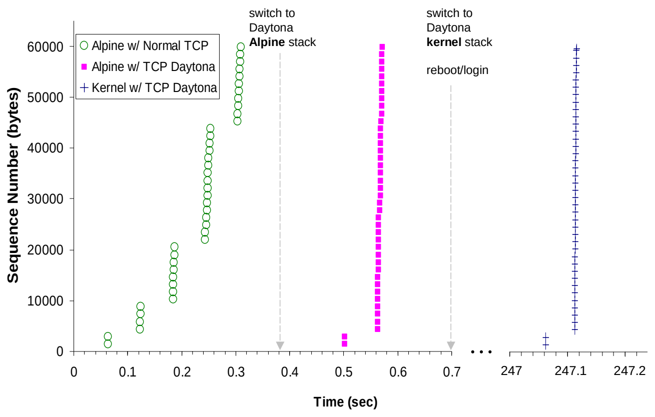


Figure 6: With Alpine very little time is needed to run an application with two different networking stacks, but it requires over four minutes to switch the kernel's networking stack.

5.2 Alpine Improves Protocol Development

By moving protocols into a user-level library, Alpine improves many aspects of protocol development. In this section, we try to give the reader an appreciation of these improvements.

Alpine shortens the revise/test cycle

With Alpine almost no time is needed between testing an application with two different networking stacks, and Figure 6 demonstrates this. The graph in Figure 6 is a time sequence plot where each mark represents a packet being received at the client. A 60 KB file is first downloaded using Alpine running a normal TCP stack, the same file is then downloaded using Alpine running TCP Daytona[6][28], and after a system reboot, the file is downloaded using the kernel stack running TCP Daytona. For all downloads, the same unmodified application was used.

The transition between the two Alpine stacks takes less than a second because only an environment variable must be changed to switch between the two stacks. However, switching to a different version of the kernel stack requires more than four minutes because the machine must be rebooted. While this example may seem extreme, it does demonstrate that time and effort are saved by eliminating the system reboot from the protocol development process.

Alpine improves protocol debugging

When an application uses Alpine, the networking stack runs in user-level. Thus, it can be examined and changed just as

[6]TCP Daytona is a set of modifications to TCP that allow the *receiver* to artificially override congestion control. This allows a client to gain an unfair share of network bandwidth without explicit help from the server.

any other part of the application source code. Any source-level debugger[7] can be used to set break points in the networking stack, step through untested modifications, or modify protocol state by changing protocol variables.

Figure 7 shows a screen shot of Alpine running in a graphical debugger. The large window in the foreground shows a telnet application stopped at the `tcp_output` function with the fields of the protocol control block (PCB) associated with this connection shown in the upper-half of the window. With Alpine, protocol state is easily displayed and modified. Tcpdump[2] runs at the bottom of the screen, showing packets that have already been transferred in the connection. The mail program and web browser running in the background are unaffected by the networking stack halted in the debugger because they are using the kernel's networking stack.

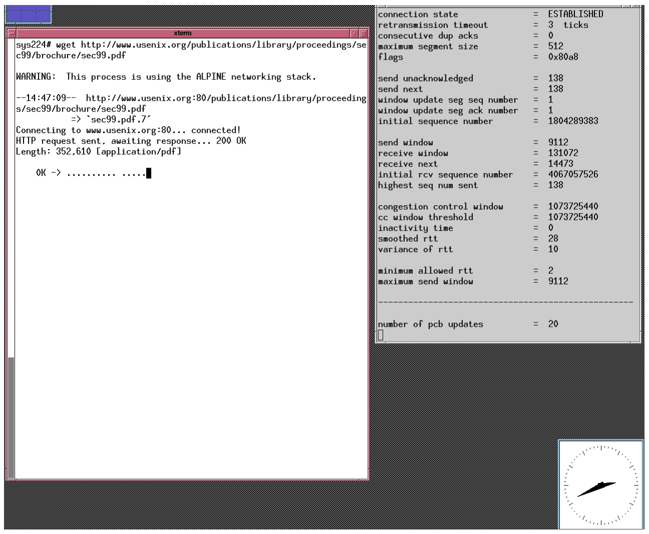
We have found that the power of user-level debuggers makes debugging protocol modifications much easier. Minor bugs that sometimes take hours to find in the kernel usually are found in only a few minutes when using Alpine.

