Implementing a Clonable Network Stack in the FreeBSD Kernel

Marko Zec

University of Zagreb, Faculty of Electrical Engineering and Computing

zec@tel.fer.hr

Abstract

Traditionally, UNIX operating systems have been equipped with monolithic network stack implementations, meaning all user processes have to cooperatively share a single networking subsystem. The introduction of the network stack cloning model enables the kernel to simultaneously maintain multiple independent and isolated network stack instances. Combined with forcible binding of user processes to individual network stacks, this concept can bring us a step closer to an efficient pseudo virtual machine functionality which opens new possibilities particularly in virtual hosting applications, as well as in other less obvious areas such as network simulation and advanced VPN provisioning. This article is focused on design, implementation and performance aspects of experimental clonable network stack support in the FreeBSD kernel.

1 Introduction

Implementing various models and levels of resource partitioning and protection has been in focus of operating systems research ever since the introduction of the multi-programming paradigm in early days of computing.

In the 1960s, IBM introduced the concept of virtual machines (VM). Each VM instance presented a close yet independent replica of the underlying physical machine which gave users the illusion of running their programs directly on the real hardware. VM also provided benefits like mutual isolation, protection and resource sharing, as well as the ability to run multiple independent flavors and configurations of operating systems (OS) simultaneously on the same physical machine.

On UNIX systems, user programs run in a simplified VM model. The OS kernel manages the allocation of protected virtual memory and schedules CPU cycles to user processes; however, the processes are not allowed to access any other hardware resources directly. Instead, the kernel provides an abstraction layer for accessing vital system resources, such as I/O devices, filesystems and network communication facilities. Traditionally, networking facilities in particular have been implemented monolithically within the kernel, meaning all the user processes have to cooperatively share the networking facilities and addressing space. Over the years this concept was completely sufficient for most of the common environments; however it presents some limits to certain emerging applications, most notably to more sophisticated scenarios for virtual hosting.

The model of clonable network stacks, presented in this article, allows multiple independent network stack instances to coexist within the OS kernel simultaneously. From the perspective of network communications, by associating groups of user processes to an individual network stack instance it is possible to achieve highly efficient *light* or *pseudo* virtual machine functionality. Each unique collection of network stack instance and group of associated user processes will further be referenced as a *virtual image*.

The fundamental difference between the traditional VM implementation and pseudo-VM or virtual image model is illustrated in Figure 1. In both cases each VM appears as an independent entity from the perspective of the outside network. However, in the traditional VM system each VM hosts an entire OS instance with dedicated partitions of hardware resources such as physical RAM, raw disks and I/O devices. All I/O operations, including those dealing with network interfaces, have to be either controlled or emulated by the VM monitor. Contrary to this approach, in a pseudo-VM system, a single OS running on physical hardware controls multiple independent network stack instances and associated user processes without the need to dedicate hardware resources to each pseudo-VM instance. This allows for significantly less overhead on the data path from applications to the physical

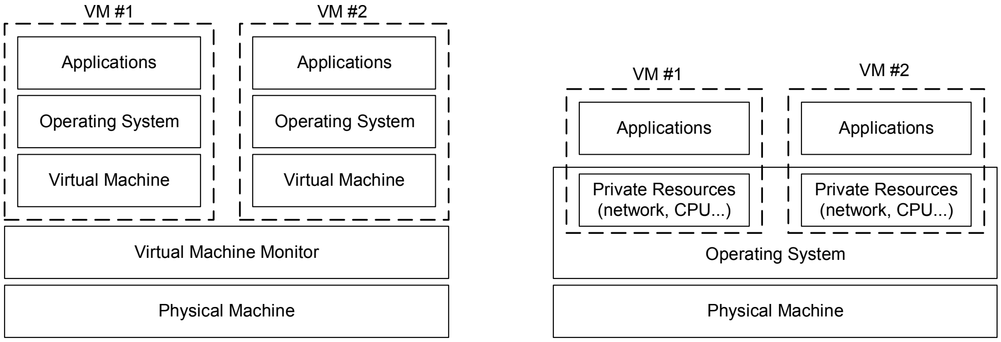

Figure 1: Traditional VM model (left) and pseudo-VM *virtual image* model (right)

network interfaces, as well as better utilization of other system resources, particularly CPU and RAM. On the other hand, the pseudo-VM system cannot run different OS flavors simultaneously and does not offer the level of mutual independence and protection available in the traditional VM model.

Although the described concept of virtual images and clonable network stacks is generic, this article is focused on its implementation in the FreeBSD kernel. With reasonable efforts and enough time, it should be possible to modify and extend other network stack implementations to offer such functionality, provided that the entire OS kernel source tree is available.

The rest of the article is organized as follows. Section 2 describes some previous work and models in partitioning system resources, with focus on network communication. Section 3 presents the basic design ideas behind the concept of network stack cloning. The implementation details are discussed in Section 4, followed by an overview of various performance implications presented in Section 5. Section 6 gives an example of how the management interface of clonable network stacks looks to the system administrator. Finally, Section 7 summarizes the properties of the clonable network stack infrastructure and outlines the directions for possible future development and research.

2 Previous work

The goal of running multiple instances of network protocol suite on a single physical machine can be accomplished in different ways. As mentioned in the previous section, one option is virtualization of the entire machine hardware and running multiple independent instances of fully self-contained OS images within the VM. Such an approach requires hardware virtualization support for efficient operation, which can usually be found only on large-scale and expensive system architectures, such as IBM's S/390 mainframe family. Software-based virtualization of hardware architectures lacking the inherent virtualization circuitry is also achievable. Such virtualization products are available both as OSS and commercial products, such as VMware [1]. However, this model introduces a significant bottleneck at emulating I/O operations in software, which results in performance degradation during data transfers to and from the network interfaces. Therefore, often this approach may not be suitable for heavy-duty network centric applications. Furthermore, the traditional VM model can lead to suboptimal use of other hardware resources; for instance, the VM monitor is typically unaware if an OS instance running within a VM no longer needs a particular memory page.

An alternative model, the *jail* [2] facility implemented in FreeBSD, provides the ability to partition the OS into multiple separated process groups with limited network addressing space. The kernel prevents user processes running in jailed environments from managing the processes and certain system resources outside their own jailed protection domain. All the jailed environments share the same network stack; however each jail is restricted to use a unique IP address, and cannot interfere with other network traffic. Creating *jailed* pseudo virtual machines in this manner has many potential uses; thus far the most popular one has been for providing highly efficient virtual machine services in Internet Service Provider environments. It should be noted that the standard *jail* architecture still uses a monolithic network stack. Therefore the *jails* do not maintain private instances of subsystems such as routing tables, traffic counters, packet filters and traffic shapers etc., so they must rely on the *master* OS environment to manage those facilities.

