Rump Kernels

No OS? No Problem!

ANTTI KANTEE, JUSTIN CORMACK

Antti Kantee got bitten by the OS bug when he was young, and is still searching for a patch. He has held a NetBSD commit bit for fifteen years and for the previous seven of them he has been working on rump kernels. As a so-called day job, Antti runs a one-man "systems design and implementation consulting" show.
pooka@fixup.fi

Justin Cormack accidentally wandered into a room full of UNIX workstations at MIT in the early 1990s and has been using various flavors ever since. He started working with rump kernels last year and recently acquired a NetBSD commit bit. He is generally found in London these days.
justin@myriabit.com

In the modern world, both virtualization and plentiful hardware have created situations where an OS is used for running a single application. But some questions arise: Do we need the OS at all? And by including an OS, are we only gaining an increased memory footprint and attack surface? This article introduces rump kernels, which provide NetBSD kernel drivers as portable components, allowing you to run applications without an operating system.

There is still a reason to run an OS: Operating systems provide unparalleled driver support, e.g., TCP/IP, SCSI, and USB stacks, file systems, POSIX system call handlers, and hardware device drivers. As the name *rump* kernel suggests, most of the OS functionality not related to drivers is absent, thereby reducing a rump kernel's footprint and attack surface.

For example, a rump kernel does not provide support for executing binaries, scheduling threads, or managing hardware privilege levels. Yet rump kernels can offer a complete enough environment to support unmodified POSIXy applications on top of them (Figure 1). In this article, we explain how rump kernels work and give you pointers on how you can benefit from them in your projects.

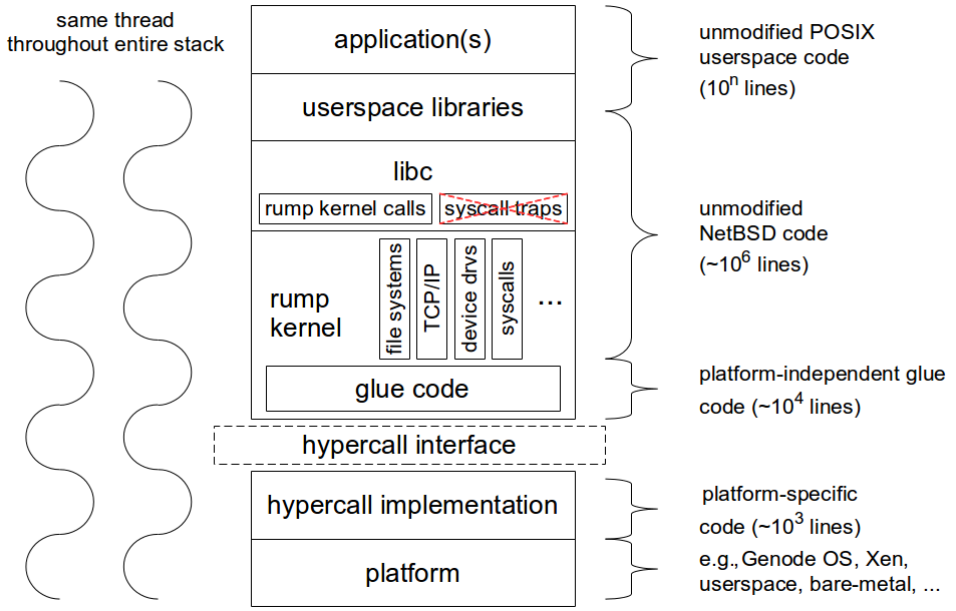

Figure 1: Rump kernels provide file system, network, and other driver support and run on bare metal systems or hypervisors by making use of a hypercall interface. In the depicted case, rump kernels provide the support necessary for running applications without requiring a full OS.

If you are building an OS-less system, why use rump kernels as your drivers? Rump kernels consist of a pool of roughly one million lines of unmodified, battle-hardened NetBSD kernel drivers running on top of a documented interface. The implementation effort for the interface, should your given platform not already be supported, is approximately 1,000 lines of C code. As the old joke goes, writing a TCP/IP stack from scratch over the weekend is easy, but making it work on the real-world Internet is more difficult. A similar joke about porting an existing TCP/IP stack out of an OS kernel most likely exists. Furthermore, the TCP/IP stack is only one driver, so you need plenty of spare weekends with the "roll your own" approach. The, we daresay, magic of rump kernels working in the real world stems from unmodified code. Driver bugs in operating systems have been ironed out over years of real-world use. Since rump kernels involve no porting or hacking of individual drivers, no new bugs are introduced into the drivers. The more unmodified drivers you use, the more free maintenance of those drivers you get. We are not suggesting that OS kernel drivers are optimal for all purposes but that it is easy to start with profiling and optimizing a software stack that works right off the bat.