Alpine enables protocol instrumentation

To demonstrate that Alpine can be used in ways the kernel's stack cannot, we instrumented the Alpine networking stack to continuously display the fields of a TCP protocol control block (PCB). A TCP protocol control block (PCB) contains all of the relevant information pertaining to a single connection such as the initial sequence numbers and the size of the congestion window. Figure 8 shows this utility monitoring the `wget` utility while a file is being downloaded. When the PCB for this connection changes, the new PCB is automatically displayed in the window on the right. Achieving this task with the kernel's stack is more complicated because without modifying an application, it is difficult to instrument the kernel on an application-by-application basis. However, with Alpine this utility required only about an hour of programming, and it works with unmodified application binaries.

Alpine simplifies protocol modification

To test Alpine's usefulness when making protocol modifications, we made modifications of varying sizes to TCP. We had many choices for how to modify TCP. For example, we could have changed TCP to get better performance over wireless or satellite links. Instead, we chose to design and implement solutions to prevent the receiver-based TCP attacks, collectively named TCP Daytona, that Savage et. al.



```
sys224# wget http://www.usenix.org/publications/library/proceedings/se
c99/brochure/sec99.pdf

WARNING:  This process is using the ALPINE networking stack.

--14:47:09--  http://www.usenix.org:80/publications/library/proceeding
s/sec99/brochure/sec99.pdf
           => `sec99.pdf.7'
Connecting to www.usenix.org:80... connected!
HTTP request sent, awaiting response... 200 OK
Length: 352,610 [application/pdf]

    OK -> .......... .....
```

```
connection state               =  ESTABLISHED
retransmission timeout         =  3  ticks
consecutive dup acks           =  0
maximum segment size           =  512
flags                          =  0x80a8

send unacknowledged            =  138
send next                      =  138
window update seg seq number   =  1
window update seg ack number   =  1
initial sequence number        =  1804289383

send window                    =  9112
receive window                 =  131072
receive next                   =  14473
initial rcv sequence number    =  4067057526
highest seq num sent           =  138

congestion control window      =  1073725440
cc window threshold            =  1073725440
inactivity time                =  0
smoothed rtt                   =  28
variance of rtt                =  10

minimum allowed rtt            =  2
maximum send window            =  9112

---------------------------------------------------


number of pcb updates          =  20
```

Figure 8: Alpine was used to extend the networking stack to display a connection's TCP PCB as it changes. The window on the right displays the values of fields in the PCB while the application on the left, which is using Alpine, downloads a file. Doing this in the kernel is much more difficult because it does not allow application-specific instrumentation. With the kernel's stack, the PCB must be displayed for all applications or none at all.

