Environmental Independence: BSD Kernel TCP/IP in Userspace

Antti Kantee
Helsinki University of Technology
pooka@cs.hut.fi

Abstract

Code is code. In the entire application stack only a very small fraction of code is special in the sense that it requires the hardware to run in privileged mode. In theory all the rest can run in either the kernel or user domain, with the tradeoffs being well-known.

Engineering an operating system core so that the same code can function both in user and kernel mode alike provides numerous benefits: testing and development, virtualization, stability and reusability of code in applications.

In the current Unix style, code is divided to kernel code and userspace code. Although some limited code modules, such as byte order and address translation routines may be shared, functionality at large is not available in both domains.

This paper discusses the nuts and bolts of running the BSD networking code in userspace. It does not advocate turning BSD into a microkernel operating system, but rather gives concrete level proof that the kernel networking stack and kernel networking applications can be run in userspace. The virtual userspace networking stack is measured to be performant enough to be able to saturate a regular network and to be over 10 times faster than the kernel networking stack running in a machine emulator. An implementation for NetBSD available from the NetBSD source repository is discussed.

1 Introduction

The BSD networking stack is the industry standard for TCP/IP interoperability. Despite it being an old, stable and well-understood piece of code [9, 13], its vast size makes full comprehension difficult even for experienced programmers with years of familiarity. This is exacerbated by the fact that the networking stack runs in the kernel, as the kernel is always to be considered to be a difficult programming environment.

This paper presents *rump* [7] *networking*: a method for running the BSD kernel TCP/IP stack as part of a userspace process using the Runnable Userspace Meta Program (rump) framework found in NetBSD [11]. The rump project aims modify the kernel structure so that kernel code can run equally well both in the kernel environment and the user environment. All networking stack code apart from the device driver layer is used directly from the main kernel - no rewriting, editing or `#ifdef` application is done on kernel code. Functionality of rump networking is equal to that of the in-kernel networking stack.

The original motivation for the work was running the kernel NFS file system client in a userspace application. Since NFS uses the kernel sockets interfaces for communication with the server, supporting the in-kernel sockets interface in userspace was necessary.

Apart from the use mentioned in the previous paragraph, the userspace networking stack can be used for networking stack debugging, development, instrumentation and testing. Additionally, the system allows to implement applications which would otherwise require a full operating system running inside a virtual machine. One potential example is the TorVM anonymizer explored in more detail in Chapter 4.4.

The remainder of this paper is laid out in the following manner. Chapter 2 describes the implementation in detail. Chapter 3 evaluates the portability and Chapter 4 focuses on the applications for the technology described in this paper. Chapter 5 concentrates on evaluating the result in terms of performance as opposed to a networking stack running in the kernel domain. Finally, Chapter 6 concludes.

1.1 The Approach

We call the operating system instance our networking kernel code process is running on the *host operating system* or host for short. Having the kernel networking stack run in a userspace process is a common sight. For instance, when running an operating system in a ma-

chine emulator such as qemu [1], the entire OS kernel is already running in userspace under emulation. Another similar situation is with a usermode operating system, such as Usermode Linux [3] or the DragonFly vkernel [5]. Here the host operating system, which is by implementation required to be the same operating system as the usermode operating system, is an analogous component to the virtual machine.

In these scenarios, the whole operating system, including the root file system, processes, etc., runs as a self-contained package. Communication with the operating system kernel is done from a process running inside the virtual operating system. This indirection difference is further illustrated Figure 1. In this paper we call virtual kernels which are not directly controllable from the host operating system *indirect virtual kernel*s. Notably, a usermode operating system might use host processes to handle its own processes, but since these are isolated from the host operating system, they are considered indirect as well.

Our approach is to extract only the relevant bits from the kernel while leaving out complex parts such as address space management. The benefits are multiple. This provides a portable library approach, with the kernel functionality portable to virtually any platform. The interface with the virtual kernel is a direct library function call interface, so writing tests and getting useful output from them is much easier. Finally, less resources from the host system are consumed since unnecessary components such as fork/exec support or file systems are not being loaded or used. We call this type of process-local virtual kernel a *first class virtual kernel* since it is directly accessible from host processes.