Several reports describe efforts to implement network stacks as specialized userland processes, with the focus in network simulation applications. Frameworks such as ENTRAPID [7] or Alpine [8] successfully accomplish their primary goals in network simulation, however at the cost of poor overall performance compared to the in-the-kernel network stack implementation on the same hardware. The Harvard network simulator [9] takes a different approach as it creates the illusion of having multiple independent kernel routing tables by providing transparent IP address remapping between user and kernel space. While the kernel still maintains a single routing table with unique (non-overlapping) entries, for each virtual node a translation map is maintained which has to be consulted on each userland-to-kernel network transaction. Despite such an approach offering notably better performance than the ENTRAPID and Alpine architectures, it is still significantly constrained by numerous translation lookups that have to be performed on each kernel-to-userland packet transition, and vice versa. However, a major advantage of the Harvard architecture over the other two mentioned simulator frameworks is the ability to use the existing UNIX network applications in a virtualized environment without any modifications.

Although none of the above network simulation frameworks have been designed for use in general-purpose or virtual hosting applications, some of their concepts and objectives were partially adopted in the implementation of the clonable network stacks model. This is particularly true for the idea of preserving full compatibility with the existing userland binaries running on a modified kernel, which was one of the important design goals in the experimental clonable network stack implementation.

3 Design concepts

By setting the goal to implement a system that would perform equally well in generic and virtual hosting, VPN packet routing/switching or network simulation applications, the idea was born to reuse an existing reliable but monolithic network stack implementation and extend it to support cloning in multiple independent and isolated instances.

To virtualize the entire network stack of the base OS, and not just the selected portions needed for certain applications, was the most important design decision. Further, it was decided to implement all the modifications entirely within the OS kernel. Such an implementation not only allows the new code to be highly efficient, but also provides a significant level of security and isolation between the virtualized environments needed in any virtual hosting application. If the virtualization extensions would be even partially implemented in userland, for example by replacing the standard system dynamic libraries, it would be left up to the userland programs whether or not they would comply with their designation in a virtualized network stack. Another problem that would arise by choosing such an approach is that it could only work transparently for dynamically linked programs. Statically linked programs would have to be recompiled, which would clearly violate the requirement for transparent API/ABI compatibility with the user programs.

Preserving the complete functionality of the base OS serving as a development platform, while making the network stack extensions as universal as possible, was another important design goal. It became apparent that it would be highly desirable to preserve complete application programming and binary interface (API/ABI) compatibility with the existing application and utility programs running on the base OS. Such an approach would ensure simplicity and speed in both code development and testing, as well as in later real-world applications. If the API were not preserved, the end result would probably be a highly specialized system tuned for a particular application, not a general-purpose one as desired. Another important objective was the preservation of the performance of the existing base system. Obviously, if the modified system noticeably underperforms the original it could probably still be useful for certain specific applications; however, in such case the general-purpose criteria could not be met.

The introduction of *virtual images* was a key concept on which the clonable network stack framework is based. Each virtual image is an isolated kernel entity with its own independent network stack instance. All network interfaces and all user processes in a virtualized system are supposed to be associated with a unique virtual image, as shown in Figure 2.

Upon system startup only one (default) virtual image exists in the system, which contains all the network interfaces and user processes. System administrators can later dynamically create new virtual images and associate real or pseudo network interfaces with them and run user processes in those environments. User processes running in one virtual image will be able to interact only with their own network stack instance, therefore only with the network traffic and interfaces that are associated with its own virtual image. The default virtual image is no exception to this rule. As the virtual images are logically organized in a hierarchical manner, the parent virtual image is allowed to spawn a new process in its child for management purposes. Obviously, the child is prohibited from managing its siblings and the parent virtual image.

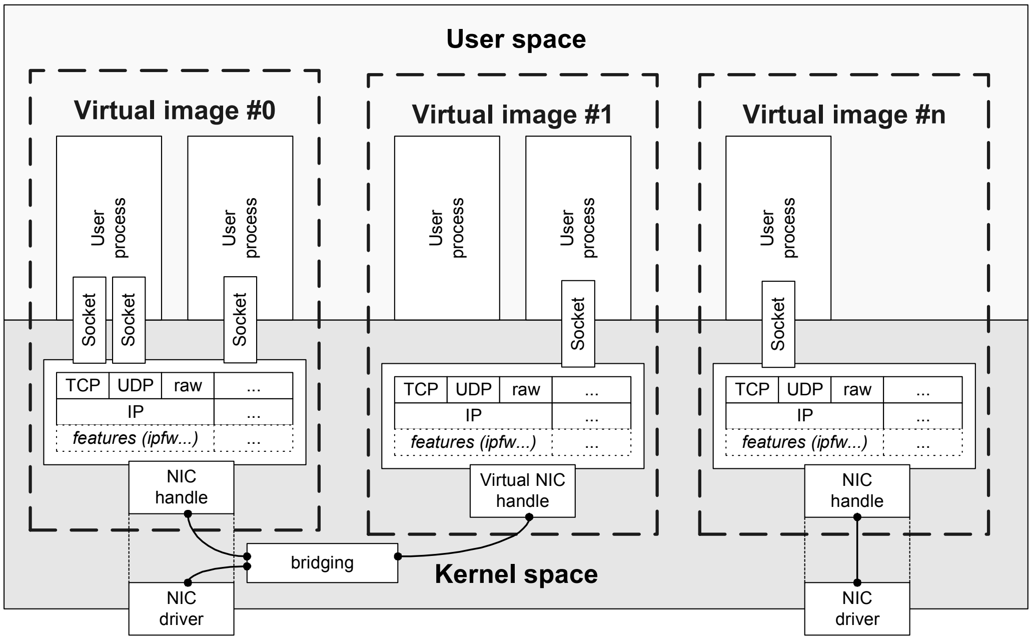

Figure 2: Operating system partitioned in virtual images

Although the high-level goal was to make virtual images as isolated and independent from each other as possible, they should be able to be interconnected if desired. This can be accomplished either via bridged virtual or physical Ethernet interfaces, or through virtual point-to-point channels constructed using the *netgraph* [10] framework. The former *bridging* method allows virtual images to become accessible from the outer world in a transparent manner, with almost neglectable overhead introduced by bridging code logic and processing.

4 Implementation

The experimental framework allowing multiple network stacks to be simultaneously active on a system was implemented as an extension to the FreeBSD 4.8 kernel. Each network stack instance was made fully independent of all others, so that each instance maintained its own private routing table, set of communication sockets and associated protocol control blocks etc. Later, the network stack virtualization experiment was extended to include optional networking facilities, such as packet filters, traffic shapers, bridging code and various sysctl [3] tunable variables controlling different aspects of network stack behavior.

