Scaling With Multiple Network Namespaces in a Single Application

PJ Waskiewicz

NetApp
Portland, OR, USA
pj.waskiewicz@netapp.com

Abstract

Namespaces and containers are growing in popularity, but it is rare for large applications to use them directly. Rather, applications rely on frameworks such as LXC (Linux Containers Project) and/or Docker to manage the containers they run in.

This paper will focus on how large applications can utilize the network namespace framework, and a large number of namespaces can be used within an application to partition up the underlying network infrastructure. An overview of parts of an application architecture using namespaces will be covered, showing the use cases driving the need for namespaces. Lessons learned around scalability and performance bottlenecks in the kernel will be shared.

Ultimately this paper will propose further improvements to the namespace framework for better programmatic management of namespaces within the kernel from userspace, as well as attention to increased scalability and efficiency of networking within the namespaces.

Keywords

networking, kernel, containers, namespaces, scaling, cloud

Introduction

The ongoing evolution of the datacenter towards cloud-based infrastructure continues to present interesting challenges to existing applications. These existing solutions (e.g. storage appliances) work to serve multiple tenants within a computing domain. However, when the infrastructure around these solutions evolves into a cloud-based, partitioned environment, these solutions must also evolve.

When an application has core logic that needs to span multiple, separate network environments, it must become container/network-namespace aware. The Linux kernel exposes an API to create and manage network namespaces within an application. This paper will focus on this API for the following:

- How to create and manage the lifecycle of a network namespace within a complex application
- How to track and process multiple network connections across multiple network namespaces in an efficient and scalable manner
- What limitations exist in this API that makes lifecycle management a challenge, along with proposals on how to improve the API for better lifecycle management
- Scalability issues encountered, how these were addressed, and proposals around scalability testing to prevent regressions

The Need For Multiple Network Namespaces

Typical network-based storage applications have various layers of the core logic separated from one another. In the example shown in Figure 1, the volume database containing metadata mappings to underlying block or object data is completely separated from the networking core, since they share no common functionality. Core logic for volume tracking in one functional area, thread management in another area, the iSCSI or Fibre Channel protocol stack handled elsewhere, and ultimately the underlying networking being handled by the underlying OS.

Consider a public or private cloud environment, where ”legacy” datacenter models migrate pieces of their infrastructure into that cloud environment. A typical storage application as Figure 1 could have all storage volumes from that legacy datacenter migrated as one tenant. However, multiple tenants may have conflicting environment configurations, such as IP address range collisions, different routing hierarchies, or different VLAN segregation within their respective network domains. By creating a completely new network stack inside of a network namespace for each of these legacy environments, the containers can fully segregate the network environments from one another, and avoid any changes to the legacy datacenter configuration.

When an application architecture such as this needs to fit into an environment where the networking is the component that needs to support multiple containers, one can observe that using Docker or LXC is not sufficient. They are limited to containerizing an entire application, not pieces of an application. Let us consider some different approaches to try and solve this problem of supporting multiple network stacks within an application.

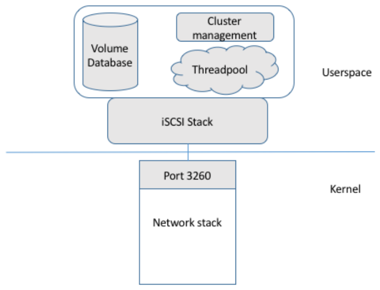

Figure 1: High-level architecture of application.

Approach 1: Fork application per-network namespace

Each network namespace that is created within an application needs to have some mechanism to process traffic inbound and outbound to it. One approach is after creating the namespace and configuring it, fork() the application completely, and assign a fork()'d copy to each namespace configured.

This approach has serious limitations. As shown in Figure 2, the core logic of the application needs to have added inter-process communication (IPC) to manage the non-networking pieces of the internals. This adds a high degree of complexity, and impacts the entire application, instead of just the underlying networking pieces. In addition, applications like this are typically complex systems, and require significant system resources such as CPU and RAM. Having multiple copies of the entire application running on a system is completely infeasible.