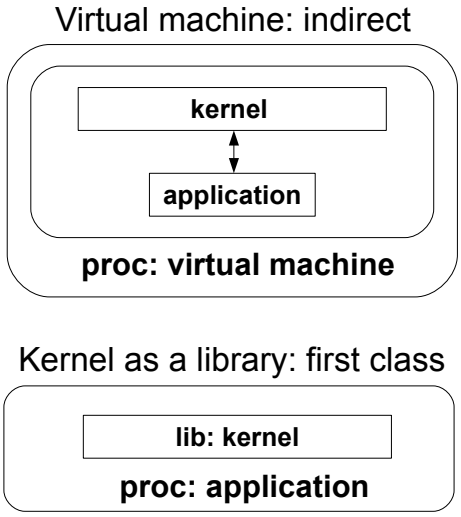

Figure 1: Two Virtual Kernel Styles

2 Implementation

For the discussion, we define two terms to differentiate between how we bring functionality to userspace. When functionality is *extracted* from the kernel, it is brought to userspace as such without any code modifications. This is practical only on a source module granularity, as it will bring in future modifications as opposed to creating another copy of the routines. Conversely, when functionality is *reimplemented*, the kernel interfaces are rewritten for userspace. To give examples, the radix trie routines are extracted, while the pmap virtual memory routines are reimplemented.

It should be noted that all the interface and variable dependencies of functionality both extracted and reimplemented should be satisfied or the final program will not link properly. This can be done either by extracting the modules depended on or reimplementing unused interfaces as stubs. The main dependency classes of sockets+TCP+IP in the networking stack are: memory allocators, synchronization routines, file descriptor subroutines, pmap, software interrupts, callouts, and miscellaneous routines such as `memcpy`() and `strlcpy`().

We must be able to access the host OS process services to implement functionality in our virtual kernel. For example, we must be able to request more memory from the host so that we can allocate memory for use in our virtual kernel. For this purpose, a special *rumpuser* interface set is defined. It provides access to approximately 40 select routines available in a userspace process namespace. While this approach requires explicit handling, it allows to implement the rumpuser interface for each host platform based on the interfaces provided by that platform instead of implicit assumptions in the kernel portion of the code. Notably, only reimplemented code can call rumpuser routines, since the rumpuser interfaces are not usable in a regular kernel and therefore not used in extracted code.

2.1 Background

To understand the userspace implementation, we must first review the basics of the BSD kernel networking implementation [13].

The sockets layer provides an abstracted view of the networking protocols to the rest of the kernel. Each protocol family, for example inet6, registers a *domain* structure to the kernel containing information about the family. Among other information, this structure contains a pointer to a *protocol switch* array. The member structures of the protocol switch array describe each protocol sup-

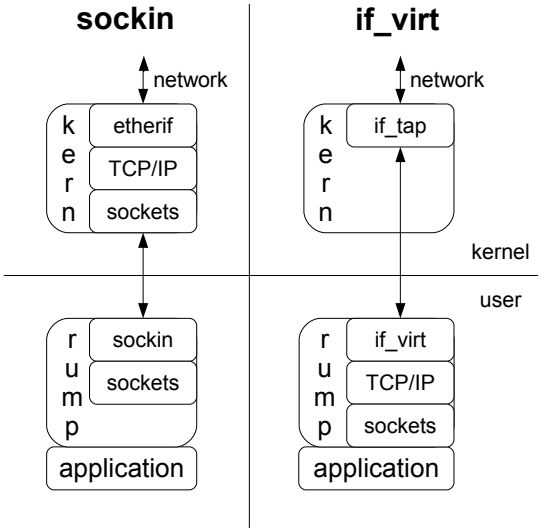

Figure 2: sockin vs. if_virt

Figure 3: sockin protosw

```
const struct protosw sockinsw[] = {
{
    .pr_type = SOCK_DGRAM, /* UDP */
    .pr_domain = &sockindomain,
    .pr_protocol = IPPROTO_UDP,
    .pr_flags = PR_ATOMIC|PR_ADDR,
    .pr_usrreq = sockin_usrreq,
    .pr_ctloutput = sockin_ctloutput,
},{
    .pr_type = SOCK_STREAM, /* TCP */
    .pr_domain = &sockindomain,
    .pr_protocol = IPPROTO_TCP,
    .pr_flags = PR_CONNREQUIRED
      |PR_WANTRCVD|PR_LISTEN
      |PR_ABRTACPTDIS,
    .pr_usrreq = sockin_usrreq,
    .pr_ctloutput = sockin_ctloutput,
}};
```

ported by the family and contain access method pointers.

When a socket is created, the relevant protocol switch is located and attached to the in-kernel socket data structure. When the sockets layer needs to access the protocol, e.g. for sending data, it does so through the protocol switch. When receiving data, it is up to the protocol to decide how it wants to map the received data to a socket. For example, for TCP and UDP this is done using the inet protocol control block module and usually means mapping the (laddr,lport,faddr,fport) 4-tuple to the correct socket.

2.2 Accessing the network

There are two fully supported possibilities for accessing the network from our userspace solution. They have different implications in what they provide and require. Both are described next and the difference is further illustrated in Figure 2.