In related work, there are a number of contemporary projects focusing on avoiding the overhead and indirection of the OS layer in the cloud: for example, MirageOS, OSv, and Erlang-on-Xen. But our goal with rump kernels is different. We aim to provide a toolkit of drivers for any platform instead of an operating environment for cloud platforms. In that sense, rump kernels can be thought of being like lwIP [1], except the scope is beyond networking (and the TCP/IP stack is larger). That said, we do also provide complete support for rump kernels on a number of platforms, including POSIXy user space and Xen. We also integrate with a number of other frameworks. For example, drivers are available for using the TCP/IP stack offered by rump kernels with user space L2 packet frameworks such as netmap, Snabb Switch, and DPDK.

The beef of rump kernels, pun perhaps intended, is allowing third-party projects access to a pool of kernel-quality drivers, and Genode OS [2] has already made use of this possibility. Although there are other driver toolkits (e.g., DDEKit [3]), we claim that rump kernels are the most complete driver kit to date. Furthermore, support for rump kernels is directly included in the NetBSD source tree. One example of the benefit of in-tree support is that in case of an attractive new driver hitting the NetBSD tree, there is no waiting for someone to roll the driver kit patches forwards, backwards, and sideways. You can simply use any vintage of NetBSD as a source of rump kernels.

We will avoid going into much technical detail in this article. The book [4] provides more detailed descriptions for interested parties.

History

Rump kernels started in 2007 as a way to make debugging and developing NetBSD kernel code easier. Developing complex kernel code usually starts out by sketching and testing the central pieces in the comfort of user space, and only later porting the code to the kernel environment. Despite virtual machines and emulators being plentiful in this age, the user space approach is still used, suggesting that there is something which makes user space a simpler platform to work with.

Even though rump kernels started out as running kernel code in user space, they were never about running the full OS there, because a user space OS is fundamentally not different from one running in a virtual machine and introduces unnecessary complexity for development purposes. From the beginning, rump kernels were about bringing along the minimum amount of baggage required to run, debug, examine, and develop kernel drivers. Essentially, the goal was to make developing drivers as easy as in user space, but without having to port kernel code to user space and back. From that desire a very significant feature of the rump kernel arose: It was necessary that exactly the same driver code ran both in debug/development mode and in the NetBSD kernel, and hacks like `#ifdef TESTING` were not permitted.

Problems related to development, testing, and debugging with rump kernels were more or less addressed by 2011, and the fundamental concepts of rump kernels have remained unchanged since then. Then a new motivation for rump kernels started emerging. The effort to make kernel drivers run in user space had essentially made most kernel drivers of NetBSD portable and easy to integrate into other environments. Adding support for platforms beyond user space was a simple step. The goal of development shifted to providing reusable drivers and a supporting infrastructure to allow easy adaptation. Testing, of course, still remains a central use case of rump kernels within NetBSD, as does, for example, being able to run the file system drivers as user-space servers.

Making Rump Kernels Work

Rump kernels are constructed out of components. The drivers are first built for the target system as libraries, and the final runtime image is constructed by linking the component-libraries together, along with some sort of application, which controls the operation of the drivers (see Figure 1 for an example). Notably, the application does not have to be a POSIXy user-space application. For example, when using rump kernels as microkernel-style user space file servers, the "application" is a piece of code that reads requests from the FUSE-like user space file systems framework and feeds them into the rump kernel at the virtual file system layer.

The starting point for coming up with the components was a monolithic kernel operating system. The problem is that we want to use drivers without bringing along the entire operating system kernel. For example, let us assume we want to run a Web server serving dynamically created content, perhaps running on an Internet-of-Things device. All we need in the rump kernel is the TCP/IP stack and sockets support. We do not need virtual memory, file systems(!), or anything else not contributing to the goal of talking TCP. As the first step, we must be able to "carve" the TCP/IP stack out of the kernel without bringing along the entire kitchen-sinky kernel, and give others an easy way to repeat this "carving." Second, we must give the rump kernel access to platform resources, such as memory and I/O device access. These issues are solved by the anykernel and the rump kernel hypercall interface, respectively.