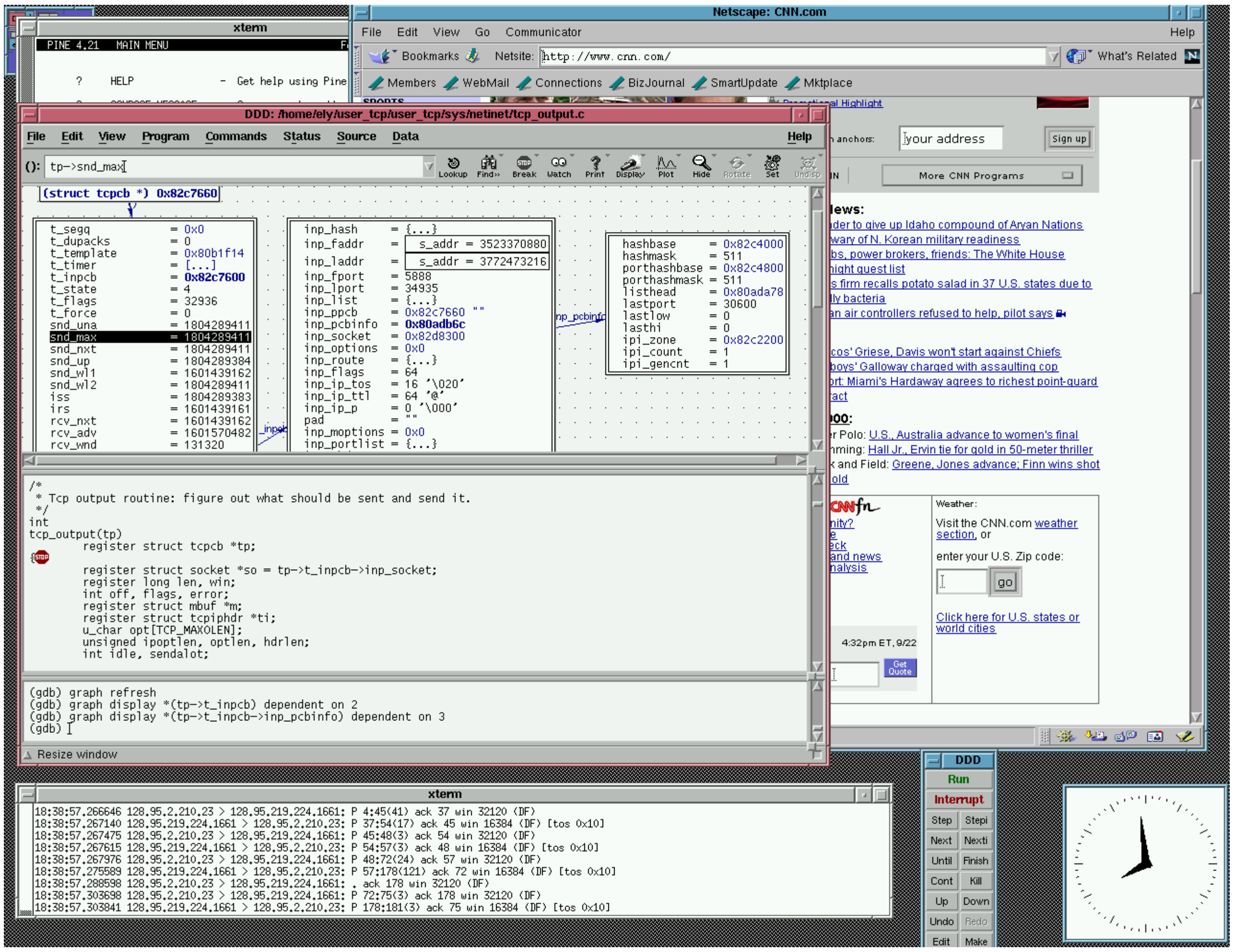
recently discovered [28]. TCP Daytona artificially forces congestion control to be overriden by manipulating *receiver* behavior, which allows the connection to gain an unfair share of network bandwidth. It is not the goal of this work to describe these attacks or discuss how we solved them. Instead, we provide insight into the benefits of using Alpine to implement and test our solutions.

Solutions to two of the attacks required modifications only to be made on the sender (e.g. server) and required approximately twenty lines of code to be added to the networking stack. We found Alpine to be a useful tool during the entire development process. Packet processing code is often written with attention paid to speed (i.e. avoiding procedure calls) instead of code modularity[8], which makes understanding the flow of control difficult. Using Alpine to step through the networking stack, we quickly found the proper place to implement these two solutions. Once our solutions were implemented, we were able to trace through the code to verify that the modifications behaved as expected, and we quickly discovered a bug, which was easily fixed. Once our solutions were sufficiently tested, they were moved into the kernel's stack without having to make any additional changes.