- **socket layer emulation** (sockin): Access to the network is handled through the host OS's sockets. The virtual kernel implements a PF_INET networking domain, which provides a mapping between the protocol switch access routines and the socket system calls. The protocol switch array of sockin is displayed in Figure 3. It only supports features so far required in use cases. For example, support for ICMP is missing from sockin.

  This approach does not require anything beyond normal user privileges to setup or use and communicates with the outside world using the host OS's IP addresses. It does not run the full networking stack in userspace and its usefulness is limited to running kernel consumers of the sockets layer, such as NFS, in userspace. Additionally, code for each networking domain is required. Currently, support is limited to UDP and TCP on IPv4, although this could be extended to IPv6 with little effort.

- **full networking stack** (if_virt): All the layers of the networking stack except for the device driver are run in userspace. Access to the network is handled through a virtual interface driver, if_virt. This driver communicates with the host networking facilities by using the host's Ethernet *tap* driver, which allows raw access to a host interface through a character device node. The tap interface on the host can further be bridged with real network interfaces to allow unrestricted network access from the virtual kernel.

  This approach requires superuser privileges to configure, but in return provides the full networking stack in userspace, along with a unique address for each virtual interface. This means for example that port binding does not depend on the ports already in use on the host OS. It also makes it possible to build complex virtual networks within one machine using many virtual kernels.

As both the real TCP/IP stack and sockin both provide a PF_INET domain, they cannot coexist in the same virtual kernel at the same time. This dual approach allows to run components which just use the kernel sockets layer, such as the kernel NFS client, without any administrative

hassle or root privileges, but still enables the use of the full TCP/IP stack when it is desired. This contrasts the choices in the Alpine [6] userspace networking facility, where the only option is using an OS-wide port server daemon, a raw socket and the pcap library.

2.3 Emulating the Kernel

Most functionality used by the kernel networking stack can be extracted and used as such. A good example is the mbuf interface. While simple in principle, a reimplementation would be very complex and error-prone because of all the intricacies with the interface. Therefore, it was an obvious candidate for extraction even when it required extra effort with the mbuf backing pool allocator.

However, not all parts can be extracted. For instance, we do not get a hardware clock interrupt in userspace. Therefore, a clock interrupt must be reimplemented if timers are to work. This chapter discusses reimplemented interfaces.

2.3.1 Threading and Concurrency

A large portion of the modern BSD kernel depends on multiprocessing capabilities and concurrency control. This includes the ability to create threads and manage concurrency using primitives such as mutexes and condition variables.

Almost all of this functionality is provided for userspace programs by the pthread library. The naming and calling conventions of pthread routines are slightly different from the kernel, e.g. kernel mutex lock cannot fail, but in general the mapping is just name translation. The only exception is the historic `tsleep`()/`wakeup`() interface, but that is essentially a condition variable and can be implemented by using one.

Multithreading implies thread scheduling. Instead of the kernel scheduler, this is handled by the pthread scheduler. Most of the kernel is under finegrained locking and will work without problems. However, for code still under the big kernel lock we need to identify all the blocking points in the rumpuser interface and wrap those calls to release and reacquire the biglock. Otherwise, blocking with the biglock held could deadlock the system.

2.3.2 Soft Interrupts

Soft interrupts are a method of deferring processing. They are commonly used from a machine interrupt context to schedule work to happen later. A well-known use in the networking stack is to schedule protocol processing as a soft interrupt once the received packet has been cleared off the network interface.

In userspace, our thread scheduled device driver receives packets in a thread context. There is no real reason to defer processing to a soft interrupt, so theoretically it would be possible to process the packet directly instead of scheduling a separate soft interrupt. However, doing the unexpected yields unexpected results: the NetBSD Ethernet input routine schedules the interrupt before it enqueues the newly received packet onto the receive queue. Attempting to process the packet directly in calling context without separately scheduling a soft interrupt will result in the packet not being on the queue when the handler runs. It is less work to emulate the system as closely as possible.

We reimplement a simple soft interrupt framework. Currently, soft interrupts get scheduled for execution in a separate thread dedicated to soft interrupt execution. However, there is room for improvement, as this does not emulate the current kernel perfectly: a real soft interrupt will run only on the CPU it is scheduled on. Since rump does not currently emulate CPUs and hence define the concept of relinquishing a CPU, the soft interrupt might run before the caller that scheduled it finishes executing. This is turn might lead to race conditions such as the one mentioned in the previous paragraph. However, experience has show that this is not a practical issue currently.

2.3.3 Timers

Networking depends on having timers present in the system. For instance, TCP retransmission depends on retry when a timeout occurs and each networking protocol can define a fast and slow timeout, which are called every 200ms and 500ms, respectively. They are centrally executed when a kernel *callout* timer expires and historically [13] all work done by protocol timers was done directly from the fast and slow timeouts. This made it possible for the system to specify a small number of callouts, which was beneficial, since the cost using timers was proportional to the number of timers in the system. It, however, required checking all possible timeouts when the timer fired. For example, for TCP this means that the cost increases with the number of connections regardless of if timeout processing is required or not.