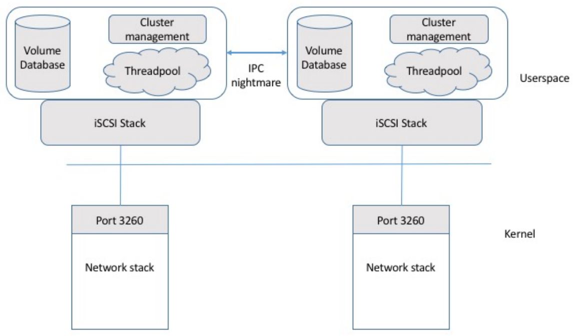

Figure 2: Multiple network namespaces with fork().

Approach 2: Thread per-network namespace

A different approach is to create and assign a new thread (LWP) to each network namespace after creation. This thread would then live in that network namespace context for its lifetime, and could process any inbound and outbound traffic to and from the namespace.

As shown in Figure 3, this approach does not affect the core logic of the application. It only requires a new thread per-network namespace, and only affects the underlying networking core of the application. However, this approach does not scale well with large numbers of network namespaces. In highly complex applications that are multi-threaded, creating dedicated worker threads for specific tasks is usually not desirable. In addition, a single thread per-network namespace severely limits the processing capabilities of that network interface, especially if the underlying hardware is multiqueue-capable.

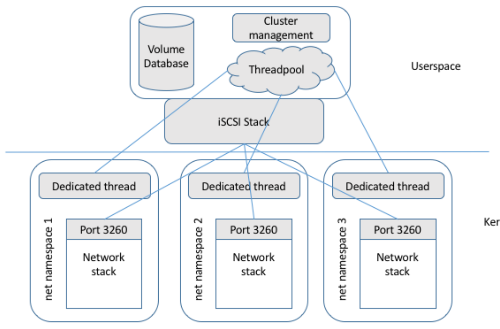

Figure 3: Multiple network namespaces with dedicated threads.

Approach 3: Global file descriptor per-network namespace

A final approach (post-namespace creation) is to switch a thread into the namespace, create a socket and bind the desired listening port inside of the namespace to that socket. Because file descriptors are not specific to each network namespace, all threads in an application have access to these file descriptors. When data needs to be read or written to a socket in a specific namespace, the currently-running thread can switch into that namespace context, and then access the socket using the global file descriptor.

As illustrated by Figure 4, each network namespace stack opens port 3260 (iSCSI), and has a file descriptor that is unique to the system assigned to it. This file descriptor is stored with the socket information, giving the core application logic the ability to switch in and out of each network namespace as needed. This does not require any additional threads or processing units, and can allow the core logic to be completely unaware of the underlying network segregation.

Network Namespace Management

It is critical for one to understand how to create and manage network namespaces at a basic level[1] prior to incorporating them into an application. Once one is familiar with how to manage network namespaces from an administrative perspective, then diving into the programmatic side is the next logical step[2].

The biggest challenge with integrating network namespaces into an existing application is managing their lifecycle efficiently. Distinguishing newly-created network namespaces from recently destroyed network namespaces, how

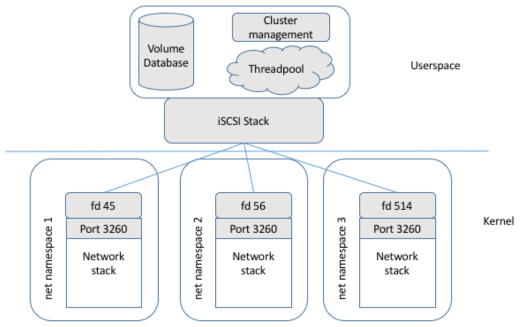

Figure 4: Multiple network namespaces with file descriptors.

to switch into a namespace, and how to switch to the default/base namespace, are all things that need to be managed.

Identifying Different Namespaces

One challenge with network namespace management is how to distinguish a network namespace beyond just its name. Each network namespace is accessed by a file descriptor returned by opening /run/netns/*namespace name* or by opening /proc/*pid*/ns/net, where *pid* is a process that currently lives inside that network namespace. In order for a complex application to have a chance at distinguishing between namespaces, one reference to each network namespace must be maintained for the life of that namespace; i.e. one file descriptor for each namespace must be maintained for all threads.

However, this is not sufficient to uniquely identify one network namespace from another. If one network namespace is destroyed and another quickly created, the file descriptor returned from open(/run/netns/*new namespace*) may be the same that was used to track the previously destroyed namespace. This can cause very unexpected results if the networking core of the application is only using file descriptors to track the namespaces.