Anykernel

The enabling technology for rump kernels in the NetBSD codebase is the anykernel architecture. The "any" in "anykernel" is a reference that it is possible to use drivers in any configuration: monolithic, microkernel, exokernel, etc. If you are familiar with the concept of kernel modules, you can think of the anykernel roughly as an architecture which enables loading kernel modules into places beyond the original OS.

We realize the anykernel by treating the NetBSD kernel as three layers: base, factions, and drivers. Note, this layering is *not* depicted in Figure 1, although one might replace the "rump kernel" box with such layers. The base contains fundamental routines, such as allocators and synchronization routines, and is present in every rump kernel. All other kernel layers are optional, although including at least some of them makes a rump kernel instance more exciting. There are three factions and they provide basic support routines for devices, file systems, and networking. The driver layer provides the actual drivers such as file systems, PCI drivers, firewalls, software RAID, etc. Notably, in addition to depending on the base and one or more factions, drivers may depend on other drivers and do not always cleanly fit into a single faction. Consider NFS, which is half file system, half network protocol. To construct an executable instance of a rump kernel supporting the desired driver, one needs the necessary dependent drivers (if any), a faction or factions, and the base.

Let us look at the problem of turning a monolithic kernel into an anykernel in more detail. Drivers depend on bits and pieces outside of the driver. For example, file system drivers generally depend on at least the virtual file system subsystem in addition to whichever mechanism they use to store the file system contents. Simply leaving the dependencies out of the rump kernel will cause linking to fail, and just stubbing them out as null functions will almost certainly cause things to not work correctly. Therefore, we must satisfy all of the dependencies of the drivers linked into the rump kernel.

Popular myth would have one believe that a monolithic kernel is so intertwined that it is not possible to isolate the base, factions, and drivers. The myth was shown to be false by the "come up with a working implementation" method.

Honestly speaking, there is actually not much "architecture" to the anykernel architecture. One could compare the anykernel to an SMP-aware kernel, in which the crux is not coming up with the locking routines, but sprinkling their use into the right places. Over the monolithic kernel, the anykernel is merely a number of changes that make sure there are no direct references where there should not be any. For example, some source modules that were deemed to logically belong to the base contained references to file system code. Such source modules were split into two parts, with one source module built into the base and the split-off source module built into the file system faction. In monolithic kernel mode, both source modules are included.

In addition, cases where a rump kernel differs from the full-blast monolithic kernel may require glue code to preserve correct operation. One such example revolves around threads, which we will discuss in the next section; for now, suffice it to say that the method the monolithic kernel uses for setting and fetching the currently running thread is not applicable to a rump kernel. Yet we must provide the same interface for drivers. This is where glue code kicks in. The trick, of course, is to keep the amount of glue code as small as possible to ensure that the anykernel is maintainable in NetBSD.

The anykernel does not require any new approaches to indirection or abstraction, just plain old C linkage. Sticking with regular C is dictated by practical concern; members of an operating system project will not like you very much if you propose indirections that hurt the performance of the common case where the drivers are run in the monolithic kernel.

Hypercalls

To operate properly, the drivers need access to back-end resources such as memory and I/O functions. These resources are provided by the implementation of the rump kernel hypercall interface, rumpuser [5]. The hypercall interface ties a rump kernel to the platform the rump kernel is run on. The name hypercall interface is, you guessed it, a remnant of the time when rump kernels ran only in user space.

We assume that the hypercall layer is written on top of a platform in a state where it can run C code and do stack switching. This assumption means that a small amount of bootstrap code needs to exist in bare-metal type environments. In hosted environments, e.g., POSIX user space, that bootstrap code is implicitly present.

Very recently, we learned about the Embassies project [6], where one of the goals is to come up with a minimal interface for running applications and implement a support for running POSIX programs on top of that minimal interface. This is more or less what rump kernels are doing, with the exception that we are running kernel code on top of our minimal layer. POSIX applications, then, run transitively on top of our minimal interface by going through the rump kernel. Interestingly, the rump kernel hypercall interface and the Embassies minimal interface for applications are almost the same, although, at least to our knowledge, they were developed independently. The convenient implication of interface similarity is the ability to easily apply any security or other analysis made about Embassies to the rump kernel stack.

Fundamental Characteristics

We present the fundamental technical characteristics of rump kernels in this section. They are written more in the form of a dry list than a collection of juicy anecdotes and use cases. We feel that presenting the key characteristics in a succinct form will give a better understanding of both the possibilities and limitations of the rump kernel approach.