4.1 Overview

As UNIX systems traditionally maintain only a single network stack within the kernel, an important design step has been selecting the optimal method for user processes to manage multiple network stacks. One option was modification of the standard Berkeley socket interface [4] by extending the argument lists with the network stack identifier. A variation of such approach was proposed in [11]. However, this concept has a significant drawback since it requires the existing user programs to be modified and recompiled in order to be able to run on the new / extended OS kernel. Therefore, an alternative approach was chosen. Each process control block (`struct proc`) in the kernel was transparently extended with a tag which associates it with a network stack instance. This tag is inherited by subsequent processes from its parents without any need for intervention from the programmer. Additionally, a new programming interface was introduced allowing a process to change its network stack association. This approach allowed for complete application programming and binary interface (API / ABI) compatibility to be preserved between the original and modified OS kernel, thus mitigating any need for modifications in the existing userland applications or utilities.

The described tagging of user processes was combined with the already available *jail* resource protection framework in FreeBSD, which resulted in user processes associated with one network stack being effectively invisible to the other processes running on the system, and vice versa. The newly developed framework, which combined different areas of resource protection mechanisms into one entity, successfully achieved the main properties of pseudo virtual machine functionality. The *virtual image* concept was further extended by including modifications to the CPU scheduler in such a way that each virtual image could be limited in average CPU usage so that runaway or maliciously constructed processes or groups of processes might be prevented from monopolizing and starving all the real CPU resources. This also allowed system load monitoring to be performed on a per virtual image basis, which provided more fine-grained control rather than accounting resource usage solely on physical machine level. Finally, a basic API for managing the virtual images was implemented, accompanied by a simple userland management utility.

The fundamental approach taken in implementation of the described modifications to the FreeBSD kernel was the introduction of a new `vimage` kernel structure, which serves as a container for all virtualized variables and symbols. Gradually, most of the global and static symbols used by network stack code were replaced by their equivalent counterparts residing in independent `vimage` structures. Network interface descriptors, which have traditionally been maintained in a single linked list, are now associated with `vimage` structures so that each network stack instance has its own list of network interfaces. Each network interface contains a pointer back to its `vimage` structure so that incoming traffic can be easily demultiplexed to the appropriate network stack depending on the interface the traffic is received on. A basic schematic diagram outlining the relationships between most important kernel structures in the clonable network stack implementation is shown in Figure 3.

4.2 Operation

As the symbols and data structures used for network processing, previously declared as global, are now placed inside the `vimage` structure, each function dealing with network traffic necessarily needs to know on which network stack it must operate. Typically, all the references to old symbols such as:

```
ip_ttl = ip_defttl;
```

had to be replaced with constructions such as:

```
struct vimage *vip = {current_vimage};
ip_ttl = vip->ip_defttl;
```

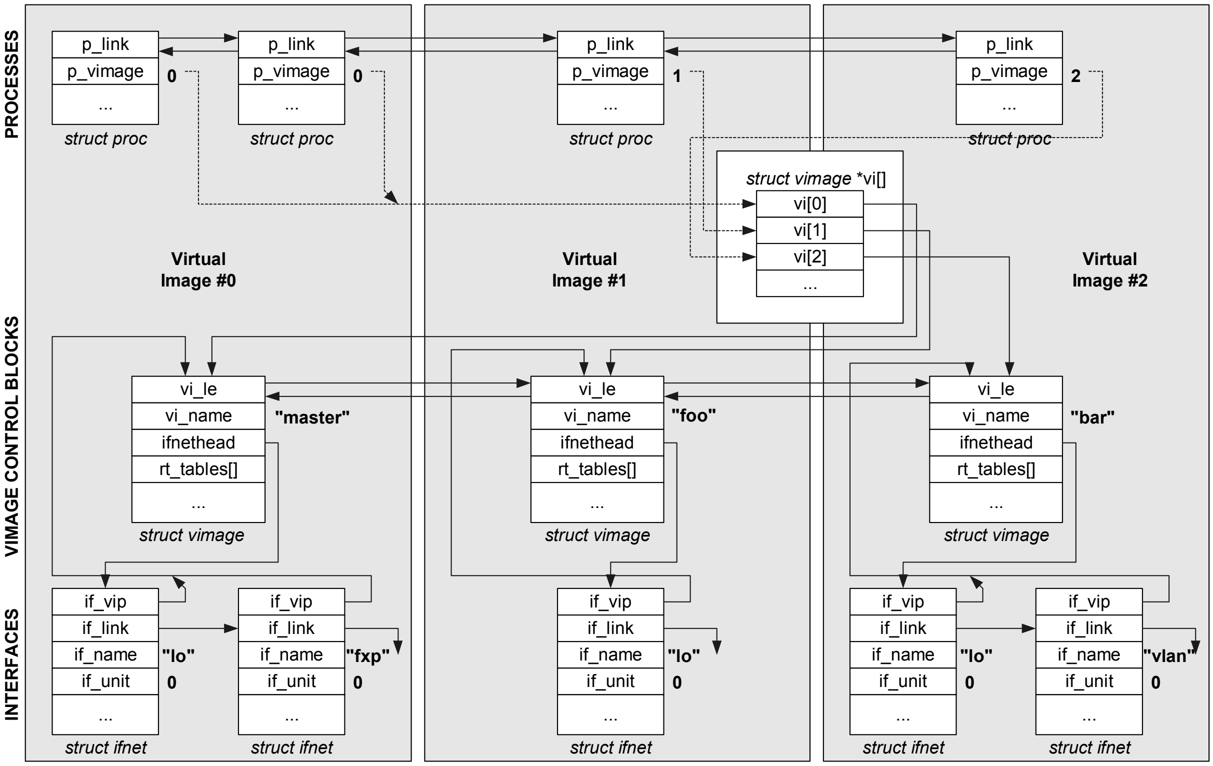

Figure 3 : Major kernel data structures separated and linked throughout *virtual images*

Replacing *all* occurrences of the relevant global variables by their virtualized counterparts throughout the kernel source tree was a textbook example of a time consuming job. After some unsuccessful attempts at automation this task was performed almost entirely manually. The initial patch against 4.7-RELEASE kernel source tree contained 5053 either modified or new lines of code in total.

Basic operations on network traffic are triggered by three types of events: arrival of packets from network interfaces, requests from user programs, and expiration of various timers.

The incoming traffic has to be demultiplexed to the appropriate network stack instance. The network stack instance is determined based on the interface the packet is received on, which can be easily accomplished since network interface descriptors have been extended to hold a pointer back to its `vimage` structure. In the 4.4BSD networking code, each packet is stored in specialized memory structures called `mbuf` [4]. A field in an `mbuf` header is dedicated for holding a pointer to the ingress interface for received traffic. Therefore, the functions that process inbound traffic can extract the information on network stack association for each received frame using the following or similar code (example):

```
void
icmp_input(m, off, proto)
   register struct mbuf *m;
   int off, proto;
{
   struct vimage *vip =
            m->m_pkthdr.rcvif->if_vip;
   ...
}
```

There are certain cases where packets passed to functions in the receiving data path are not tagged with `rcvif` field (it gets set to `NULL` instead). Some typical cases include `dummynet` [12] header processing at the beginning of `ip_input()`, or other subroutines that are used both for inbound and outbound traffic processing, such as portions of the tcp_syncache [13] facility. In such cases additional logic was implemented to ensure proper packet demultiplexing.

