Virtual Servers and Checkpoint/Restart in Mainstream Linux

Sukadev Bhattiprolu
IBM
sukadev@us.ibm.com

Eric W. Biederman
Arastra
ebiederm@xmission.com

Serge Hallyn
IBM
serue@us.ibm.com

Daniel Lezcano
IBM
dlezcano@fr.ibm.com

ABSTRACT

Virtual private servers and application checkpoint and restart are two advanced operating system features which place different but related requirements on the way kernel-provided resources are accessed by userspace. In Linux, kernel resources, such as process IDs and SYSV shared messages, have traditionally been identified using global tables. Since 2005, these tables have gradually been transformed into per-process namespaces in order to support both resource availability on application restart and virtual private server functionality. Due to inherent differences in the resources themselves, the semantics of namespace cloning differ for many of the resources. This paper describes the existing and proposed namespaces as well as their uses.

Categories and Subject Descriptors

C.5.5 [**COMPUTER SYSTEM IMPLEMENTATION**]: Security
; B.8.1 [**PERFORMANCE AND RELIABILITY**]: Reliability, Testing, and Fault-Tolerance

General Terms

Reliability, Security

Keywords

Survivability, Reliability, Security, Checkpoint, Restart, Mobility, Virtualization

1. INTRODUCTION

A namespace is a fundamental concept in computer science. An instance of a namespace serves as a dictionary. Looking up a name, if the name is defined, will return a corresponding item. In this sense, of course, kernel-provided resources have always been namespaces. For instance, multiple process ID or pid, tables can co-exist in the system but one task will use one table to look for a task corresponding to a specific pid, in other word this table is relative to the task.

For the past several years, we have been converting kernel-provided resource namespaces in Linux from a single, global table per resource, to Plan9-esque [31] per-process namespaces for each resource. This conversion has enjoyed the fortune of being deemed by many in the Linux kernel development community as a general code improvement; a cleanup worth doing for its own sake. This perception was a tremendous help in pushing for patches to be accepted. But there are two additional desirable features which motivated the authors. The first feature is the implementation of virtual private servers (VPS). The second is application checkpoint and restart (ACR) functionality.

This paper is the first to present recent work in the Linux kernel which builds an infrastructure for supporting VPS and ACR. The remainder of this introduction will present a general overview of VPS and ACR. Section 2 will summarize related work. Section 3 will describe the general namespace support in Linux and its usage, and detail the semantics of existing and planned namespaces.

1.1 Virtual Private Servers

Otherwise frequently referred to as jails or containers, virtual private servers (VPS) describe an operating system feature that isolates and virtualizes the resources used by a set of processes, so that two VPSs on a single host can be treated as though they were two separate systems. The consolidation of many VPSs onto a single physical server can provide significant power and cost savings. The fact that computational power and memory are not used to emulate hardware and run many instances of operating systems reduces cost in terms of power, hardware, and system administration. The fact that a single operating system serves many VPSs also eases the sharing of resources among them, such as disk space for system libraries.

The implementation of VPSs requires resource isolation between each VPS and the host system, and resource virtualization to allow each VPS the use of the same identifier for well-known resources. It also requires administrative compatibility, so that setting up a VPS is done using the same tools as setting up a host, and it requires transparency, so that applications can run unmodified on a VPS. The requirements for compatibility and transparency should not however be interpreted as a requirement that a VPS user cannot tell that they are using a VPS rather than a host.

While some may consider that a worthwhile goal, it does not contribute to VPS usability, and has been explicitly rejected as a goal for Linux VPSs.

Linux now implements per-process namespaces for many resources. Each process references a set of namespaces (usually) shared with other processes in the same VPS. When a task asks, for instance, to send a signal to process 9920, the kernel will search the signaling task's *pid namespace* for a process known in that namespace as 9920. There may be many processes known as 9920 in some namespace or other, but only one task will have that ID in the signaling task's namespace. Any task that does not have an ID in the signaling task's *pid namespace* cannot be signaled by that task [7]. The namespaces thus provide both isolation and virtualization.

In order to prevent VPS users from subjecting each other or host users to service denials, resource management is also needed. Namespaces themselves do not facilitate resource management. Linux provides resource management through the *cgroups* interface [28].

1.2 Application Checkpoint and Restart

ACR allows a running application's state to be recorded and later used to create a new instance of the application, which continues its computation where the original had left off at the time of the checkpoint. Common uses for this include migration, i.e. moving an application to a less loaded server or off a server that will be shut down, and survivability of application death or either planned or accidental system shutdown.

One obvious requirement for ACR is the ability to dump all relevant task data to userspace in the form of a file or data stream. Another requirement is the ability to recreate all resources in use by the application, and to do so using the identifiers by which the application knows them. For instance, if an application consists of tasks T1 and T2, T1 may have stored the pid for T2 so as to signal it at some later time. Upon application restart, T2 must be restarted with the same pid. There must therefore be no other task already known by T2's pid.

In Linux, the per-process namespaces can be used to meet this particular part of the requirement. When the application is restarted, it is restarted with a new, most often empty [1], set of namespaces. This guarantees the availability of the required resource IDs within the application's namespace.

2. RELATED WORK

The use of virtualization has many motivations [11], including resource isolation, hardware resource sharing or server consolidation, and operating system and software development for unavailable hardware. Logical partitioning is strictly a method of hardware resource sharing, where several distinct operating systems can be run on a subset of the available hardware [26, 38]. There are several ways virtual machines are typically implemented. Full hardware emulation in software [3] is the slowest but most comprehensive. In virtualization [20, 18] or para-virtualization [2, 33] much of the virtual machine is run natively on the host system rather than being fully emulated, with a smaller piece of software called the Virtual Machine Monitor or hypervisor providing the remaining emulation. Each of these approaches involves execution of a full operating system for each virtual machine, incurring large memory and cpu costs.