The solution to the third TCP Daytona attack required modifications to both the sender and the receiver (e.g. server

[7] To seamlessly integrate with applications, we must prevent the SIGALRMs from being delivered while the application is stopped (e.g. at a breakpoint). This is easily done by using a debugger initialization file to install hooks that stop these signals when the application stops and resumes them when the application resumes. Alpine provides this initialization file for `gdb`, a popular debugger that is used as a back-end for most Unix-based debuggers.

[8] For example, the two functions used to send and receive TCP packets, `tcp_input` and `tcp_output`, are over 1500 and 700 lines respectively. Understanding the flow of control in such functions is difficult without using a source-level debugger to step through the code.


DDD: /home/ely/user_tcp/user_tcp/sys/netinet/tcp_output.c
File Edit View Program Commands Status Source Data Help
(): tp->snd_max
Lookup Find>> Break Watch Print Display Plot Hide Rotate Set Undisp
Netscape: CNN.com
http://www.cnn.com/
xterm
PINE 4.21 MAIN MENU
DDD
Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make

```
(struct tcpcb *) 0x82c7660

t_segq        = 0x0              inp_hash      = {...}               hashbase      = 0x82c4000
t_dupacks     = 0                inp_faddr     = s_addr = 3523370880 hashmask      = 511
t_template    = 0x80b1f14        inp_laddr     = s_addr = 3772473216 porthashbase  = 0x82c4800
t_timer       = [...]            inp_fport     = 5888                porthashmask  = 511
t_inpcb       = 0x82c7600        inp_lport     = 34935               listhead      = 0x80ada78
t_state       = 4                inp_list      = {...}               lastport      = 30600
t_flags       = 32936            inp_ppcb      = 0x82c7660 ""        lastlow       = 0
t_force       = 0                inp_pcbinfo   = 0x80adb6c           lasthi        = 0
snd_una       = 1804289411       inp_socket    = 0x82d8300           ipi_zone      = 0x82c2200
snd_max       = 1804289411       inp_options   = 0x0                 ipi_count     = 1
snd_nxt       = 1804289411       inp_route     = {...}               ipi_gencnt    = 1
snd_up        = 1804289384       inp_flags     = 64
snd_wl1       = 1601439162       inp_ip_tos    = 16 '\020'
snd_wl2       = 1804289411       inp_ip_ttl    = 64 '@'
iss           = 1804289383       inp_ip_p      = 0 '\000'
irs           = 1601439161       pad           = ""
rcv_nxt       = 1601439162       inp_moptions  = 0x0
rcv_adv       = 1601570482       inp_portlist  = {...}
rcv_wnd       = 131320
```

```
/*
 * Tcp output routine: figure out what should be sent and send it.
 */
int
tcp_output(tp)
        register struct tcpcb *tp;
{
        register struct socket *so = tp->t_inpcb->inp_socket;
        register long len, win;
        int off, flags, error;
        register struct mbuf *m;
        register struct tcpiphdr *ti;
        u_char opt[TCP_MAXOLEN];
        unsigned ipoptlen, optlen, hdrlen;
        int idle, sendalot;
```

```
(gdb) graph refresh
(gdb) graph display *(tp->t_inpcb) dependent on 2
(gdb) graph display *(tp->t_inpcb->inp_pcbinfo) dependent on 3
(gdb) 
```

```
18:38:57.266646 128.95.2.210.23 > 128.95.219.224.1661: P 4:45(41) ack 37 win 32120 (DF)
18:38:57.267140 128.95.219.224.1661 > 128.95.2.210.23: P 37:54(17) ack 45 win 16384 (DF) [tos 0x10]
18:38:57.267475 128.95.2.210.23 > 128.95.219.224.1661: P 45:48(3) ack 54 win 32120 (DF)
18:38:57.267615 128.95.219.224.1661 > 128.95.2.210.23: P 54:57(3) ack 48 win 16384 (DF) [tos 0x10]
18:38:57.267976 128.95.2.210.23 > 128.95.219.224.1661: P 48:72(24) ack 57 win 32120 (DF)
18:38:57.275589 128.95.219.224.1661 > 128.95.2.210.23: P 57:178(121) ack 72 win 16384 (DF) [tos 0x10]
18:38:57.288598 128.95.2.210.23 > 128.95.219.224.1661: . ack 178 win 32120 (DF)
18:38:57.303698 128.95.2.210.23 > 128.95.219.224.1661: P 72:75(3) ack 178 win 32120 (DF)
18:38:57.303841 128.95.219.224.1661 > 128.95.2.210.23: P 178:181(3) ack 75 win 16384 (DF) [tos 0x10]
```