A rump kernel is always executed by the host platform. The details, including how that execution happens, and how many concurrent rump kernel instances the platform can support, vary on the platform in question. For user space, it's a matter of executing a binary. On Xen, it's a matter of starting a guest domain. On an embedded platform, most likely the bootloader will load the rump kernel into memory, and you would just jump to the rump kernel entry point.

The above is in fact quite normal; usually operating systems are loaded and executed by the platform that hosts them, be it hardware, virtual machine, or something else. The difference comes with application code. A kernel normally has a way of executing applications. Rump kernels contain no support for executing binaries to create runtime processes, so linking and loading the application part of the rump kernel software stack is also up to the host. For simplicity and performance, the application layer can be bundled together with the rump kernel (see, e.g., Figure 1). In user space, it is also possible to run the rump kernel in one process, with one or more applications residing in other processes communicating with the rump kernel (so-called "remote clients"). In both cases the applications are still linked, loaded, and executed by the host platform.

The notion of a CPU core is fictional. You can configure the number of "cores" as you wish, with some restrictions, such as the number must be an integer >0. For a rump kernel, the number of cores only signifies the number of threads that can run concurrently. A rump kernel will function properly no matter what the mapping between the fictional and physical cores is. However, if performance is the goal, it is best to map a rump kernel instance's fictional cores 1:1 to physical cores, which will allow the driver code to optimize hardware cache uses and locking.

Rump kernels do not perform scheduling. The lack of thread scheduling has far-reaching implications, for example:

- Code in a rump kernel runs on the platform's threads—nothing else is available. Rump kernels therefore also use the platform's thread-scheduling policy. The lack of a second scheduler makes rump kernels straightforward to integrate and control, and also avoids the performance problems of running a thread scheduler on top of another thread scheduler.
- Synchronization operations (e.g., mutex) are hypercalls because the blocking case for synchronization depends on invoking the scheduler. Notably, hypercalls allow optimizing synchronization operations for the characteristics of the platform scheduler, avoiding, for example, spinlocks in virtualized environments.

A less obvious corollary to the lack of a scheduler is that rump kernels use a "CPU core scheduler" to preserve a property that code expects: no more than one thread executing on a core. Maintaining this property in rump kernels ensures that, for example, passive synchronization (e.g., RCU, or read-copy-update) and lock-free caches continue to function properly. Details on core scheduling are available in the book [4].

Since core scheduling is not exposed to the platform scheduler, there are no interrupts in rump kernels, and once a rump kernel core is obtained, a thread runs until it exits the rump kernel or blocks in a hypercall. This run-to-completion mode of operation is not to be confused with a requirement that the platform scheduler must run the thread to completion. The platform scheduler is free to schedule and unschedule the thread running in a rump kernel as it pleases. Although a rump kernel will run correctly on top of any thread scheduler you throw under it, there are performance advantages to teaching the platform scheduler about rump kernels.

Rump kernels do not support, use, or depend on virtual memory. Instead, a rump kernel runs in a memory space provided by the platform, be it virtual or not. The rationale is simplicity and portability, especially coupled with the fact that virtual memory is not necessary in rump kernels. Leaving out virtual memory support saves you from having to include the virtual memory subsystem in a rump kernel, not to mention figuring out how to implement highly platform-dependent page protection, memory mapping, and other virtual memory-related concepts.

The more or less only negative effect caused by the lack of virtual memory support is that the mmap() system call cannot be fully handled by a rump kernel. A number of workarounds are possible for applications that absolutely need to use mmap(). For example, the bozohttpd Web server uses mmap() to read the files it serves, so when running bozohttpd on top of a rump kernel, we simply read the mmap'd window into memory at the time the mapping is made instead of gradually faulting pages in. A perfect emulation of mmap() is hard to achieve, but one that works for most practical purposes is easy to achieve.

Machine (In)Dependencies

Rump kernels are platform-agnostic, thanks to the hypercall layer. But can rump kernels be run literally anywhere? We will examine the situation in detail.

One limitation is the size of the drivers. Since NetBSD drivers are written for a general purpose OS, rump kernels are limited to systems with a minimum of hundreds of kB of RAM/ROM. One can of course edit the drivers to reduce their size, but by doing so one of the core benefits of using rump kernels will be lost: the ability to effortlessly upgrade to later driver versions in order to pick up new features and bug(fixe)s.

As for the capabilities of the processor itself, the only part of the instruction set architecture that permeates into rump kernels is the ability to perform cache-coherent memory operations on multiprocessor systems (e.g., compare-and-swap). In a pinch, even those machine-dependent atomic memory operations can be implemented as hypercalls—performance implications notwithstanding—thereby making it possible to run rump kernels on a generic C machine.