It is important to note that although clonable network stacks are designed to be isolated, "global" facilities such as *bridging* and *netgraph* are used to explicitly allow network communication between virtual images and the outside world. In such scenarios it is common that the packet received on one physical interface is bridged to a different (typically virtual) interface residing in another network stack. Therefore the bridging code and similar multiplexers must provide proper retagging of the `rcvif` field in the `mbuf` header when passing the packets in the upstream direction.

The standard 4.4BSD model for processing incoming network traffic is split into two stages [4], resembling the concept of network protocol layering. During the first stage, which deals with *data link* layer, the received frames are demultiplexed to specific network protocol queues (IP, IPX...) and a software interrupt is scheduled by calling `schednetisr()` for the appropriate protocol handler. After all received packets are enqueued, the protocol-level processing is performed in a `netisr()` loop, until all protocol-specific receive queues are completely flushed. While this model works more or less seamlessly in the original monolithic network stack implementation, in the clonable stack framework it does not scale well with large numbers of network stack instances. The problem is that during `netisr()` processing the independent inbound queues of *all* network stacks would have to be checked for pending packets, which is a task with complexity of *O(N)*, where *N* denotes the number of network stack instances present in the system. It is apparent that most of the checking would be completely unnecessary, since it can be expected that only a small number of network stacks are active at the same time.

Therefore the described model of linear traversing through all network stacks during `netisr()` processing has been replaced with a more scalable solution, which introduced a single global receive queue for all network stack instances. However, when flooded with excessive amounts of inbound traffic such as during the typical denial of service (DoS) attacks, a global queue with limited length would start to indiscriminately discard all incoming packets, potentially resulting in all the network stack instances becoming crippled by excessive inbound traffic aimed to only one network stack. On the other hand, the approach with a huge or unlimited global inbound queue would not work properly in such situations either; as such a queue could consume too much `mbuf` resources possibly leading to even more catastrophic consequences. A solution was found in form of a hybrid global queue implementation, with multilevel queue length limiting - both at per network stack and global level at the same time. Such a method in effect emulates multiple independent per network stack inbound queues using a single global queue, while reducing the `netisr()` processing complexity from *O(N)* to *O(1)*. The hybrid global inbound queue model scales well with the number of concurrent network stacks, as shown in Figure 4. The measurements were performed on an AMD Athlon uniprocessor system with a CPU clock of 1200 MHz, a bus clock of 100 MHz, 256 Mbytes of SDRAM, and two Intel 82558B fast Ethernet cards connected to a 32-bit 33 MHz PCI bus.

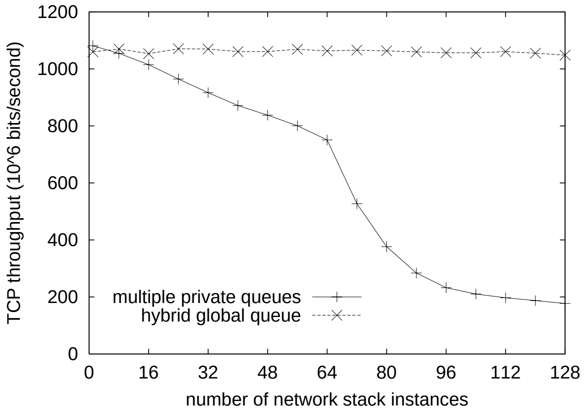

Figure 4: loopback TCP throughput in one network stack instance – comparisson between linear traversing through independent per network stack queues vs. single hybrid global queue. MTU size is 1500 bytes.

The same machine also served as a referent platform in all other measurements presented further in text.

User requests are treated differently depending whether they perform socket operations or other tasks. At the time of creation, each socket is associated with a network stack instance that the owner process is currently bound to, using the newly introduced `so_vip` pointer in `struct socket` descriptor. For the whole duration of its lifetime, the socket remains tied to this network stack instance, even if the owning process changes the network stack association. The functions operating on sockets will therefore determine the network stack instance (or more precisely the *virtual image*) to work with similarly to the following example:

```
static int
tcp_usr_send(struct socket *so, ...)
{
   struct vimage *vip = so->so_vip;
   ...
}
```

Non-socket operations use the `p_vimage` tag in `struct proc` of the calling process to determine the stack instance on which to operate:

```
static int
rip_pcblist(SYSCTL_HANDLER_ARGS)
{
   struct proc *p = req->p;
   struct vimage *vip = vi[p->p_vimage];
   ...
}
```

In the current implementation the `p_vimage` tag is a 16-bit index to an auxiliary array of pointers `struct vimage *vi[]`. Preserving ABI compatibility with userland was the only reason for such a design, as no spare room for holding a direct pointer to `struct vimage` in `struct proc` was available. Retaining both the same size and structure of `struct proc` in the modified kernel was a prerequisite for userland utilities such as `ps` or `top` to operate correctly without recompiling. In the future, the `vi[]` auxiliary array is expected to become obsolete.

Timeout operations, which are typically associated with subsystems such as TCP and ARP processing or *dummynet* [12] delay queues, have to be performed periodically. Those tasks are implemented as linear traversing through all network stack instances and calling the `{slowtimo|fasttimo}` handlers in each supported protocol suite, with pointer to the appropriate `struct vimage` as the argument. Since such events occur synchronously and with significantly smaller frequency than the reception of incoming traffic (only a couple of times per second typically), polling all network stack instances on each invocation of timeout processing subroutines generally cannot have noticeable influence on the overall system performance.

Besides those "core" network stack components described above, some other standard userland-to-kernel interfaces had to be adjusted and extended to support clonable network stacks.

The `sysctl` interface was originally designed to allow system administrators to conveniently monitor and adjust tunable parameters controlling different portions of OS behavior, including the network stack. As the standard `SYSCTL` primitives operate with symbols defined globally within the kernel, they were not directly suitable for accessing the symbols "hidden" inside a `struct vimage`. Therefore a collection of new primitives was introduced. For example the `SYSCTL_V_INT()` macro was implemented for allowing the access to integer variables within a `struct vimage`, replacing the original macro `SYSCTL_INT()` aimed for accessing *global* integer symbols. The `SYSCTL_V` family of macros determines the virtual image they operate on based on the `p_vimage` tag of the calling process.

The kernel symbol lookup interface (`kldsym / kvm_nlist`) is used by userland utilities such as `netstat` to locate symbols in the kernel address space. As many of the commonly accessed symbols including routing tables, protocol control blocks etc. have been replaced by their virtualized counterparts residing in `struct vimage`, the `kldsym / kvm_nlist` programming interfaces were not able to automatically find the addresses of such symbols. Therefore, extensions to those functions were implemented in a similar manner as in the case of the `sysctl` interface, so that the virtualized symbols could be located using the `kldsym` interface. Again, the virtual image instance to be searched in is determined based on `p_vimage` tag of the calling process.