After an O(1) callout facility for BSD was crafted [2] and implemented in NetBSD, there was no strict reason to process all timers from a single callout. In fact, having separate callouts for all connections makes processing necessary only when there is a reason for it.

We extract the kernel callout module to userspace. The module depends on receiving a soft interrupt for doing processing. We use a timer thread which wakes up every clock tick and calls the callout hardclock routine. This, if necessary, schedules a callout soft interrupt for processing the callouts.

3 Portability

The networking stack is in theory portable to all operating systems. However, currently the binary representations of various symbols are not likely to be the same from one system to another. For instance, the size of `struct sockaddr_in` or the value of the symbol `SOCK_STREAM` may vary from platform to platform. Therefore, supplying the host representation of these symbols to the NetBSD first class virtual kernel is not correct. Likewise, giving the NetBSD kernel representation of some symbols to the rumpuser interface will confuse the host system.

If the host is any of the BSD systems, the problem is less severe, since common history means that most symbols will have the same values. Also, the compatibility code of NetBSD helps with many issues. For example, in the development branch leading to NetBSD 6.0 `time_t` was made a 64bit integer. This changed the size of `struct timeval` used in the `SO_RCVTIMEO` and `SO_SNDTIMEO` options of the `setsockopt()` system call. However, new values were given to those two symbols. If a call is made with the old value, the NetBSD kernel will assume a 32bit `time_t` field and do compatibility processing. Since e.g. older versions of NetBSD and FreeBSD define the old values for `SO_RCV/SNDTIMEO`, calls from them to rump with the host values will work even thought they have a 32bit `time_t`.

However, compatibility does not provide a full guarantee. To fully address the problem, the interface between the host and the first class virtual kernel should be binary-independent. A viable way to accomplish this is to use the NetBSD proplib [10] library, which can be used to manage property lists and transform them to an external ASCII representation independent of the host operating system.

4 Applications

Obvious applications for rump networking are rump applications which need to use the kernel networking facilities. The original motivation was given in the introduction: the kernel NFS client. Other applications of rump networking, both from a development and an application perspective, are presented in this chapter.

4.1 Application Interface

To access the virtual kernel's network facilities, an interface is required. On a normal system this interface is the system call interface. However, since the process containing the virtual kernel is not isolated from the host system, a regular system call will be directed to the host kernel instead of the virtual kernel. Therefore, a way of specifying that the request is directed to the virtual kernel is required.

Figure 4: Adapting ttcp to rump

```
#include <rump/rump.h>
#include <rump/rump_syscalls.h>

#define    accept(a,b,c)\
  rump_sys_accept(a,b,c)
#define    bind(a,b,c)\
  rump_sys_bind(a,b,c)
#define    connect(a,b,c) \
  rump_sys_connect(a,b,c)
#define    getpeername(a,b,c)\
  rump_sys_getpeername(a,b,c)
#define listen(a,b) \
  rump_sys_listen(a,b)
#define    recvfrom(a,b,c,d,e,f)\
  rump_sys_recvfrom(a,b,c,d,e,f)
#define    sendto(a,b,c,d,e,f)\
  rump_sys_sendto(a,b,c,d,e,f)
#define    setsockopt(a,b,c,d,e)\
  rump_sys_setsockopt(a,b,c,d,e)
#define    socket(a,b,c) \
  rump_sys_socket(a,b,c)
```

So solve this, we provide *rump system calls*, which behave exactly like regular system calls, but instead of trapping to the kernel they make a function call to access the virtual kernel.

The rump system calls have the same interface and basename as regular system calls. The only difference is that they are prefixed with a `rump_sys_` prefix before the system call name. For an example, `rump_sys_socket(PF_INET, SOCK_DGRAM,0);` will open a UDP socket in the virtual kernel.

To make an existing application use the virtual kernel in select places instead of the host kernel, two approaches are possible:

- **link symbol wrappers**: In this scenario the program is linked with a library which overrides the desired libc system call stubs. For instance, the library provides the symbol `socket()` which just calls `rump_sys_socket()`. For dynamic binaries on most systems just setting `LD_PRELOAD` is enough. For static binaries the program must be relinked.