To demonstrate their machine independence, rump kernels were run through a C->Javascript compiler so that it was possible to execute them in Web browsers. Running operating systems in browsers previously has been accomplished via machine emulators written in Javascript, but with rump kernels the kernel code went native. If you have always wondered what the BSD FFS driver looks like when compiled to Javascript and wanted to single-step through it with Firebug, your dreams may have come true. The rest of us will probably find more delight in being amused by the demo [7] for a few minutes. And, no, the NetBSD kernel did not and still does not support the "Javascript ISA," but rump kernels do.

So, yes, you can run rump kernels on any platform for which you can compile C99 code and which has a minimum of some hundreds of kilobytes of RAM/ROM.

Virtual Uniprocessor and Locking

Avoiding memory bus locks is becoming a key factor for performance in multiprocessor environments. It is possible to omit memory bus locks almost entirely for rump kernels configured to run with one fictional core, regardless of the number of physical cores visible to the platform. This optimization is based on the property of the rump kernel CPU core scheduler. Since there can be at most one thread running within the rump kernel, there is no need to make sure that caches are coherent with other physical cores, because no other physical core can host a thread running in the same rump kernel. Appropriate memory barriers when the rump kernel core is reserved and released are enough. The fastpath for locking becomes a simple variable check and assignment that can fully be handled within the rump kernel. Only where the lock is already held does a hypercall need to be made to inform the scheduler.

This locking scheme can be implemented in a single file without touching any drivers. In the spirit of the project, the name of the Uniprocessor locking scheme was decided after careful consideration: *locks_up*. A scientific measurement of a POSIXy application creating and removing files on a memory file system showed a more than 30% performance increase with locks_up. The actual benefit for real-world applications may be less impressive.

From Syscalls to Application Stacks

First, we introduce some nomenclatural clarity. Since there are no hardware privilege levels or system traps in rump kernels, there are strictly speaking no system calls either. When we use the term "system call" or "syscall" in the context of rump kernels, we mean a routine which performs the service that would normally be executed via a kernel trap.

From nearly the beginning of this project, rump kernels have supported NetBSD-compatible system call interfaces. Compatibility exists for both API and ABI, apart from the distinction that rump kernel syscalls were prefixed with "rump_sys" to avoid symbol collisions with libc when running in user space. ABI compatibility meant that in user space it was possible to `LD_PRELOAD` a hijacking library so that most system calls were handled by the host, but some system calls—e.g., ones related to sockets—could be handled by rump kernels.

On a platform without an OS, this approach of course does not work: There is no OS that can handle a majority of the system calls. The solution was simple (see Figure 1): we took NetBSD's libc and built it without the syscall bits that caused kernel traps. We then removed the "rump_sys" prefix for the rump kernel syscall handlers, because there was no host libc to conflict with. Regular user-space libraries—i.e., everything apart from libc and libpthread—and applications require no modification to function on top of a rump kernel; they think they are running on a full NetBSD system.

Among the three factions, rump kernels currently support roughly two-thirds, or more than 200, of the system calls offered by NetBSD. Some examples of applications tested to work out-of-the-box on top of a rump kernel include thttpd, the LuaJIT compiler, and wpa_supplicant.

Interestingly, getting the full application stack working in user space required more effort than getting it to work in an environment without a host OS. This is because user space gets crowded: The rump kernel stack provides a set of symbols that can, and almost certainly will, conflict with the hosting OS's symbols. However, it turns out that with judicious symbol renaming and hiding it is possible to avoid conflicting names between the host OS and the rump kernel stack. Having the full application stacks work in user space allows you to compile and run NetBSD-specific user space code (e.g., ifconfig) against rump kernels on other operating systems. Listing 1 illustrates this in more detail.

Trying It Out

The easiest way to familiarize yourself with rump kernels is to do it in the comfort of user space by using the buildrump.sh script. Clone the repository at http://repo.rumpkernel.org/buildrump.sh.git and, on a POSIXy open source operating system, run:

```
./buildrump.sh
```

When executed without parameters, the script will fetch the necessary subset of the NetBSD source tree and build rump kernel components and the POSIXy user space implementation of the hypercall interface. Follow the build by running a handful of simple tests that check for example file system access, IPv4/IPv6 routing, and TCP termination. Running these tests under GDB in the usual fashion—buildrump.sh builds everything with debugging symbols by default—and single-stepping and using breakpoints is an easy way to start understanding how rump kernels work.