In dynamically loadable kernel modules that implement functions closely related to network processing, such as *bridging* or *ipfw* packet filtering, the load / unload interfaces had to be updated. As multiple network stack instances might be active at the time of module loading, the module has to attach and initialize its private set of data in all of the network stack instances. During deactivation the reverse operation has to be performed, as all data structures in each network stack instance used by the module have to be freed.

4.3 Management interface

A basic API to implement management functions, such as creating, monitoring and modifying the properties of virtual images had to be introduced. The API uses a specialized `vi_req` structure to pass requests and return results to / from the kernel. As a minimum security precaution, regardless in which virtual image the process accessing the API is running, it has to have super-user (*root*) privileges to be allowed to perform any, even read-only operations.

Creation and initialization of new virtual images is the most basic function the API has to support, provided the current virtual image is allowed to create offspring. Similar to the standard UNIX processes, virtual images are logically organized in a hierarchical manner; despite being represented as only one linked list in the kernel. A *master* virtual image is always present, as it is created automatically upon system boot. A system running only with the *master* virtual image is practically indistinguishable from the standard FreeBSD OS. The *master* has the ultimate authority to manage the whole hierarchy of virtual images, and can empower its children to create more offspring, if desired.

To create a new virtual image, the kernel must accomplish several tasks. First it reserves memory to hold the new `struct vimage`, and inserts the new struct into the linked list of all virtual images. The kernel then creates a new loopback network instance and attaches it to the virtual image. Further, it calls initialization routines for all registered network protocols with a pointer to the new virtual image as argument, in a similar manner as during system startup in the original 4.4BSD networking code. Finally, special facilities such as *ipfw* firewalling are initialized for use in the new virtual image. During the whole sequence all interrupts are disabled, which is a sufficient locking method on a single-threaded kernel such as FreeBSD 4.8, however in the future a different locking scheme has to be implemented in the process of migration to more recent OS versions with SMP support in the kernel.

At the time of creation, a *chroot* directory in which all the processes in the virtual image will run can be specified, together with a limit on the average percentage of CPU time the virtual image is allowed to consume. Although these parameters are optional, it is very likely they will be used in any serious virtual hosting scenario. The CPU percentage limit can be adjusted at any time; however changing the *chroot* directory requires no processes to exist within the virtual image. The chrooted directory tree has to be set up in the same manner as for the "classic" jailed environments.

Deletion of virtual images requires the protocol initialization sequences performed during the virtual image creation to be followed *backwards* during detachment of each network protocol instance. A virtual image can be deleted only when no user processes and sockets are attached to it. The deletion routine has to walk through all the configured protocol instances and gracefully free private memory structures used by each specific protocol suite, such as interface addresses, routing and hash tables, protocol control blocks etc. Furthermore, all the pending timers associated with protocols such as ARP or TCP have to be canceled, before the virtual image is unlinked from the global list of all virtual images, and the corresponding `struct vimage` can be freed.

The network interfaces can be *moved* from one virtual image to another. For example, this makes it possible for 802.1q VLAN interfaces bound to a "base" physical interface in one virtual image to be reassigned to other virtual images. Another use of this concept might be on machines with multiple physical interfaces to allow each interface to be assigned to an independent virtual image. As the interface moves from one virtual image to another it is automatically assigned a new index; for example if Ethernet interface `fxp1` is moved to another virtual image where no `fxp` interface instances exist, it will be renamed to `fxp0`.

User processes can use the API to switch to one of the children of their virtual image; however as a security precaution they cannot switch back to their parents or siblings. This in effect results for such switching to be used solely for spawning new processes in desired virtual images. As the only application to use this API, the `vimage` management utility has an option to prevent the new process from becoming chrooted before being spawned in the new virtual image, which can be useful for monitoring purposes and as a rescue option in cases when the private directory tree in the target virtual image becomes damaged or unusable.

Although under normal circumstances it can be expected that management operations on virtual images will be executed relatively infrequently, the current implementation requires the whole kernel to be locked for the entire duration of all virtual image management system calls. Fortunately, the typical execution time of management functions is generally short enough not to introduce unacceptable delays, even on busy servers. Table 1 shows the typical execution duration of management operations, measured from the userland perspective on an otherwise idle system.

Function	***Duration***
Create a new vimage	212 µs
Delete a vimage	72 µs
Move network interface to a vimage	47 µs
Switch a process to another vimage	20 µs
Modify vimage parameters	21 µs
Fetch statistics for one vimage	25 µs

Table 1: Average execution time of virtual image management operations on an idle system

4.4 Other pseudo-VM functions

Implementing a clonable network stack was the original and primary focus of the experimental code, without presumption whether the application will be virtual hosting, VPN provisioning, network simulation or something completely different. However, extending the experimental framework to provide some basic pseudo-VM functions, like hiding user processes running in one *virtualized* environment, turned out to be a fairly trivial task. This part of the implementation was easy because the basic framework offering such functionality was already there – the popular *jail* [2] facility in FreeBSD. The original *jail* implementation uses the `PRISON_CHECK(p1,p2)` macro at various points in the kernel to determine whether two processes are supposed to "see" and interact with each other by checking if they both belong to the same *jail*. It was sufficient to extend this macro by checking if the processes belong to the same network stack instance to achieve the same separation functionality originally present in *jails*. This basic facility was further extended with new functions related to management of CPU resources.

CPU load and usage accounting has been reorganized from system-wide to per virtual image, by migrating the `cp_time` and `averunnable` variables to `struct vimage`. This allowed the CPU time spent in *user* and *system* contexts to be charged to the appropriate virtual image, which can help providing the administrators with better control and overview over the system behavior, and further restricting the global view on the system from within the virtual images.

The problem of accurately defining and finding the actual consumer of CPU time spent in the *interrupt* context is still an open area in OS research. Traditionally, UNIX systems pragmatically (and unfairly) charge the unlucky current process with the time spent in servicing interrupts. We found out that on the level of virtual images it is more practical to charge *all* virtual images for the time slice spent in the *interrupt* context, instead of only charging the current one. Such an approach simplified the implementation of individual per virtual image overall system load estimation. However, further experiments have shown that although it might be difficult or impossible to determine the actual consumer for a broad range of interrupt triggered events, for example for disk I/O DMA notifications, charging the appropriate virtual image for network traffic related interrupt context events can be implemented fairly simply and efficiently. Since the standard UNIX accounting is performed on statistical basis by periodically probing the state of CPU execution context (user, system, interrupt or idle), it was necessary to provide the accounting routine with more detailed information on which virtual image is currently active in the interrupt context. This was accomplished by introduction of a new global variable `vintr` which points to the current virtual image on which network processing is performed in the interrupt context.