As pointed out by Price and Tucker [32], when the primary goal of a virtual machine is to consolidate applications, then what is mainly needed is the namespace isolation features. By providing namespace isolation without requiring any instruction translation, hardware emulation, or even a private copy of an operating system for each virtual machine, virtual private servers (VPS) become a more efficient solution. The correctness of this observation is borne out by the large number of implementations [37, 19, 32, 35, 36, 41], making a strong case for the implementation of private namespaces for kernel-provided resources in the mainstream Linux kernel.

Single System Image (SSI) refer to a distributed operating system providing the appearance of a single machine on top of a cluster. An SSI, such as SPRITE [5] or KERRIGHED [43], usually transparently migrates jobs between machines on the underlying cluster to take advantage of idle machines [27]. MOSIX [1] offered application migration for VAX and MOTOROLA and then for pentium-class computers in 1998 for Linux.

In contrast, the use of migration to achieve load-balancing among a pool of systems can be seen as achieving the benefits of SSI without the cost of constantly synchronizing resources to provide the illusion of a single system. ACR technology has been developed for years. Early ACR implementations focused on checkpointing the whole system in the case of power failure or other system shutdown [15]. Later operating systems such as KEYKOS [22] did the same thing. FLUKE [8] provided checkpoint of applications or nested virtual environments in 1996. Commercially, IRIX 6.4 [44] implemented ACR functionality as specified by the (unratified) POSIX 1003.1m draft 11 in 1996. In 2002, CRAY offered UNICOS/mp [16], which was based on IRIX 6.5 and thus offered the same ACR functionality. Recently IBM has offered ACR for AIX WPARs [14] These different implementations demonstrated the validity of the ACR concept and shown the interest of users, especially from the high performance computing world, for this technology.

In Linux, ACR technology has studied several different approaches: user space approaches [23, 46], fully in-kernel approaches [41, 21], and mixed user and kernel solutions [6]. The userspace-only solutions were good attempts to examine how far the technology can go without modifying the kernel, but the restrictions imposed by this approach are too heavy [24, 46] to be used by a wide scope of users. In particular, all the attempts to have the ACR available without modifying the kernel faced three major problems: identify the kernel resource to be checkpointed (eg. how to know if an IPC resource should be checkpointed or not); resolve kernel resource conflict at restart (eg. how to assign a pid to a process if there is another process in the system with the same pid); and access the internal kernel data structure so as to recreate the resource exactly in its original, checkpointed state.

The namespace work presented here resolves the two first problems. A checkpoint-able job is spawned with a new, empty set of namespaces, so that all kernel resources available to it are in fact worth checkpointing because there are safely isolated from the rest of the system. A job is also restarted in a new set of namespaces, so that recreated re-

[1] Except for *hostname* and *mounts namespaces*, where the concept of resource ID conflicts does not make sense.

sources can not conflict with other jobs on the system.

VPS and ACR implementations on Linux have been coming and going for a long time. Current examples of VPS implementations include Linux-VServer [37] and OpenVZ [41]. ACR implementations include Zap [21], MetaCluster [10], and OpenVZ [41]. By addressing a common need of both VPS and ACR, a need whose solution must inherently be very invasive to the core kernel and therefore greatly increase the size of any kernel patch, the namespace work greatly reduces the amount of work needed to implement both.

3. NAMESPACE INFRASTRUCTURE

The use of per-process namespaces as an OS concept can be traced back to Plan9 [31]. Since another, even more famous, design feature of Plan9 was that “everything is a file”, it should not be surprising that Plan9 namespaces are mostly thought of as filesystem namespaces. But whereas in Plan9 filesystem namespaces suffice to support namespaces for everything, the same is not true in Linux.

The main design goals and requirements for namespaces for Linux are:

- Transparency and Compatibility: Applications must run inside the namespace just as if they are running on the host system.
- Performance: The namespaces should not introduce any significant overhead to applications.
- Usability and Administration: Existing utilities and administrative tools should continue to work both inside and outside namespaces.
- Acceptance to main-stream Linux: Overall design and implementation of namespaces must be homogeneous with the Linux kernel and not just a customized kernel or module.

The list of namespaces currently included or being implemented in the Linux kernel includes *hostinfo* (Section 3.3), *system V IPC* (Section 3.4), *mounts* (Section 3.5), *pid* (Section 3.6), *network* (Section 3.7), *userid* (Section 3.8), and *devices* (Section 3.9).

3.1 Operations on namespaces: Cloning and unsharing

In Linux new processes are created using the `clone()` system call. `Clone()` differs from the traditional `fork()` system call in UNIX, in that it allows the parent and child processes to selectively share or duplicate resources. The resources to be shared or duplicated are specified in a `clone_flags` parameter to the system call. For instance, parent and child share virtual memory if the `CLONE_VM` flag is set in the call to `clone()`. The process of duplicating resources between parent and child using the `clone()` system call is referred to ”cloning” the resource. While most resources must be duplicated at the time of process creation, some can be duplicated after the child process has been fully created using the `unshare()` system call. We use the terms, cloning or unsharing interchangeably to refer to both operations.

3.2 *nsproxy*

A `task_struct` describes an active task in the system. Rather than provide a pointer to each namespace in every `task_struct`, it was decided that a ’namespace proxy’, or `nsproxy`, should be referenced from a `task_struct`. In addition to space savings due to the presumably numerous tasks storing only one pointer (to `nsproxy`) instead of many (to each namespace), there is also a small performance improvement, since each ordinary task clone required only incrementing one reference count for all namespaces.

The process of cloning or unsharing a namespace using the `nsproxy` is best explained with the simple, *hostinfo namespace* below.