  The benefit of this approach is that it does not require access to the source code. The downside is that it may be difficult to control which calls to wrap and which not to in case the same system call

Figure 5: Network Interface Configuration Using *rump syscalls*

```
/* get a socket for configuring the interface */
s = rump_sys_socket(PF_INET, SOCK_DGRAM, 0);
if (s == -1)
        err(1, "configuration socket");

/* fill out struct ifaliasreq */
memset(&ia, 0, sizeof(ia));
strcpy(ia.ifra_name, IFNAME);
sin = (struct sockaddr_in *)&ia.ifra_addr;
sin->sin_family = AF_INET;
sin->sin_len = sizeof(struct sockaddr_in);
sin->sin_addr.s_addr = inet_addr(MYADDR);

sin = (struct sockaddr_in *)&ia.ifra_broadaddr;
sin->sin_family = AF_INET;
sin->sin_len = sizeof(struct sockaddr_in);
sin->sin_addr.s_addr = inet_addr(MYBCAST);

sin = (struct sockaddr_in *)&ia.ifra_mask;
sin->sin_family = AF_INET;
sin->sin_len = sizeof(struct sockaddr_in);
sin->sin_addr.s_addr = inet_addr(MYMASK);

/* toss to the configuration socket and see what it thinks */
rv = rump_sys_ioctl(s, SIOCAIFADDR, &ia);
if (rv)
        err(1, "SIOCAIFADDR");
rump_sys_close(s);
```

should be directed to both the host and virtual kernel. A limited number of heuristics can be used in the wrapper: for instance, if the `read()` call needs partial wrapping, the `socket()` wrapper can return a file descriptor number which is very unlikely to be received from the host kernel and use the file descriptor number to decide whether the call should be directed to the host kernel or the virtual kernel. However, these heuristics are never fully robust, and if available, modifying the source may be a better approach.

- **modify the source**: In this option the source code is modified and the application recompiled. The modification can be done both for all calls of a certain type by using the C preprocessor as well only certain calls. A hybrid approach was used to adapt the ttcp measurement tool used for performance evaluation in Chapter 5. Most of the calls could be handled by the preprocessor, but the `read()` and `write()` calls were done on a case-by-case basis. The preprocessor portion is demonstrated in Figure 4.

  The benefit of this approach is that there is no need for heuristics and the control is absolute. The downside is that source code is required. Another problem is that modifying a program is error-prone. Finally, if a library does the system call, it is not possible to solve the problem by editing just the application. For solving this the wrapper approach might be effective.

A good technique in figuring out which symbols need to be wrapped is to use the `nm` utility. The set of unresolved symbols contain the ones needing redirection.

An example of how to configure the address of a networking interface is presented in Figure 5.

4.2 Testing and Development

This chapter gives an example of how rump networking can be used to develop and test the kernel TCP/IP stack. Speaking from the author's personal experience

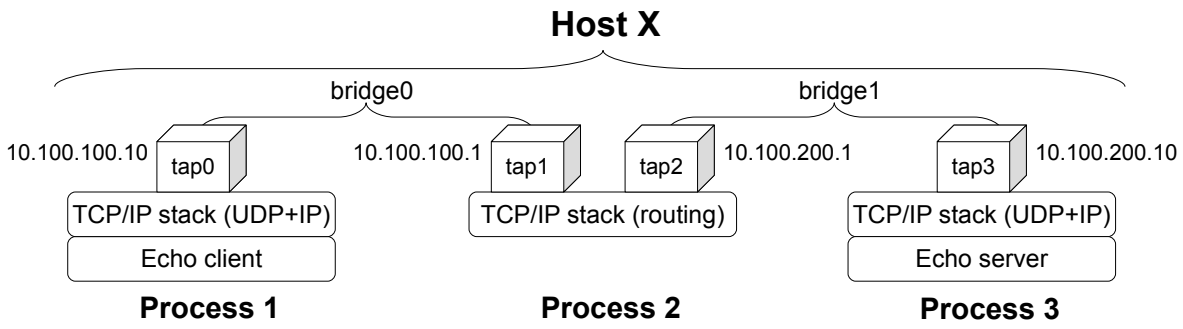

Figure 6: Virtual Router Test Setup

with BSD networking stack development [8] [1], developing infrastructure changes in a userspace program is multitudes easier than trying to do the same in the kernel, even if using a virtual machine.

As the case study, we illustrate how it is possible to test a routing setup within one machine using multiple processes each containing their own copy of a networking stack. While the same approach is possible with other forms of virtualization, as stated in the approach discussion in Chapter 1.1, a first class virtual kernel does not consume unnecessary resources and thus allows building more complex tests on the same machine.

The setup is illustrated in Figure 6. We run three processes. One of them acts as a router between two subnets. The two other processes have the router process configured as their default gateway. The tap driver is used as the backend for all interfaces and the subnets are bridged together using the host's *bridge* driver. As tap and bridge do not depend on hardware, they can be dynamically created on the host to suit the test setup needs.

Since packet routing is done in the kernel, the user portion of our router process is very simple. It merely has to configure the relevant interfaces and not accidentally exit while the kernel library does the forwarding work. The router's program body is illustrated in Figure 7.