Figure 7: Alpine enables network protocols to be debugged using any source-level debugger. The application being debugged has stopped at a breakpoint in the `tcp_output` function, and the connection's TCP protocol control block (PCB) is being examined. Programs running in the background including a mail client and a web browser continue to use the kernel's networking stack.

and client), and required incorporating almost 100 lines of code into the networking stack. These changes were much more complex and required changing TCP's packet processing code in several places. Beyond the benefits listed previously, making these modifications exposed two other significant benefits of Alpine. First, Alpine allowed us to implement our solution in small increments. The high penalty of making a kernel modification leads many protocol developers to implement and test large pieces of code at once. This increases the number of bugs and probably is more costly in the long run. Our modifications were incrementally implemented and tested because of the absence of the kernel's high modification penalty. Second, Alpine enabled us to run both the sender and receiver concurrently on the same machine. We found this convenience to be very useful because we were able to trace through both stacks concurrently verifying that our modifications behaved as expected. Once this solution was completely tested, it was also moved into the kernel without any additional changes.

6 Future Work

Currently, Alpine's largest drawback is that root privileges are needed to use the infrastructure. (Opening a raw socket, capturing packets using libpcap, and installing a firewall all require root access.) A possible solution to this limitation is to move more functionality, specifically sending and receiving packets, into the central server, which is already used to manipulate the firewall. Once the central server is installed with root access, applications could use Alpine without any special privileges. The central server could also verify the source address of all packets being transmitted to prevent users from abusing the privilege of sending raw packets.

We also have not addressed the issue of what happens when an application forks. Because a forked process inherits its parent's open sockets, handling this issue is tricky. We plan to solve this problem by converting the socket system calls into RPCs that communicate with another process that contains only the Alpine networking stack. In this scenario, a separate user-level networking stack is not created when an application forks, and multiple applications can share a single Alpine stack.

Besides continuing the ongoing maintenance of Alpine, we also hope to port this development infrastructure to Linux and other Unix environments to facilitate protocol development on other platforms. Also, since the FreeBSD version of Alpine relies on almost no platform specific code, it should be easily ported to Linux, allowing an unmodified FreeBSD stack to run on top of the Linux operating system.

7 Conclusion

We have presented an argument for the necessity of a user-level infrastructure for developing network protocols. Developing outside the kernel has many advantages, including easy source-level debugging and quick turnaround between revisions. We discussed our design and implementation of Alpine, which is a publically available tool that enables an unmodified FreeBSD kernel stack to execute in a user-level library. We also presented guidelines for virtualizing other kernel services in a user-level environment. Finally, we showed that Alpine offers many improvements over traditional kernel protocol development. For more information about Alpine or to download the latest version of Alpine, please visit `http://alpine.cs.washington.edu/`.

References

[1] Ns network simulator. See `http://www-mash.cs.berkeley.edu/ns/`.

[2] Tcpdump home page. See `http://www.tcpdump.org/`.