3.3 Creating namespaces

As each new namespace is introduced into the Linux kernel, a new field is added into the `nsproxy` describing the namespace. A new clone-flag is used to identify the namespace when cloning or duplicating the namespace. For example, to clone or duplicate the *hostinfo namespace*, the `CLONE_NEWUTS` flag is used with the `clone()` or `unshare()` system call.

Unsharing a namespace, say the *hostinfo namespace*, causes a new `nsproxy` to be created. All namespace-references, except *hostinfo* are copied from the parent `nsproxy`, and their reference counts are bumped. The `nsproxy` references a new *hostinfo namespace* that is taken as a fresh copy of the original.

Changes in the *hostinfo* information in the new namespace are not reflected in the other. Thus two sets tasks in separate *hostinfo namespaces* on the same machine can have different values for host name and domain names.

As each task exits, the reference count on the corresponding `nsproxy` is decremented. When no tasks remain referencing an `nsproxy`, the `nsproxy` is freed and references to all its namespaces dropped. In turn, if the `nsproxy` is the last one pointing to any of its namespaces, then those namespaces are also freed.

3.4 System V IPC

System V Inter-Process Communications (IPC) provide shared memory, semaphores, and message queues. Each IPC object must have a unique ID that cooperating processes can use to access the shared resource. Previously, three tables translated the IDs into the actual resource. To support per-process namespaces, we made these tables a member of the `ipc_namespace` structure, which is referenced by the `nsproxy`. When a new *ipc namespace* is cloned, it is created empty, rather than as a copy of the parent namespace. The *ipc namespaces* therefore form a simple, completely isolated and disjoint sets. As we are about to see, other namespaces introduce far more complicated relationships.

3.5 Mounts

*mounts namespace*s were introduced to Linux by Al Viro in 2000. The namespace is essentially a tree (or graph) of mounts. A new mounts namespace is created as a copy of the original, after which the namespaces are completely private: updates in any one namespace are not seen in other namespaces. For several years, this feature saw little use for two reasons: the isolation was too strict, and, for some uses, task creation turned out to be a poor time at which to clone a new *mounts namespace*.

3.5.1 Unshare

In 2004, RedHat and IBM collaborated on an LSPP [17] certified version of the RedHat distribution. As an LSPP system incorporates Multi-Level Security (MLS), a user may log into the same system at varying levels. However, applications expect to be able to share directories such as /tmp and /home. MLS becomes problematic because /tmp and /home/user1 must either be labeled at a high level, in which case user1 cannot read it while at a lower level, or it must be labeled at a low level, in which case the user cannot write while classified at a higher level.

The typical solution to this is directory poly-instantiation, which involves redirecting lookups under /home/user1 to some other directory specific to user1's current security clearance. For instance, while logged in with some clearance X, a lookup under /home/user1/ might be redirected to /home/user1/X/. Mounts namespaces lend themselves to a novel method of providing directory poly-instantiation. A user's login process spawns a new *mounts namespace* and mounts /home/user1/.X/ onto /home/user1. Ideally this would be done in a PAM module [9], but using clone() to create a new *mounts namespace* from PAM is not possible. Clone only places the new task in the new *mounts namespace*, while the calling process remains in the original namespace.

To address this, the unshare() system call was introduced [4]. This call creates a new *mounts namespace* and attaches it to the calling process. A PAM library function can unshare its *mounts namespace*, and when the library function returns the calling process will find itself in the new namespace.

3.5.2 Mounts Propagation

The problem remained that after login, the user's mounts tree was completely isolated from the system's mounts tree. So if a system administrator were to mount a new NFS filesystem after a user has logged in, or the user inserts a CD-ROM expecting a running automounter to mount the media, the user would not see the new filesystems.

The problem was finally solved using a design by Al Viro and implementation by Ram Pai of mounts propagation [45, 13]. A relationship is introduced between mount points: a mount can be peer, slave, or unrelated to another mount. Two unrelated mounts do not share mount events. If two mounts are peers, then a mount event under one is propagated to the others. If one is a master to others, then mount events under the master are propagated to its slaves.

Figure 1 illustrates the motivating example solved using mounts propagation. User smith has logged in, and the system has created a new *mounts namespace* for his login process. It then bind-mounted the directory /tmp/.priv/smith onto /tmp in the new namespace, leaving /tmp in the original namespace untouched. After his login, the automounter, running in the initial *mounts namespace*, responded to a CD-ROM insertion by mounting the CD-ROM. In figure 1(a), the mounts trees in the two namespaces are unrelated, so the mount does not appear in smith's namespace.

In figure 1(b), / was made a recursively shared tree during boot. After creating smith's new *mounts namespace*, the login program made / in the new namespace recursively slave to the original namespace. The subsequent binding of /tmp/.priv/smith onto /tmp is not reflected in the master namespace, but a later mounting of the newly inserted CD-ROM is reflected in smith's namespace.

This example demonstrates mounts propagation between two namespaces, but the propagation relationships are actually defined between mount points. Instead of creating a copy of a mounts tree by creating a new namespace, we could do so using by recursively bind-mounting the mounts tree. Mount points in the original and new mounts trees would have the same relationships as with a full namespace clone, and the same mount propagation semantics would hold.

3.6 PID

The essential requirement for *pid namespace* is similar to that of other namespaces - to enable virtualization, the pids in one *pid namespace* must be independent of pids in other namespaces. And to enable ACR, the pids must be selectable within a new namespace.

But to properly monitor all activity in a system, an administrator must be able to view and signal processes in all *pid namespaces*. This brings up a second requirement in that existing utilities like ps, top, kill etc must still be usable to monitor and control the entire system.

A simple implementation of *pid namespace* would provide completely disjoint *pid namespaces* akin to the *IPC namespace* semantics. Upon a clone(NEWPID), the process would receive pid 1 in a clean empty *pid namespace*. While such an implementation would meet first requirement above, it would not meet the second requirement. Administrators would require additional tools and mechanisms, such as modified waitpid() semantics, to monitor the processes in different pid-namespaces.