To test that our setup is functional, we run an echo daemon as one of the endpoint processes and send packets to it from another endpoint. The measured RTT for this setup is 0.16 ms. For a comparison, pinging an operating system running inside qemu on the same machine is 0.24 ms. However, the virtual machine ping has one less hop than the rump setup and ping is handled on a lower level in the networking stack, so in reality the difference is much higher in favor of rump (Chapter 5).

Since we are not interested in transporting packets to an outside network in this test, it would possible to replace the tap and bridge drivers with dedicated driver which only does interprocess communication. This would allow building even bigger simulated networks within a single test host.

[1] The implementation described in the cited paper was done on an embedded operating system called MMLite, which uses the BSD networking stack. The "NTUbig" operating mode of MMLite inspired the libsockin library described in this paper.

Figure 7: Router main program

```
int
main(int argc, char *argv[])
{

        rump_init();
        configure_interfaces();
        pause();

        return 0;
}
```

4.3 Virtualization

Network stack virtualization allows multiple networking stack instances to coexist on the system and be used by various applications. While indirect virtual kernels are a case of multiple different networking stacks on the same machine, they are outside the direct use of processes running on the host system.

On the other hand, the FreeBSD clonable network stack [14] offers instances of the same native networking stack for applications running on a FreeBSD host.

Like indirect virtual kernels, our approach allows running multiple different fully isolated networking stacks on a single host, but the solution is directly available to processes on the host. However, the decision to make use of it is dictated by userland policy instead of kernel policy, so in its current form it cannot be used to enforce isolation like the FreeBSD solution. Still, there is no fun-

damental reason policy enforcement cannot be done as future work for the rump networking technology.

When we evaluate the performance in Chapter 5, we run the NetBSD 5.99.7 TCP/IP stack both on NetBSD 4.0 and NetBSD 5.0.

4.4 Potential Application: TorVM

Tor [4] is a TCP anonymizer which encrypts and routes traffic through a network of nodes. For use, it can either be configured on an application-to-application basis, or by using set of a packet filter rules to redirect traffic to another local machine which handles Tor processing.

TorVM [12] is an application which provides transparent Tor for all applications on a host. Instead of complex packet filter rules, TorVM aims to simplify the process: it dropped in as an application onto the host machine and the default gateway is configured. TorVM handles all traffic going through the host by using a virtual machine to run and operating system for routing traffic.

Using rump networking it is possible to do the same with an application. No virtual machine or full operating system installation within the virtual machine is necessary. An implementation has not yet been done, but the application scenario exists. Using rump networking instead of a virtual machine would provide the potential benefits of a leaner installation. It also provides better performance, as is seen in the next chapter.

5 Performance

To evaluate performance, we measure three different setups for the roundtrip time and throughput. The roundtrip is measured with a simple ping program, which just waits for a packet and sends a response back. Throughput is measured using ttcp modified for rump. RTT is always reported for UDP and throughput for TCP. We measured both protocols in both cases but did not find a difference and therefore do not report both protocols for both tests.

The machines used for the test were a 2GHz Core2Duo laptop running NetBSD 5.0_BETA and a uniprocessor 1.6GHz Pentium 4 PC server running NetBSD 4.0_STABLE. The NetBSD 5.99.7 networking stack was used as the backend of rump networking.

All tests are reported from the perspective of the laptop host OS: ”out” means a test initiated by the laptop and ”in” means a test initiated by the peer. In tests two and three we measure each combination of host networking and rump networking. In the bars group’s legend the first component always denotes the networking used on the laptop and the second denotes the server. For example, ”reg-rump” means that the laptop was not using rump and the server was using rump networking.