As the interrupt context network processing in 4.4BSD kernels is split into two stages, the global `vintr` variable has to be set both on each entry to NICs device driver interrupt service routine and later during `netisr` processing. At exit the interrupt service routines are responsible to set the `vintr` back to `NULL`. When the `statclock()` accounting routine preempts a thread of execution running in the interrupt context, it checks whether `vintr` is set, in which case it can increment the corresponding counters only for the current virtual image. Otherwise, if `vintr` equals to `NULL`, *all* virtual images are charged for the consumed timeslice in interrupt context.

The described model of interrupt context time accounting is far from being accurate for two main reasons. First, it cannot properly classify the context switching and interrupt dispatching periods before the device driver can set the `vintr` variable, as well as after the `vintr` variable is cleared. Secondly, in the case when the traffic has to be *bridged* to virtual image different from the one the physical NIC resides in, it is highly probable that over a longer period of time *both* virtual images will accrue interrupt context timeslots. Such an accounting is logically invalid, since only the virtual image that is the final destination of the incoming traffic should be charged. Nevertheless, despite not being entirely accurate, the per virtual image interrupt context accounting model can be a valuable indicator to system administrators in which virtual image to look for inbound traffic triggered CPU congestions.

CPU usage limiting was another function implemented as a straight follow up on the individual CPU usage accounting. Based on CPU usage limits set by the system administrator, the original 4.4BSD algorithm for scheduling the user processes on the CPU was extended to simply *skip* the processes in the active *run queue* belonging to virtual images that have recently consumed more CPU time than they were allowed to. As soon as it can be determined that the CPU usage limit has been exceeded, a process belonging to another virtual image is rescheduled, or the system enters the idle loop. A digital decay filter periodically lowers the accumulated per virtual image CPU usage averages, thus allowing the processes running in administratively constrained virtual images to be rescheduled later.

Receive livelock [14] is a situation when more incoming packets arrive than the system is capable to handle, resulting in the CPU being permanently locked into servicing interrupt requests. This presents a serious threat to stability of systems based on interrupt driven kernels, such as FreeBSD. In an OS split in multiple virtual machine environments the problem becomes even more emphasized since the entire system could become crippled under livelock resulting from only one network stack instance being flooded by excessive inbound traffic patterns (typical for today's frequent DoS attacks).

After successfully implementing per virtual image interrupt context CPU usage accounting, it was almost trivial to implement a simple feedback-based control algorithm for mitigating receive livelock. The interrupt context load is dampened by controllable discarding of received packets from the inbound protocol queues during `netisr()` processing. The interrupt context load threshold, based on which the decision is made whether to drop or pass the packets to further protocol processing, can be independently tuned for each virtual image. The described method is based on an assumption that in average more CPU cycles are consumed in protocol level processing than in device drivers, which only have to demultiplex received packets to the appropriate protocol inbound queue. Therefore dropping the packets in the `netisr()` loop can significantly lower the CPU time spent in interrupt context processing only for traffic flows that require complex protocol-level processing (such as TCP or IPSEC), or in scenarios where complex packet filtering rules have to be traversed for each received packet. However, it should be noted that the above method for dampening interrupt context load does not provide an ultimate solution in mitigating receive livelocks. *Interrupt coalescing* and especially *polling* [15] can provide more adequate protection against livelocks under extreme overloads.

4.5 Memory resources

The kernel memory footprint is affected by the introduced modifications when running multiple network stacks. Compiled for the IA-32 platform, the modified kernel image file is 1663393 bytes big, which is an increase of only 5977 bytes compared to an equally configured unmodified kernel. However, the real issues with memory allocation arise when multiple virtual images are active in the system. Those issues can be classified in two groups.

Various private memory pools such as protocol control blocks (PCB), hash tables etc. which are created for each virtual image independently introduce specific implications. Although the size of `struct vimage` itself is relatively small (8012 bytes), during virtual image creation additional space for private memory pools has to be reserved. Thus `vmstat -m` reports that each virtual image consumes a total of approximately 23 Kbytes of kernel memory upon initialization. Unfortunately, this figure is not relevant, as it does not account for reserved but unused memory pages allocated by the *zone allocator* for private PCBs and the TCP syncache. Using the standard limits for maximum number of sockets and TCP syncache size, each virtual image could consume a total of 1161 pages or 4664 Kbytes of kernel memory. It is obvious that this mechanism presents a serious obstacle in scaling to large number of simultaneous virtual images. Optimal use of kernel memory could be accomplished by implementing common zone pools for all network stack instances; however, such an approach would violate the design requirement of making network stacks as independent as possible. An alternative model is therefore employed, which retains private memory pools, but enforces lower limits on number of sockets per virtual image, thus significantly reducing the per virtual image memory footprint. Such an approach offers additional tuning possibilities on resource limiting, since it allows system administrators to specify a per virtual image limit on the number of sockets.

The `mbuf` packet buffers present a different issue, as they are shared among all network stack instances. For example, a heavily utilized traffic shaper or delay queue in one network stack could bind and effectively exhaust all the `mbuf` buffers in the system, rendering other network stack instances unusable. Mitigation of these issues is not a trivial task to solve, as it would require implementing new limit checking mechanisms to the `mbuf` allocator, accompanied by additional checking done at all places in the kernel where `mbuf` buffers are either created or handed over from one network stack instance to another. As such an approach would require significant programming efforts it has not been implemented in the experimental network stack cloning framework. Therefore the system's stability depends on network stack instances to cooperatively share the global `mbuf` pools.

5 Performance

Extending any piece of software with a new functionality always rises the question what impact it will make on system performance. In the case of extending the network stack to support cloning, this question becomes highly emphasized since *all references* to symbols and variables used throughout the network stack have one additional level of indirection. To compare the performance between the standard and modified kernels, a series of simple tests have been run.

Microbenchmarking tests were performed in order to determine the execution duration of some typical kernel functions involved in packet processing. In each observed function, both in the standard and modified kernel, counter *start* and *stop* hooks were embedded for capturing the elapsed time. The CPU-embedded *TSC* clock cycle counter was used as the time reference. Execution of each observed section was provoked by appropriate external traffic on an otherwise completely idle test system.

Table 2 shows the comparative results of a series of such tests. The first two columns mark the observed function and the test traffic used, followed by average execution duration and its standard deviation from the 5000 subsequent iterations, for the modified and standard kernel. Throughout the tests the modified kernel was running with only a single virtual image / network stack instance, resembling the functionality of the standard kernel.

From the broader series of tests, only those results are included that were both repeatable and with reasonably small standard deviation. Examining the results it becomes apparent that accurate measurement of execution time becomes more difficult as the observed code section becomes longer and more complex, which can be partially explained by the influence of phenomena such as system bus contentions, CPU cache coherence etc. Nevertheless, the test results are valuable in that they show no dramatic difference in execution time between code sections running on the modified and standard kernel. In fact, in the modified kernel some functions seem to execute slightly faster than in the standard one; however, as the functions shown in Table 2 present only a portion of tasks that the kernel performs in providing network communication, these small deviations can be considered insignificant.