More adequate but still simple *pid namespace* semantics would enforce a strict two-level hierarchy. This could take several forms, but one often-mentioned form and as implemented in Vserver, Zap and OpenVZ, would be for processes to have one "real" pid, and potentially one 'virtual' pid.

A process in a private *pid namespace* would only see other processes in its *pid namespace*, and would know them by their virtual pid. A process in the initial *pid namespace* would see all processes, and know them by their real pid. This approach would suffice for simple VPS and ACR implementations. However, the intent was for Linux to support both, to the point of supporting a VPS VS_1 nesting another VPS VS_2, under which a batch job was running with private namespaces to support migrating the batch job. The batch job, VS_2, VS_1, and the whole system each must have a private *pid namespace*.

3.6.1 Nested pid namespaces

This requirement of nested namespaces makes *pid namespace* the most complicated namespace in terms of semantics.

An administrator on the native system must be able to view all processes. An administrator in VS_1 must be able to view all processes in VS_1, including processes in VS_2 and the batch job, but must not be able to see tasks outside VS_1. Similarly, an administrator in VS_2 must be able to view exactly all jobs in VS_2 and the batch job, but no others. A process in the batch job should only see its own tasks. Finally, any of the batch job, VS_2, or VS_1 should be migrate-able, meaning that the pid of a process in the batch job must be both unique and selectable upon restart at each of the three lower levels - in VS_1, VS_2, and the batch job's namespace.

The above example describes precisely the semantics as implemented [7]. The *pid namespaces* form a simple tree headed by the initial *pid namespace*.

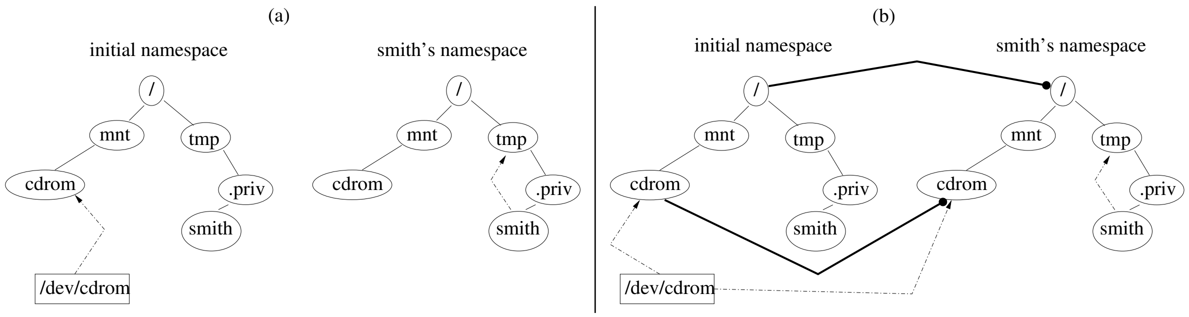

Figure 1: Simple use of mounts propagation.

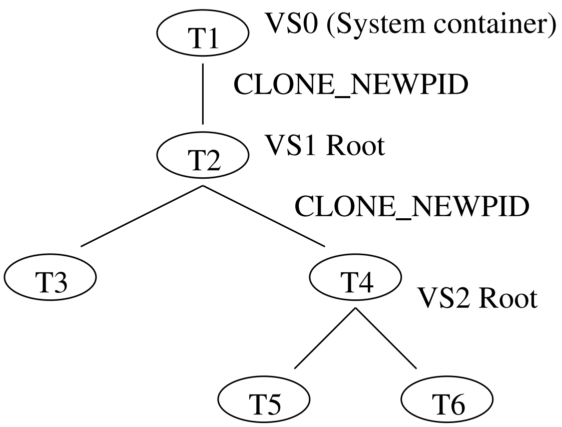

Figure 2: A sample process tree with tasks in 3 pid namespaces.

Task	Virtual and **Real** Pids
T^1	{ $\mathbf{P^1_0}$ }
T^2	{ P^1_1, $\mathbf{P^2_0}$ }
T^3	{ P^2_1, $\mathbf{P^3_0}$ }
T^4	{ P^1_2, P^3_1, $\mathbf{P^4_0}$ }
T^5	{ P^2_2, P^4_1, $\mathbf{P^5_0}$ }
T^6	{ P^3_2, P^5_1, $\mathbf{P^6_0}$ }

Figure 3.6.1 shows a sample process tree with processes in 3 *pid namespace*. In the table P^i_v denotes pid of task `i` in namespace `v`.

Task T^1 is in the initial *pid namespace* (VS_0) and it has a single pid, P^1_0. It clones a task T^2 with the `CLONE_NEWPID` flag. This creates a new *pid namespace*, VS_1 and a new task T^2. T^2 has two pids, P^1 in VS_1 P^2 in VS_0. We represent these multiple pids of T^2 as { P^1_1, P^2_0 }.

Similarly when task T^2 clones a task T^4 with `CLONE_NEWPID` flag, a third *pid namespace*, VS_2 is created. Task T^4 then has 3 pids, one in each of VS_2, VS_1 and VS_0 - { P^1_2, P^3_1, P^4_0 }

The *pid namespaces* VS_0, VS_1, VS_2 are themselves hierarchical and each *pid namespace* is fully contained in a parent pid-namespace. In general, a process has a pid, and is visible in, each ancestor *pid namespace* but the process is not visible to any process in a descendant *pid namespace*.

So an administrator in *pid namespace* VS_0 can see all processes in the system but an administrator in *pid namespace* VS_1 can only see processes in *pid namespaces* VS_1 and VS_2. Since the initial process in a *pid namespace* appears as a normal process to its parent process even though the parent process is in the parent *pid namespace*, the parent process could continue to use `waitpid()` to wait for the child and no special handling is required.