Since rump kernel stacks work the same way in user space as they do on an embedded IoT device, once you learn one platform you've more or less learned them all. The flipside of the previous statement also applies: When you want to debug some code for your embedded device, you can just debug the code in user space, presence of hardware devices notwithstanding.

Also make sure to note that if your host is running on desktop/server hardware of a recent millennium, the bootstrap time of a rump kernel is generally on the order of 10 ms.

Listing 1 offers an idea of the component-oriented quality of rump kernels and shows how easily you can configure them as long as you are familiar with standard UNIX tools. Further up-to-date instructions targeting more specific use cases are available as tutorials and how-tos on wiki.rumpkernel.org.

Run a rump kernel server accepting remote requests, set up client programs to communicate with it, and check the initial network configuration.

```
rumpremote (NULL)$ rump_server -lrumpnet_netinet
    -lrumpnet_net -lrumpnet unix://ctrlsock
rumpremote (NULL)$ export RUMP_SERVER=unix://
    ctrlsock
rumpremote (unix://ctrlsock)$ ifconfig -a
lo0: flags=8049 mtu 33648
    inet 127.0.0.1 netmask 0xff000000
```

Oops, we want IPv6, too. Let's start another rump kernel with IPv6, listening to requests at a slightly different address.

```
rumpremote (unix://ctrlsock)$ rump_server -lrumpnet_
    netinet6 lrumpnet_netinet -lrumpnet_net
    -lrumpnet unix://ctrlsock6
rumpremote (unix://ctrlsock)$ export RUMP_SERVER=
    unix://ctrlsock6
rumpremote (unix://ctrlsock6)$ ifconfig -a
lo0: flags=8049 mtu 33648
    inet6 ::1 prefixlen 128
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    inet 127.0.0.1 netmask 0xff000000
```

Better. We check that the original is still without IPv6, and see which file systems are mounted in the new one.

```
rumpremote (unix://ctrlsock6)$ env
    RUMP_SERVER=unix://ctrlsock ifconfig -a
lo0: flags=8049 mtu 33648
    inet 127.0.0.1 netmask 0xff000000
rumpremote (unix://ctrlsock6)$ mount
mount: getmntinfo: Function not implemented
```

Oops, we did not include file system support. We will halt the second server and restart it with file system support.

```
rumpremote (unix://ctrlsock6)$ halt
rumpremote (unix://ctrlsock6)$ rump_server -lrumpnet_
    netinet6 -lrumpnet_netinet -lrumpnet_net
    -lrumpnet -lrumpvfs unix://ctrlsock6
rumpremote (unix://ctrlsock6)$ mount
rumpfs on / type rumpfs (local)
rumpremote (unix://ctrlsock6)$
```

Listing 1: Example of rump kernels running in user space. The process rump_server contains kernel components. The utilities we use contain the application layers of the software stack. In user space, the two can communicate via local domain sockets. This model allows for very natural use. The output was captured on Ubuntu Linux. $PATH has been set so that NetBSD utilities that are running on top of the rump kernel stack are run.

Conclusion

We present rump kernels, a cornucopia of portable, componentized kernel-quality drivers such as file systems, networking drivers, and POSIX system call handlers. Rump kernels rely on the anykernel architecture inherent in NetBSD, and can be built from any vintage of the NetBSD source tree. The technology is stable, as far as that term can be used to describe anything related to operating system kernel internals, and has been developed in NetBSD since 2007.

Everything we described in this article is available as BSD-licensed open source via **rumpkernel.org**. Pointers to usual community-type elements for discussing use cases and contributions are also available from rumpkernel.org. We welcome your contributions.

References

[1] lwIP, a lightweight open source TCP/IP stack: http://savannah.nongnu.org/projects/lwip/.

[2] Genode Operating System Framework: http://genode.org/.

[3] DDEKit and DDE for Linux: http://os.inf.tu-dresden.de/ddekit/.

[4] *The Design and Implementation of the Anykernel and Rump Kernels*: http://book.rumpkernel.org/.

[5] Rump kernel hypercall interface manual page: http://man.NetBSD.org/cgi-bin/man-cgi?rumpuser++NetBSD-current.

[6] Embassies project: https://research.microsoft.com/en-us/projects/embassies/.

[7] Javascript rump kernel: http://ftp.NetBSD.org/pub/NetBSD/misc/pooka/rump.js/.