The effective throughput for different types of traffic was measured during the second series of tests, again comparing the performance between standard and modified kernels. The test machine was powerful enough to easily drive the physical network interfaces at full media speed using most of the interesting traffic patterns, so it became apparent that testing with external traffic would not yield particularly usable results. Therefore it was decided to perform all the testing *inside* the referent machine, using it at the same time as source and drain of the test traffic. The traffic was looped back between traffic source and destination, eliminating the contentions that physical media such as Ethernet could introduce.

The TCP throughput was measured using the *netperf* [5] utility. Figure 5 shows the measured relation

		clonable stack		**standard stack**			
function	***traffic***	***average***	***stdev***	***average***	***stdev***	***+/- cycles***	***+/- %***
ip_input()	ICMP echo	224	9,5	144	12,6	80	56%
ip_output()	ICMP echo reply	502	29,3	492	17,3	10	2%
icmp_input()	ICMP echo	504	204,9	660	213,1	-156	-24%
icmp_reflect()	ICMP echo / reply	181	4,8	153	4,8	28	18%
tcp_input()	TCP data	4978	953,4	5263	1002,1	-285	-5%
tcp_input()	TCP ack	2280	479,1	2471	491,9	-191	-8%
total		***8669***		***9183***		***-514***	***-6%***

Table 2: Execution duration (in clock cycles) of certain packet processing functions in modified and standard kernel

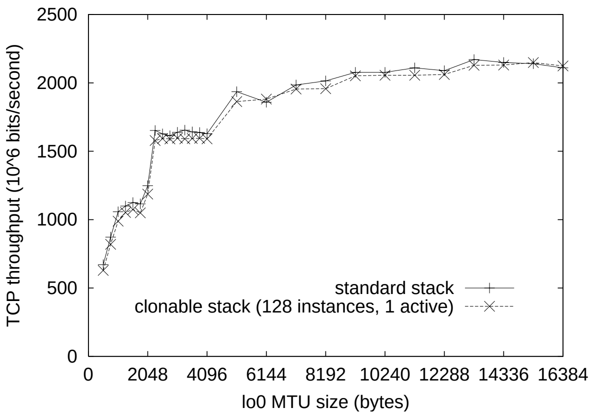

Figure 5: loopback TCP throughput

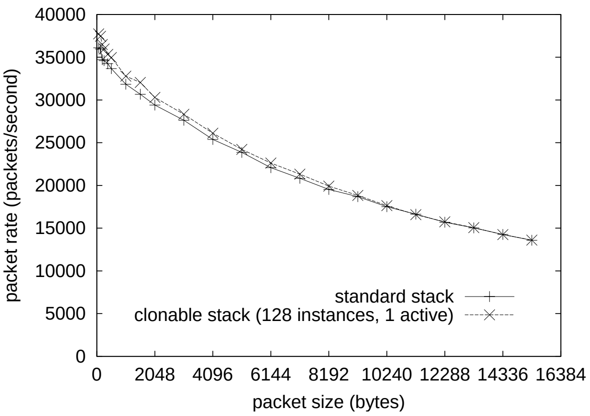

Figure 6: loopback packet rate

between maximum TCP throughput and the MTU size configured on the loopback network interface. It can be observed that the throughput obtained on the modified kernel running with a single network stack instance is marginally lower than on the unmodified kernel. For the MTU size of 1500 octets the achieved throughput using the modified kernel was around 93% of the value observed on the standard system. However, it should be noted that during this test the traffic passed through the network stack twice: once when data was transmitted by the server process (`netserver`), and once when the same data was received by the `netperf` client. The one-way throughput degradation is even less significant, and can be estimated as a square root of the obtained throughput ratio between standard and modified stack for both sending and receiving side processing. Therefore, for MTU=1500 we can estimate one-way maximum TCP throughput of the clonable network stack to be around 96.5% of the unmodified system.

Measuring the maximum packet rate for ICMP traffic, generated by the "flooding" `ping -qf` command and reflected by the kernel, yielded some interesting results. As shown in Figure 6, the maximum packet rate obtainable on the modified kernel was up to 5.7% *higher* than using the standard kernel. It is difficult to explain such a phenomenon, but it can be speculated that this might be due to better CPU cache coherency, since in the modified kernel most of the symbols involved in network processing are located close to each other in the `vimage` structure, while in the standard kernel they are interleaved with symbols used in other kernel functions not related to the network stack.

To summarize, all the presented measurements clearly indicate that the implemented network stack cloning code generally does not significantly degrade the system and network performance.

6 Application examples

The concept of network stack cloning was conceived with the goal of supporting a broad class of applications. Most system administrators will however be curious how the new framework fits in *virtual hosting scenarios*. In Figure 7 a simple virtual hosting configuration is shown where the system is split into three virtual images: "`master`", which is the default; and two subordinated virtual images called "`client1`" and "`client2`". The client virtual images are each assigned its private virtual Ethernet "`ve`" network interface, which are all *bridged* to the physical LAN through the real Ethernet interface (`fxp`0) residing in the "`master`" virtual image. The client virtual images reside in *chrooted* directory trees, which can be created using the standard methods for setting up the *jailed* environments, described in `jail(8)` online manual.

The following command sequence can be used to initially configure the described virtual hosting environment. The `vimage` command is used for

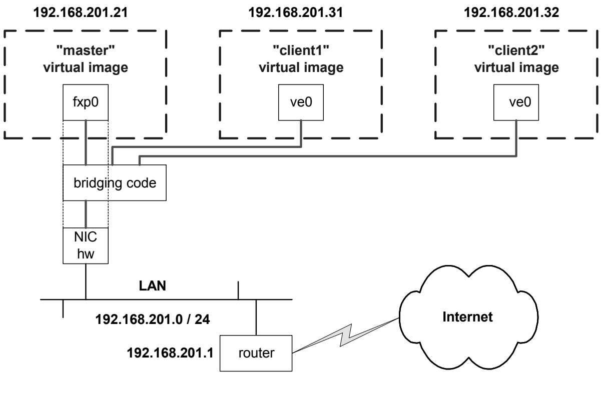

Figure 7: A simple virtual hosting scenario

managing the virtual images and network interfaces, with modifier determining the action to be performed. It is assumed the private directory tree for each client virtual image have been set up previously.

```
# create virtual ethernet interfaces
ifconfig ve0 create link 40:0:0:0:0:1
ifconfig ve1 create link 40:0:0:0:0:2

# create new virtual images
vimage -c client1 chroot /v/client1
vimage -c client2 chroot /v/client2

# move interfaces to new virtual images
vimage -i client1 ve0
vimage -i client2 ve1

# configure the bridge
sysctl \
  net.link.ether.bridge_cfg= \
  client1.ve0,client2.ve0,fxp0
sysctl net.link.ether.bridge=1
```