For simplicity, we do not allow unsharing pid-namespaces. Also for simplicity, when a *pid namespace* is terminated, all its descendant *pid namespaces* are also terminated. So the list of *pid namespaces* that a process is visible in is known at the time of process creation and does not change during the life of the process.

3.6.2 `struct pid` *and* `upid_list`

Some kernel subsystems need to uniquely identify the user or owner of certain resources. Using pids for this is vulnerable to pids wrap-around, in which the pids is reassigned to a new and unrelated process. The Linux kernel uses a `task_struct` to represent an active process in the system and an obvious solution to the pid wrap-around problem would be for the subsystems to hold a reference to the `task_struct` until the pid can be freed. But the `task_struct` is too large to be kept around long after the task exits. Therefore a `struct pid` was introduced. It is small enough to be kept around until all references to the task are dropped, preventing wrap-around problems.

A `struct pid` uniquely represents a pid in the system and refers to tasks, process-groups, and sessions. There is a single `struct pid` for each `task_struct` (or active process or group or session) in the system. The `struct pid` also serves as a proxy to the different pids a process has in different namespaces. These pids are plain numbers and are typically referred to as `upid` (short for user-pid) or `pid_ts` in contrast to the `struct pid`.

3.6.3 /proc pseudo filesystem

Another challenge we faced during implementation of *pid namespaces* was that utilities like `ps` refer to `/proc` to list active processes in the system. But for an administrator in VS_2 to run `ps` and see only processes belonging to VS_2, the contents of `/proc` in VS_2 must be different from the contents of VS_1.

A simple and obvious solution for this problem would be to filter the pid entries in /proc depending on the whether the process reading `/proc` is in VS_1 or VS_2. But if two processes, one in VS_1 and other in VS_2 simultaneously read the `/proc/p1` directory, they would expect to see information about two different processes. This would require the

/proc pseudo-fs to invalidate the directory entry (dentry) cache after each read i.e repeated accesses of /proc/p1 directory from different namespaces would result in unnecessary thrashing in the dentry cache.

To avoid such thrashing and resulting performance loss, each /proc mount is associated with the *pid namespace* of the task which performs the mount. As a result, a task cloned into a new *pid namespace* must remount /proc. Due to race conditions resulting from the tight coupling between process creation, process termination, and the /proc filesystem, we still must mount and unmount the /proc filesystem in the kernel while creating and destroying *pid namespaces*.

3.7 Network

Networking protocols are normally developed in layers, with each layer responsible for a different facet of the communications [40]. A protocol suite, such as TCP/IP, is the combination of different protocols at various layers. TCP/IP is normally considered to be a four-layer system.

A full featured VPS should provide network isolation and virtualization. This secures a VPSs communication from other VPSs, allows the same networking application (eg. apache, sshd) running in different VPSs without conflict, manage network resources per VPS and group these resources with the VPS for ACR.

The link level and transport level isolation and virtualization are used in the different VPS implementation. The first one provides a full network stack virtualization and the second one ensures isolation at the network level. Each approach has its set of pros and cons. The link level isolation provides a full network stack but the virtualization must be implemented in all the protocols in the kernel (network and transport) while the network isolation is easier to implement because it is focused in the network level but the restrictions due this approach reduces considerably the use cases.

The Linux network isolation implementation acts at the link layer. Each *network namespace* has its own set of network devices and its own network stack. This allows a VPS to fully control networking through the existing tools, and configuration can be done the same way inside a VPS as on a normal (non-VPS) host without any special VPS knowledge, hence a VPS can use unmodified networking startup scripts from a distribution.

3.7.1 The Network virtualization

XXX Each *network namespace* has its own set of network devices. Any network device except loopback or tunnels can be *migrated* between namespaces. Of course the number of containers is expected to outnumber the number of physical network devices, so a new network device, called veth or "network pair device", has been introduced to facilitate communication between namespaces. One task creates a device pair, then migrates one of the devices to another namespace. A packet sent in one side is received at the other side.

The instantiation of the veth virtual network driver creates two network interfaces linked together, acting like a ethernet tunnel. Moving one side of this tunnel to a namespace allow to have it to pass network traffic across namespaces : each time a packet goes through this interface, it will be received by the other side of the network device, that is outside of the namespace.

To communicate with the outside world, the usual configuration is to have a host configured with a bridge and the physical network device, and to attach the veth side to the bridge. Other configurations are possible using NAT or routing, but these would quickly complicate configuration of a host with many VPSs.

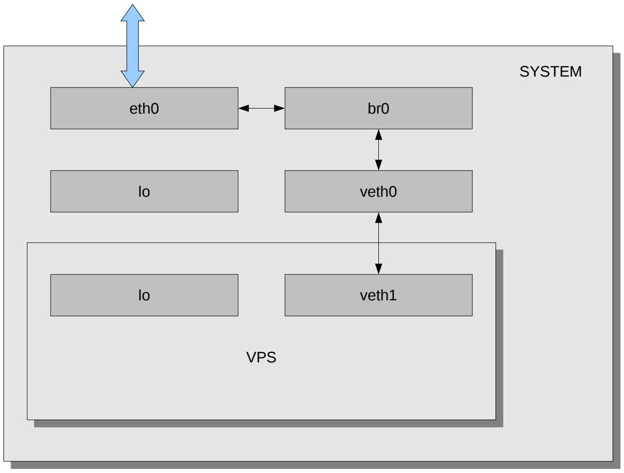

Figure 3: One VPS configured on the system

3.7.2 The network namespace infrastructure

To support per-namespace network resources we must handle the fact that network components can be initialized (or removed) at any time during the system life cycle. For example, the ipv6 module can be loaded after some *network namespaces* have been created. The *network namespace* infrastructure provides a subsystem registration and a *network namespace* tracking. When a subsystem is loaded, it is registered for every *network namespaces* and each one will use the initialization routing of this subsystem to initialize its own network stack. This infrastructure ensures that the namespace will be safe for module loading and unloading.