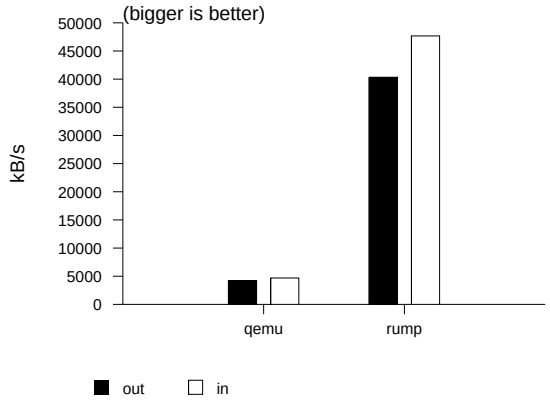

Figure 8: Virtualized Throughput

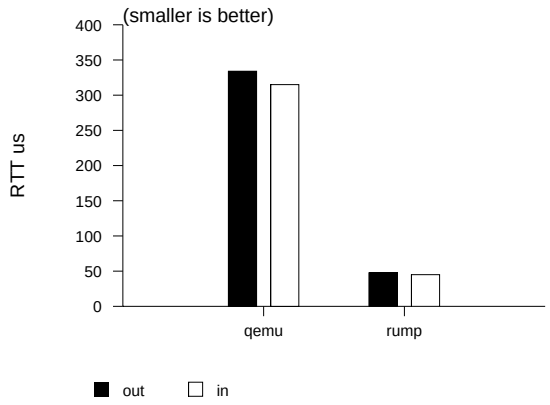

Figure 9: Virtualized RTT

1. **Virtualized**: A regular application using the host networking stack is run against both the application running in qemu and an application using the full NetBSD 5.99.7 TCP/IP stack in userspace. These tests were run on the Core2Duo laptop. The results are presented in Figures 8 and 9.

2. **Sockin**: only the sockin component of rump networking is used. These tests were performed over a 54Mbps wireless network. Care was taken to ensure that the wireless network remained at 54Mbps throughout the test. The results are presented in Figures 10 and 11.

3. **Full TCP/IP**: this test measures the performance of using the full NetBSD 5.99.7 TCP/IP networking stack in userspace. The tests were run over a 100Mbit LAN. The results are presented in Figures 12 and 13.

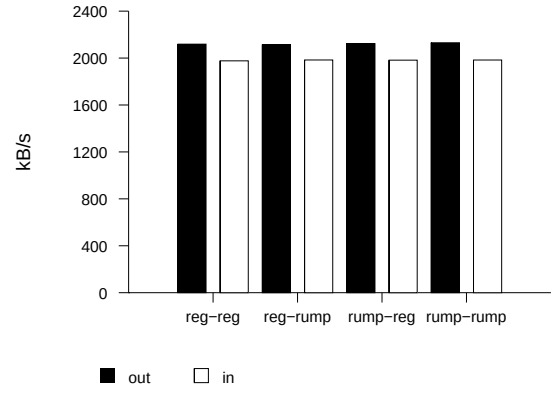

Figure 10: Sockin WLAN Throughput

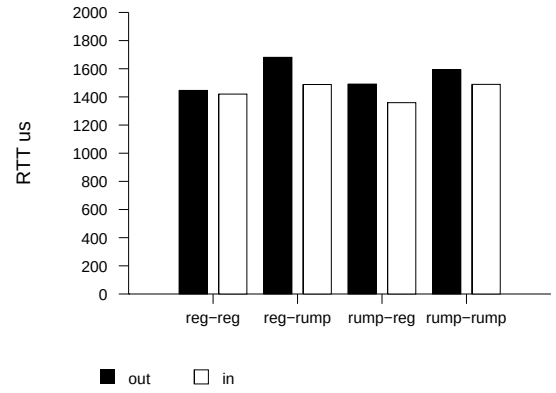

Figure 11: Sockin WLAN RTT

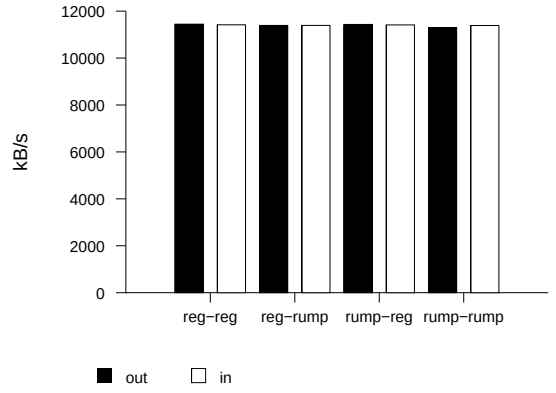

Figure 12: Userspace TCP/IP 100Mbit Throughput

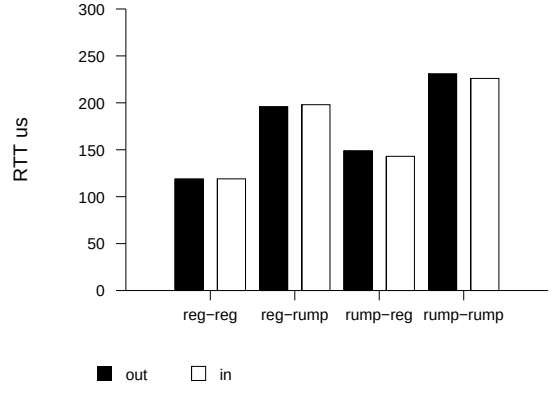

Figure 13: Userspace TCP/IP 100Mbit RTT

5.1 Analysis

Virtualize: When put against qemu, the rump solution is around 10 times faster than a full machine virtualization solution. This is expected, since the machine emulator has to first be scheduled within the host OS, after which the emulator must schedule the appropriate process. The majority of the performance penalty appears to come from the emulator scheduling the application, since as was measured in Chapter 4.2, icmp ping on an OS running in qemu is relatively much cheaper.