At this point no user processes exist in the "client" virtual images. Provided the virtual images are properly configured using the standard `rc.conf` file residing in their private directory tree, they could be started using the `/etc/rc` script, in similar manner as used for starting the standard *jails*. This could be accomplished using the following commands:

```
# start new virtual images
vimage client1 /bin/sh /etc/rc
vimage client2 /bin/sh /etc/rc
```

Unlike in jails, the parent virtual image can spawn a new process in its child at any time. The following example shows how IP configuration of virtual images can be performed manually:

```
vmbsd# vimage
master
vmbsd# vimage client1
Switched to vimage client1
# ifconfig
ve0:   flags=8903
<UP,BROADCAST,PROMISC,SIMPLEX,MULTICAST>
mtu 1500 ether 40:00:00:00:00:01
lo0: flags=8008
<LOOPBACK,MULTICAST> mtu 16384
# ifconfig lo0 localhost
# ifconfig ve0 192.168.201.31
# route add default 192.168.201.1
# inetd
# ps -ax
  PID  TT  STAT      TIME COMMAND
  248  ??  Ss     0:00.02 inetd
  242  p1  S      0:00.06 vimage (csh)
  249  p1  R+     0:00.01 ps -ax
# hostname freenix
# exit
```

It is now possible to verify if the new virtual image can be accessed over the network:

```
vmbsd# telnet -K 192.168.201.31
Trying 192.168.201.31...
Connected to 192.168.201.31.
Escape character is '^]'.

FreeBSD/i386 (freenix) (ttyp4)
...
```

The above example illustrates only the initial sequences in the management of virtual images. As at the first glance the achieved functionality might seem very similar to what can be done with the traditional FreeBSD jails, it should be noted that the real differences can be observed in the areas which are *not* supported in jails. Maintaining multiple IP addresses, independent packet filters and routing tables, access to the routing and raw sockets as well as `bpf` traffic capturing are among the key new functions the network stack cloning model introduces to the jail-styled virtual hosting scenarios.

The other applications that can benefit from the clonable network stack infrastructure range from fast and efficient real-time network simulations to advanced *overlayed* VPNs with independent and potentially overlapping addressing schemes. Unfortunately, the scope of this article limits further discussion on possible applications in these areas.

7 Conclusions and future work

The main contribution of this work is demonstrating the concept of network stack cloning can be efficiently implemented as an extension to the existing FreeBSD networking code. The experimental implementation has successfully preserved both the same overall performance level and full API/ABI compatibility with the original kernel, which were the two key requirements for adopting the network stack cloning model in a broad range of applications. When running with only a single network stack instance, looking from the userland perspective it is practically impossible to distinguish between a modified kernel and the original one, both regarding the general appearance, functionality, application interface, overall performance and memory footprint. However, by creating new network stack instances associated with *virtual images*, system administrators now can conveniently and efficiently partition the OS into highly independent *pseudo* or *light* virtual machine entities.

A substantial amount of work has yet to be completed before the network stack cloning model could be even considered for inclusion in an official FreeBSD source tree. As the experimental implementation covers only virtualization of basic IPv4 networking code, obviously the cloning support should be extended to other protocols, starting with IPv6 and IPSEC. Further, the framework should be ported and kept in sync with a development *(-current)* source tree, since the original experimental implementation is based on the *-stable* 4.x FreeBSD branch. And even if and when a highly polished patch against a development tree would become available, it is imminent that the question would arise whether the potential benefits of network stack cloning could overweight the compatibility issues associated to maintaining different private or parallel source trees, which would become obsolete once the cloning patch would get integrated in the official kernel source tree.

In parallel with bringing the code in closer sync with the FreeBSD –current branch, the original concept of partitioning the OS in *virtual images* could be further extended by virtualizing other system resources, such as real and virtual memory, network and disk bandwidth, etc. An interesting option for further development could certainly be the reimplementation of *virtual images* as a modular *resource container* [6] type facility. Each resource instance (network stack, process group, CPU, memory etc.) would be represented by its own data structure, and `struct vimage` would only contain pointers to such structures. In such an environment, the system administrator could freely combine only the desired virtualized system resources in a *virtual image*, depending on the specific environment and application requirements.

The experimental code for network stack cloning support, along with additional technical information and application examples is available for download under a BSD-style license at http://www.tel.fer.hr/zec/vimage/. At the time of this writing, the source is maintained as a set of patches against the FreeBSD 4.8-RELEASE kernel.

Acknowledgements

The author would like to thank Guido van Rooij, who was shepherding this work, for thorough and critical commentaries throughout the development of the paper, as well for all his previous encouragement and support in presenting the network stack cloning concept to the FreeBSD community. Many thanks go to anonymous reviewers for their valuable comments and suggestions. Finally, the author would like to thank Julian Elischer for helping with presenting this work at the conference.

References

[1] Sugerman, J., Venkitachalam, G., and Lim, B. *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor*, In Proc. of the USENIX Annual Technical Conference, 2001.

[2] Kamp, P., and Watson, R. *Jails: Confining the Omnipotent root*, In Proc. of the 2nd International SANE Conference, 2000.

[3] McKusick, M., et al. *The design and implementation of the 4.4BSD operating system*, Addison-Wesley, 1996.

[4] Wright, G., and Stevens, W. *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley, 1994.

[5] netperf: http://www.netperf.org/

[6] Banga, G., Druschel, P., and Mogul, J. *Resource containers: A new facility for resource management in server systems*, In Proc. of the Symposium on Operating System Design and Implementation, 1999.

[7] Huang, X., Sharma, R., and Keshaw, S. *The ENTRAPID Protocol Development Environment*, In Proc. of the IEEE INFOCOM, 1999.

[8] Ely, D., Savage, S., and Wetherall, D. *Alpine: A User-Level Infrastructure for Network Protocol Development*, In Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems, 2001.

[9] Wang, S., and Kung, H. *A Simple Methodology for Constructing Extensible and High-Fidelity TCP/IP Network Simulators*, In Proc. of the IEEE INFOCOM, 1999.

[10] Cobbs. A. *All About Netgraph*, http://ezine.daemonnews.org/200003/netgraph.html

[11] Scandariato, R., and Risso, F. *Advanced VPN support on FreeBSD systems*, In Proc. of the 2nd European BSD Conference, 2002.

[12] Rizzo, L. *Dummynet: A simple approach to the evaluation of network protocols*, ACM Computer Communication Review, 1997.

[13] Lemmon, J. *Resisting SYN flood DoS attacks with a SYN cache*, In Proc. of the BSDCon 2002.

[14] Mogul, J., and Ramakrishnan, K. *Eliminating receive livelock in an interrupt-driven kernel*, ACM Transactions on Computer Systems, Vol 15, No. 3, 1997.

[15] Rizzo, L. *Device Polling support for FreeBSD*, http://info.iet.unipi.it/~luigi/polling/