The *network namespace* life cycle differs from the other namespaces. The *network namespace* is deeply refcounted in the network stack and introduces asynchronous destruction of network resource related to a namespace. For instanc, a process having sent data through a TCP socket may exit while the network stack buffers the data it is transmitted to the peer. During this time the namespace must stay alive.

3.7.3 Virtualizing the link layer - L2

The first thing to do for the *network namespace* is to create a new loopback instance and make it to belong to the namespace. The network stack originally had a single system-wide global variable for the loopback. The loopback is now dynamically allocated. At the namespace creation, the loopback is instantiated like any usual network device. This modification was positively received because the community was looking for the loopback to be treated like any other network device.

A function to move a network device between namespaces has been implemented. This function takes the pid of a process belonging to the *network namespace* and the network device to be moved as parameters. The *network namespace* is retrieved from the task, which is identified by its pid, and used to register the network device. When the *network* *namespace* is destroyed, all the network devices are moved to the default network namespace init_net. If a name conflict occurs, the kernel will assign another name to the network device to be reassigned to `init_net`.

3.7.4 Resources around the network

All the network resources an user can interact must be taken into account, that implies to handle the *proc* and the *sys* file systems

- `/proc/net` : Each *network namespace* has its own `/proc/net` directory. This increases security and allows networking tools to work within the calling process' *network namespace*, like the `netstat` or `ifconfig` commands that look at `/proc/net/dev`.
- `/proc/sys/net` : Each *network namespace* can configure at least a subset of the `sysctls`. Tasks should not be able to change the configuration of other namespaces.
- `sysfs` : The network devices are isolated. The `sysfs` tree must reflect the view of the network devices regarding with the network namespace isolation. For example, `/sys/class/net` should show only the network devices belonging to the namespace.

3.8 Userid

As discussed in Section 1.1, in order to support VPSs, it must be possible to isolate users in different servers. However, UNIX uses integer user IDs (uids) to identify users and integer group IDs (gids) to group uids, and these integer values are used to store ownership information on persistent filesystems. Short of forcing all VPS administrators to cooperate in assigning uids and gids which are unique throughout the whole system, it becomes necessary to turn the user table into a namespace. As of this writing, only the simplest part, an actual per-process *uid namespace* providing only separate per-user account, has been implemented. The semantics are currently akin to those of the *IPC namespace*. That is, *uid namespaces* are completely disjoint, and are created empty save for an entry for the root user.

While the design for the completion of *uid namespace*s is in progress as of this writing, the following hypothetical design is presented here as an embodiment of the requirements that we intend to satisfy.

The uid to user mappings will continue to be disjoint, but we must provide a mechanism to meaningfully and unambiguously refer to a user in order to store ownership information for files on persistent file systems. To that end, two changes will be made. First, a user is said to own any *uid namespace*s which it has created. Second, a *uid namespace* can be identified using a "Universal NameSpace IDentifier", or `unsid`. The `unsid` lets us sanely correlate a *uid namespace*, which is ephemeral, to a disk store, which is persistent. So every time a VPS is restarted or an ACR job is migrated, the corresponding user namespace can be tied to the same unsid. The `unsid` is initially NULL (invalid), and setting the `unsid` is a privileged operation. A task with an invalid `unsid` receives user `nobody` permissions to all files, and may not create any files.

A user Sallie can trace or send signals to any processes which belong to a *uid namespace* she owns. A root process only has privilege over tasks and files belonging to the `unsid` in which the process is root. When a file is created, the current uid and `unsid` are stored, as are the uid and `unsid` of the owning user, and so on up to the uid and `unsid` of the user in the initial *uid namespace*. In this way, user Sallie owns all root-owned files belonging to a VPS which she created, while being root in her VPS does not allow her to read root-owned files in another VPS or on the host system.

Until the implementation of *uid namespace*s is more mature, the SELinux [25] or Smack [34] security modules can be used to isolate VPSs. Both SELinux and Smack label processes and files with security contexts. By assigning different security labels to different VPSs, a VPS can be forbidden from signaling tasks in another VPS or reading its files. However this is not a sufficient solution, if only because some sites do not wish to incur either the burden of security policy maintenance or the run-time performance cost [2].

3.9 Devices

One of the remaining unsolved problems is how to do deal with devices, particularly when applications migrate. The kernel provides several kinds of logical devices that are disconnected from real hardware: pseudo terminals [39], loopback devices, `/dev/null`, etc. Such devices should always be available after a migration event. Pseudo terminals are most significant as they can be used by unprivileged user processes and are used heavily in day to day applications.

With the static device configurations of the past there are workarounds can be employed, such as only creating those devices that a container should use or using a device access white-list [12]. As devices get more dynamic and the generic device layer gets ever richer, we need to properly isolate the device layer into its own namespace, to allow for solid VPS and application containment and migration. We expect this namespace to virtualize the mapping of device numbers to devices. Complications will include filtering the visibility of devices in `sysfs`, and filtering the device events sent to processes in a container.

3.10 Security

Whenever possible, design decisions were made with a long-term hope of allowing unprivileged unsharing of namespaces. However, until the semantics of all namespaces are finalized and more experience with them has been gained, unsharing of any namespace requires privilege. The precedent for requiring such privilege started with the *mounts namespace*. This may appear odd since Plan9 has no such requirement. The primary reason why unsharing a *mounts namespace* may not be safe in Linux is the added "feature" of setuid root binaries, which allow any user to execute programs with privilege. By executing such programs in a private namespace, it may be possible to confuse these programs, potentially corrupting the system or even gaining full superuser privilege. Work is being done to allow the manipulation of parts of the mounts tree without privilege [42], but addressing unprivileged unsharing of *mounts namespace*s is work which remains to be done.