The solution is to associate an additional piece of information with the namespace descriptor to ensure a ”fairly” safe match. By adding the mtime of the kernel handle that /run/netns/*namespace name* is backed by from stat(), this can provide sufficient granularity that very quick destruction and addition of namespaces can be distinguished from one another. The mtime in this case is the same as when the special file was created, and is not updated by the kernel if changes are made to the namespace. This makes it a great candidate to help with mostly unique indentification of each namespace along with its single file descriptor reference.

Switching Threads Between Namespaces

When using the approach in Figure 4 to create and manage network namespaces, it is very simple to switch into the namespace. A simple call to open() to get the file descriptor of the namespace is needed, then a call to setns() to switch. Refer to Figure 5. This will switch the current thread into the new namespace context, but it will also leave the thread in that context.

```
int ns_fd = open("/run/netns/namespace_1", O_RDONLY);
int rc = setns(ns_fd, CLONE_NEWNET);
```

Figure 5: Basic switching to new network namespace

If the thread needs to be left in a known state, such as returning to the base or default namespace, then see Figure 6. This assumes that PID 1 is in the default namespace of the kernel, or the default for the PID namespace if one is being used. If there is a different default/base namespace to return to, then one would cache a different PID's namespace file descriptor in the same fashion.

```
int base_fd = open("/proc/1/ns/net", O_RDONLY);
int rc = setns(base_fd, CLONE_NEWNET);
```

Figure 6: Basic switching to default network namespace

Efficient Namespace Switching

In order to utilize the same threadpool for any number of network namespaces, decisions need to be made when to switch between namespaces. If the application needs to incur the setns() system call for each network transaction in flight, the context switching can get very expensive, and is more than likely unnecessary. To make selective switching happen, one needs to cache a bit more information.

At this point, the application has cached the global file descriptor for each active network namespace, the global file descriptor for the base namespace, and what namespace each listening network port is assigned to. The last piece of information to cache is what namespace each thread is currently in. That can be stored inside thread-local storage (TLS) for each thread in the application. Using this information, when a socket operation needs to be performed, the namespace information can be pulled from the socket, and the current namespace of the thread in TLS can be compared to the socket's namespace. If they are the same, the setns() call can be avoided. Otherwise, the thread can be switched into that network namespace's context, and then the socket processing can continue.

Network Namespace Scalability

Making good decisions on when to switch a thread between namespaces is only part of the scalability story. The kernel itself also needs to be very efficient when dealing with any number of network namespaces, and how it performs the requested switch. We observed that in older 3.x Linux kernels, switching between namespaces had a massive performance impact. The latency of how long a switch took increased more than exponentially as more namespaces were added. This was due to using RCU protection for the running task's nsproxy, versus using task_lock() (i.e. a spin_lock()). This was addressed in a patch by Eric Biederman [3], and had a drastic improvement in performance.

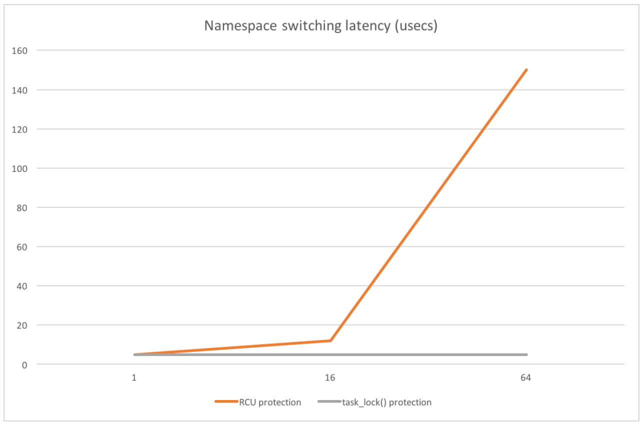

Figure 7: Network namespace switching latency, close-up

In Figure 7, the namespace switching took 5 μsecs with 1 network namespace, before and after the patch. However, approaching 64 network namespaces, pre-patch took an average of 150 μsecs per switch, where post-patch it remained around 5 μsecs. Expanding this out to 512 network namespaces, Figure 8 shows that the switching latency at 512 namespaces pre-patch took over half a second per switch, versus post-patch, the switching latency remains around 5 μsecs.