[3] M. Allman. On the generation and use of TCP acknowledgments. *INFOCOM '98*, 28(5):4–21, October 1998.

[4] M. Allman. TCP byte counting refinements. *Computer Communications Review*, 29(3), July 1999.

[5] M. Allman, D. Glover, and L. Sanchez. Enhancing TCP over satellite channels using standard mechanisms. Request for Comments 2488, Internet Engineering Task Force, January 1999.

[6] H. Balakrishnan. *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*. PhD thesis, Computer Science Division, Univ. of California at Berkeley, Berkeley, CA, August 1998.

[7] A. Banerji, J. Tracey, and D. Cohn. Protected shared libraries-A new approach to modularity and sharing. In *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA*, pages 59–75, Berkeley, CA, USA, January 1997. USENIX.

[8] A. Basu, V. Buch, W. Vogels, and T. von Eicken. Unet: A user-level network interface for parallel and distributed computing. In *Proceedings 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO, December 1995.

[9] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain CO, December 1995.

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services: IETF RFC 2475, December 1998.

[11] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings, 1994 SIGCOMM Conference*, pages 24–35, London, UK, August 31st - September 2nd 1994.

[12] R. Draves and S. Cutshall. Unifying the user and kernel environments. Technical Report MSR-TR-97-10, Microsoft Research, March 1997.

[13] R. Durst, G. Miller, and E. Travis. TCP extensions for space communications. In *Proceedings, 1996 MobiComm Conference*, November 1996.

[14] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[15] M. Fiuczynski and B. Bershad. An extensible protocol architecture for application-specific networking. In *USENIX 1996 Annual Technical Conference*, San Diego, California, January 1996.

[16] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. Internet Draft,

Internet Engineering Task Force, February 1999. Work in progress.

[17] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The flux oskit: A substrate for os and language research. In *16th ACM Symposium on Operating Systems Principles*, October 1997.

[18] T. Henderson and R. Katz. Transport protocols for internet-compatible satellite networks. *IEEE Journal of Selected Areas in Communications*, 17(2):326–344, February 1999.

[19] X. Huang, R. Sharma, and S. Keshav. The Entrapid protocol development environment. In *INFOCOM '99*, New York, March 1999.

[20] V. Jacobson, B. Braden, and L. Zhang. TCP extension for high speed paths. *RFC 1185*, October 1990.

[21] M. F. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997.

[22] S. Keshav. Real 5.0 network simulator. See `http://www.cs.cornell.edu/skeshav/real/overview.html`.

[23] L. Larzon, M. Degermark, and S. Pink. UDP lite for real time multimedia applications. Technical Report HPL-IRI-1999-001, HP Labroratories, April 1999.

[24] C. Maeda and B. Bershad. Protocol service decomposition for high-performance networking. In *14th ACM Symposium on Operating Systems Principles*, December 1993. also CMU Technical Report CMU-CS-93-131.

[25] R. Ramanathan. TCP for high performance in hybrid fiber coaxial broad-band access networks. *IEEE/ACM Transactions on Networking*, 6(1):15–29, February 1998.

[26] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *Computer Communications Review*, 27(1):31–41, January 1997.

[27] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating system. In *Proceedings USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 39–69, 1992.

[28] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *Computer Communications Review*, 29(3), October 1999.

[29] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, October 1993.

[30] F. Theodore, J. Touch, and W. Yue. The TIME-WAIT state in TCP and its effect on busy servers. In *INFOCOM '99*, New York, March 1999.

[31] V. Visweswaraiah and J. Heidemann. Improving restart of idle TCP connections. Technical Report 97-661, ISI, Marina del Ray, California, November 1997.

[32] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–267, Gold Coast, Australia, May 1992. ACM Press.

[33] D. Wallach, D. Engler, and M. F. Kaashoek. ASHs: application-specific handlers for high-performance messaging. *Computer Communications Review*, 26(4):40–52, October 1996.