4. PERFORMANCE

While some small amount of performance impact is to be expected by adding extra layers of indirection to translate

[2] The latest reported overhead incurred by SELinux was approximately 7%, while there are no known Smack performance measurements as of yet.

IDs through namespaces, the precise impact of the namespace work in Linux is impossible to measure. The code implementing namespaces is fundamental and can not be easily removed. However, it was done cleanly enough and efficiently enough that it is not deemed a problem.

While it is possible to configure a kernel with all or some namespaces disabled, this does not fully disable namespaces. It mainly disables the cloning of namespaces, so that administrators can disable them until they are deemed more mature. Nonetheless, a small amount of reference counting required to manage the life-cycles of the objects representing namespace instances can be eschewed when a namespace is disabled. Compiling out namespace support can also slightly decrease the size of a kernel by allowing some setup and shutdown code to be compiled out, which is favored by much of the embedded community. However, since the much more frequently-exercised code to look up a resource by its ID still flows through the regular namespace lookups, we would expect essentially no performance difference with our without namespace support. Likewise, in order to provide a clean conceptual model of namespaces, the ”initial” namespace is generally no different from any other namespaces, so an application running in a container should suffer no performance degradation compared to an application which is ”not in a container.”

Showing the true cost of namespaces therefore would require testing a modern kernel to one predating namespaces. This of course is not feasible as these two kernels would be different in virtually all respects.

The *network namespace* is unique from the other namespaces in that expected usage will be for the initial namespace's network devices to be physical, while other namespaces will use virtual network devices. Virtual devices will need to pass data through physical ones, which could impact network performance.

We can pass physical devices between *network namespaces* just as we can virtual devices. We did just that in order to show the effect of the namespace code without the processing overhead of a virtual device and bridge.

These tests were made with the `netperf` tool `ftp://ftp.netperf.org` with two hosts on the same network and with network gigabyte offloading capable cards. The hosts are identical, the first one is installed with a Fedora Core 8 and the second one with a Fedora core 8 + the netns kernel. The `netperf` benchmarking program has been run on the system using a kernel with and without the *network namespace* compiled in, followed by a run inside a *network namespace* using a physical network device and, finally, using the virtual network device veth. For each scenario, we want to measure the throughput and the cpu usage. The throughput shows if there is a bottleneck in the packets trip through the namespaces, the cpu usage points if there is more processing for the packets.

More scenarii, details and results are available at `http://lxc.sourceforge.net`

There is no performance degradation when comparing the results of the benchmarking using this physical device inside and outside the namespace. If the *network namespace* code is not compiled in, the benchmarks shows there is no difference with the results when the *network namespace* code is compiled in. The conclusion of these results is that the *network namespace* code does not add extra overhead.

In the case of the `veth` usage, the results show no degra-

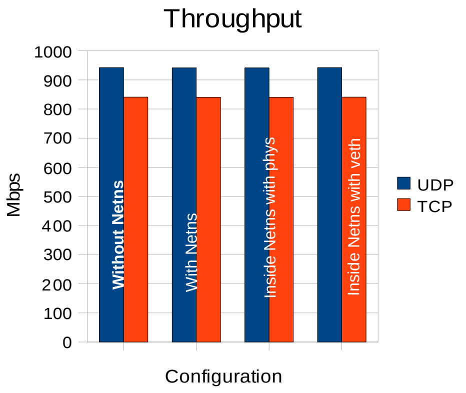

Figure 4: Throughput - Higher is better

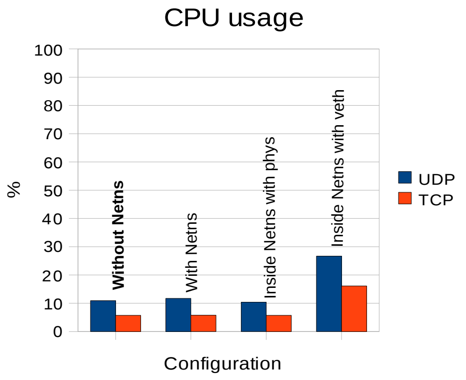

Figure 5: CPU usage - Lower is better

dation with the throughput, but a significant processor overhead. This is directly related with the network configuration and the different paths used by the packets through the network layers due to the `veth`. If the physical device does offloading, the performance overhead is more significant because veth does no checksum offloading and the network stack inside the namespace is not aware of the physical device hardware capabilities. This issue is more related to an optimization area than a *network namespace* design. It is been already spotted for Xen [30] and optimized later [29].

5. CONCLUSION

Virtual Private Servers and Application Checkpoint and Restart have historically been advanced Operating System features and not generally available in common end-user systems. With the implementation of namespaces for kernel-provided resources in Linux, groundwork has been laid for common availability of both features. While much work remains in order to provide both features, the acceptance of this work and commitment to continued progress on remaining work, the promise of fully migrate-able VPSs on end-user machines becomes a very real possibility.

6. LEGAL STATEMENT

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM is a registered trademark of International Business Machines Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries .

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

7. ACKNOWLEDGMENTS

The authors would like to thank Heather Crognale, Michael Halcrow, anonymous ACM reviewers, and our Sigops paper shepherd, Oren Laadan, for valuable feedback on early drafts.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

8. REFERENCES

[1] Oren La'adan Amnon Barak. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13:361–372, 1998.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM symposium on Operating systems principles*, 2003.

[3] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. *Usenix Annual Technical Conference*, 2005.

[4] Jonathan Corbet. A System Call for Unsharing. `http://lwn.net/Articles/135321/`, 2005.