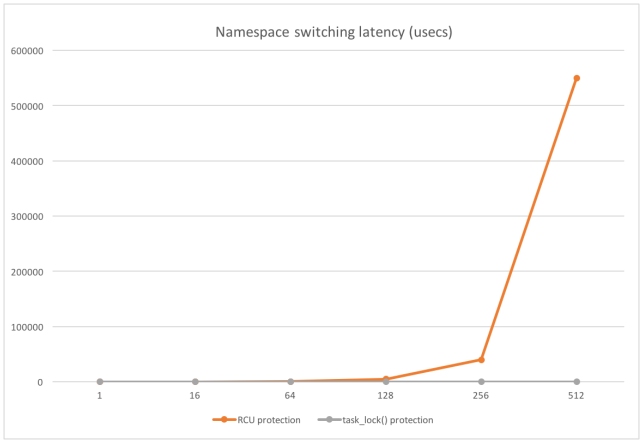

Figure 8: Network namespace switching latency

Proposed Improvements

The integration of network namespaces into our application highlighted the power of Linux kernel namespaces, and how to use them beyond single-use-per-application. However, some workarounds were needed, as discussed in previous sections of this paper. We believe the API and exposed framework can be extended to address some of these issues.

More Uniqueness

The biggest functional challenge with using multiple network namespaces inside of an application was how to identify the uniqueness of a namespace. As covered in Section *Identifying Network Namespaces*, a combination of file descriptor and mtime of the kernel special file for the namespace was ”enough” to identify it. However, in large multi-threaded applications, basing uniqueness on time is never a great idea. A proposal is to extend the metadata for a namespace, perhaps through exposing additional information in /proc, that would identify a namespace with a UUID, or something equally unique. This way the kernel can provide something meant to identify resources from each other, rather than overloading other metadata and hoping for the best.

Performance Regression Testing

As is shown in both Figures 7 and 8, such a small change in the patchset [3] can have a massive impact on scalability. This latency would also not be observed in more traditional container frameworks, since an application is started, moved into its namespace, and lives in that context for the duration of its lifetime. The kernel may observe internal latencies when scheduling that process, but the application would see no latency impact.

Improvements can be made in the kernel build framework to include latency regression tests, much like the RCU torture tests. While this would not necessarily be executed inline as part of a kernel build, it would include necessary framework to provide unit tests to detect if something regressed when someone decided to run the tests.

Conclusion

The network namespace API in the Linux kernel provides a very powerful framework for application developers. It allows large applications that need to span multiple network namespaces to not require complete architectural upheavals of core logic outside of the networking layers. Using some clever tricks and techniques, the namespace switching and lifecycle management can be very efficient. Moving forward, we believe there is still more that can be done to improve this API, and make it even more powerful as containers continue to grow in popularity and wide-spread deployment across the computing industry.

Acknowledgments

We would like to acknowledge the engineers who contributed to this network namespace application design: Carl Seelye, Marshall McMullen, Jared Cantwell, Tom Distler, and Joe Roback.

We would also like to acknowledge the Netdev1.2 selection committee for inviting us to submit and present this paper.

References

[1] Jake Edge 2014. *Namespaces in operation, part 7: Network namespaces*. https://lwn.net/Articles/580893/

[2] Linux man-pages project 2016. *setns(2) manpage, Linux Programmer's Manual*. http://man7.org/linux/man-pages/man2/setns.2.html

[3] Eric Biederman 2014. *namespaces: Use task_lock and not rcu to protect nsproxy*.

https://lists.linuxfoundation.org/pipermail/containers/2014-July/034787.html

Author Biography

PJ Waskiewicz is a Principal Software Engineer at NetApp in the SolidFire division. Prior to SolidFire/NetApp, PJ worked for many years as a network kernel engineer and device driver developer in the Networking Division of Intel. There he maintained and helped create the igb, ixgbe, and i40e wired Ethernet network drivers, the initial Tx multiqueue support in the Linux kernel network stack, and added Data Center Bridging support to the Linux kernel. He also worked in Intel's Open Source Technology Center on the x86 kernel tree, enabling advanced features in the Broadwell and Skylake microarchitectures.