Sockin: This test measured the slowdown due to using the kernel sockets layer. On a wireless network, the effect to throughput is not visible. The latency figures are slightly elevated for sockin, suggesting a slight increase in overhead. However, as the wireless network is a very jittery environment, latency should be expected.

An interesting fact to note is that outgoing throughput is consistently faster in all cases, but outgoing latency is slower. It is unknown what causes this effect.

Full TCP/IP: The throughput on the network is equal for both cases. However, in the latency measurements rump networking is clearly slower. The increased RTT for userspace TCP/IP is much greater if used in the server than if used on the laptop. Since the server is uniprocessor and has only one CPU, we can conclude that the userspace networking stack causes extra CPU demand.

5.2 Future Directions

Even though currently already performant, there is still more work than can be done to make the system perform better. Currently, all memory allocation is relegated to libc `malloc()`. As the networking stack does many allocations and releases of known-size mbufs and mbuf clusters, allocating these with malloc instead of from pools is expensive. Making it possible to use the kernel pool allocator in rump is a short-term future goal. This will make virtualized rump networking less CPU-hungry and bring the latency overhead of full TCP/IP down.

6 Conclusions

This paper presented rump networking: the use of kernel networking code as part of userspace applications through the means of a virtual userspace kernel using the Runnable Userspace Meta-Program (*rump*) framework available in NetBSD. This makes the kernel networking stack run in a regular application process. The approach can be used for debugging, testing, development, virtualization and special applications.

The performance of rump networking for the NetBSD 5.99.7 TCP/IP stack was compared against the same networking stack running in a qemu virtual machine and measured to be roughly 10 times faster. When measured against the kernel stack on a LAN, the throughput was the same but latency increased by 25-50% (around 50μs). Possible performance improvements were discussed in the form of using the kernel pool allocator for memory allocation instead of libc malloc.

Rump networking is a first class virtualized kernel service, i.e. directly a part of an application. This makes debugging and development easier, since the networking stack can be controlled directly from a host operating system process with function calls. We examined how to build a test setup to test IP routing within a single machine and implemented the setup.

The rump networking is accessed based on an application policy: only system calls with the rump_sys-prefix are executed in the rump virtual kernel. This means that the application must at some level be aware of the rump networking stack. This typically means a wrapper library or editing the application source code. Future work includes making these policy decisions transparent for the application and enforcing them in the kernel.

Availability

An experimental version of rump networking will be available in NetBSD 5.0 (as of writing this Release Candidate 1 is out). Work on rump continues in the current development branch. This paper described the status in the development branch at the time of writing.

The source code is available for examination and use from the NetBSD source repository. The core networking library is available from the directory `src/sys/rump/librump/rumpnet` and the individual networking libraries are available from `src/sys/rump/net/lib`. The rump web page [7] contains information on how to use them.

Acknowledgments

The Finnish Cultural Foundation provided funding.

References

[1] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[2] COSTELLO, A. M., AND VARGHESE, G. Redesigning the BSD Callout and Timer Facilities. Tech. Rep. WUCS-95-23, Washington University, 1995.

[3] DIKE, J. A user-mode port of the Linux kernel. In *ALS'00: Proc. of the 4th Annual Linux Showcase & Conference* (2000).

[4] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 303–320.

[5] ECONOMOPOULOS, A. A Peek at the DragonFly Virtual Kernel, 2007.

[6] ELY, D., SAVAGE, S., AND WETHERALL, D. Alpine: A User-Level infrastructure for network protocol development. In *Proc. of USENIX Symp. on Internet Technologies and Systems* (2001), pp. 171–184.

[7] KANTEE, A. Runnable Userspace Meta Programs. http://www.NetBSD.org/docs/rump/.

[8] KANTEE, A., AND HELANDER, J. Implementing Lightweight Routing for BSD TCP/IP. In *Proc. of 5th EuroBSDCon* (2006).

[9] MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. *The design and implementation of the 4.4BSD operating system*. Addison Wesley, 1996.

[10] NETBSD LIBRARY FUNCTIONS MANUAL. *proplib – property container object library*, 2007.

[11] NETBSD PROJECT. http://www.NetBSD.org/.

[12] PECK, M., AND WILLIAMS, K. A Tor Virtual Machine Design and Implementation, 2008.

[13] WRIGHT, G. R., AND STEVENS, W. R. *TCP/IP Illustrated, Volume 2*. Addison Wesley, 1995.

[14] ZEC, M. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *USENIX Annual Technical Conference, FREENIX Track* (2003), pp. 137–150.