[5] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[6] Jason Duell, Paul Hargrove, and Eric Roman. The Design and Implementation of Berkeley Labs Linux Checkpoint/Restart. `http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml`, 2003.

[7] Pavel Emelyanov and Kir Kolyshkin. PID Namespaces in the 2.6.24 Kernel. `http://lwn.net/Articles/259217/`, 2007.

[8] Bryan Ford, Mike Hibler, Jay Lepreau, Patric Tullmann, Godmar Back, and Stephen Clawson. Microkernels Meet Recursive Virtual Machines. *Proceedings of the Second Symp. on Operating Systems Design and Implementation*, pages 137–151, 1996.

[9] Kenneth Geisshirt. *Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers*. Packt Publishing, 2006.

[10] Cedric Le Goater, Daniel Lezcano, Clement Calmels, Dave Hansen, Serge Hallyn, and Hubertus Franke. Making applications mobile using containers. *Proceedings of the Ottawa Linux Symposium*, pages 347–367, 2006.

[11] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–45, June 1974.

[12] Serge Hallyn. cgroups: Implement Device Whitelist LSM. `http://lwn.net/Articles/273208/`, 2008.

[13] Serge E. Hallyn and Ram Pai. Applying Mount Namespaces. `http://www.ibm.com/developerworks/linux/library/l-mount-namespaces.html`, 2007.

[14] IBM. IBM Workload Partitions Manager for AIX. `http://www-03.ibm.com/systems/p/os/aix/sysmgmt/wpar/`.

[15] IBM. Customer Engineering Announcement: IBM System/360. `http://archive.computerhistory.org/resources/text/IBM/IBM.System_360.1964.102646081.pdf`, 1964.

[16] Cray Inc. *Cray X1 System Overview - S-2346-23*. Cray software distribution center, 2002.

[17] NSA Information Systems Security Organization. Labeled Security Protection Profile. `http://www.commoncriteriaportal.org/files/ppfiles/lspp.pdf`, 1999.

[18] Ganesh Venkitachalam Jeremy Sugarman and Beng-Hong Lim. Virtualizing I/O Devices on VMWare Workstation's Hosted Virtual Machine Monitor. *Usenix Annual Technical Conference*, 2001.

[19] Poul-Henning Kamp and Robert Watson. Jails: Confining the Omnipotent Root. *SANE*, 2000.

[20] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. *Proceedings of the Linux Symposium*, 2007.

[21] Oren Laadan and Jason Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. *Usenix Annual Technical Conference*, pages 323–336, 2007.

[22] Charles R. Landau. The Checkpoint Mechanism in KeyKOS. *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, september 1992.

[23] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. `http://www.cs.wisc.edu/condor/doc/ckpt97.pdf`, 1997.

[24] Miron Livny and the Condor team. Condor: Current Limitations. `http://www.cs.wisc.edu/condor/manual/v6.4/1_4Current_Limitations.html`.

[25] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. *USENIX Annual Technical Conference, FREENIX Track*, pages 29–42, 2001.

[26] Michael MacIsaac, Mike Duffy, Martin Soellig, and Ampie Vos. S/390 Server Consolidation - A Guide for IT Managers. `http://www.redbooks.ibm.com`, October 1999.

[27] John Mehnert-Spahn. Container Checkpointing. `http://www.kerrighed.org/docs/KerrighedSummit07/JM-Container_Checkpointing.pdf`, 2007.

[28] Paul B. Menage. Adding Generic Process Containers to the Linux Kernel. *Proceedings of the Ottawa Linux Symposium*, 2007.

[29] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing Network Virtualization in Xen. `http://www.usenix.org/events/usenix06/tech/menon.html`, 2006.

[30] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. http://www.usenix.org/events/vee05/full_papers/p13-menon.pdf, 2005.

[31] Rob Pike, Dave Presotto, Ken Thompson, Howard Tricke y, and Phil Winterbottom. The Use of Name Spaces in Plan 9. *Operating Systems Review*, 1992.

[32] Daniel Price and Andrew Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. *Usenix LISA*, 2004.

[33] Rusty Russell. Lguest: The Simple x86 Hypervisor. http://lguest.ozlabs.org/, 2007.

[34] Casey Schaufler. The Simplified Mandatory Access Control Kernel. http://linux.conf.au/programme/detail?TalkID=92, 2008.

[35] Brian K. Schmidt. Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches. http://www-suif.stanford.edu/~bks/publications/thesis.pdf, August 2000.

[36] Jason Nieh Shaya Potter and Matt Selsky. Secure Isolation of Untrusted Legacy Applications. *Usenix LISA*, 2007.

[37] Stephen Soltesz, Herbert Potzl, Marc Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. *ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 275–287, 2007.

[38] Clifford Spinac. Dynamic logical partitioning for Linux on POWER. http://www-128.ibm.com/developerworks/systems/library/es-dynamic/, 2005.

[39] Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[40] Richard Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Indianapolis, 2001.

[41] SWSoft. OpenVZ User's Guide. http://download.openvz.org/doc/OpenVZ-Users-Guide.pdf, 2005.

[42] Miklos Szeredi. Mount Ownership and Unprivileged Mount Syscall. http://lwn.net/Articles/273729/, 2008.

[43] Kerrighed team. Kerrighed. http://www.kerrighed.org/wiki/index.php/Main_Page, 2008.

[44] Bill Tuthill, Karen Johnson, and Terry Schultz. *IRIX Checkpoint and Restart Operation Guide*. SGI Technical Publications, 2003.

[45] Al Viro. [RFC] Shared Subtrees. http://lwn.net/Articles/119232/, 2005.

[46] Victor C. Zandy. ckpt - Process Checkpoint Library. http://pages.cs.wisc.edu/~zandy/ckpt/README.