Feather-Weight Network Namespace Isolation Based on User-Specific Addressing and Routing in Commodity OS*

Maoke Chen[1] and Akihiro Nakao[2]

[1] National Institute of Information and Communications Technology (NICT), Tokyo, Japan
[2] The University of Tokyo, Japan

Abstract. Container-based virtualization is the most popular solution for isolating resources among users in a shared testbed. Container achieves good performance but makes the code quite complicated and hard to maintain, to debug and to deploy. We explore an alternative philosophy to enable the isolation based on commodity OS, i.e., utilizing existing features in commodity OS as much as possible rather than introducing complicated containers. Merely granting each user-id in the OS a dedicated and isolated network address as well as specific routing table, we enhance the commodity OS with the functionality of network namespace isolation. We posit that an OS's built-in features plus our feather-weight enhancement meet basic requirements for separating activities among different users of a shared testbed. Applying our prototype which has been implemented, we demonstrate the functionality of our solution can support a VINI-like environment with marginal cost of engineering and tiny overhead.

Keywords: slice computing, name space isolation, socket, networking.

1 Introduction

It is evident that large-scale testbeds in wide-area network such as PlanetLab [13, 14] are viable for bringing innovations in computer networks. The proposal for GENI (Global Environment for Network Innovations) also stresses the significance of building such a testbed. Isolation of resources, i.e., enabling an environment where individual experiments will not be affected by the other ones, is a mandatory feature of such platforms. PlanetLab calls such an isolated set of resources as a "slice".

Virtualization is an instrument to implement isolation. There is a wide spectrum of virtualization technologies. Hypervisor-based solutions, like VMware [6] and Xen [7, 8], provide flexibility for running arbitrary operating systems, but are sub-optimal in scalability due to their computational overhead. A large-scale

* This work has been partly supported by Ministry of Internal Affairs and Communications (MIC) of the Japanese Government.

slice-based network testbed emphasizes the requirement for little overhead in virtualization. Container-based solutions, VServer [3] and FreeBSD Jail [1], satisfy this requirement and have been successfully utilized in PlanetLab and Emulab [12], respectively. OpenVZ [5] and its commercial counterpart, Virtuozzo, also fall into the category of the container-base approaches. Container-based virtualization is considered suitable for a planetary scale test-bed, since it is often “light-weight” and enables over thousands of slices to run concurrently. However, when it comes to “light-weightness”, we must consider not only the low overhead in *performance*, but also for that in *deployment*, including implementation, installation, daily maintenance and continuous upgrading. The PlanetLab kernel is based on VServer that requires a large amount of patches to the stock kernel and the patches may not necessarily keep up with the kernel development. Missing the latest hardware support in the kernel sometimes means we cannot utilize the cutting edge features of processors and devices in the testbed. The desire of having an isolation approach light-weight in both performance and deployment encourages new explorations on virtualization techniques.

We observe that most commodity operating systems already implement security isolation to some extent. It applies file/directory access permissions and disk quotas towards *user id*s, and offers CPU and bandwidth scheduling regarding processes and communication sessions, respectively. In the light of this observation, we pose a question: Can we support isolation only with standard OS features? The term “standard features” means functionality built-in into the current or a near-future native OS. Employing the standard features as much as possible should save the cost for deployment and incur a marginal extra overhead in performance. We elect to add a slight enhancement to the standard feature for network namespace isolation, thus, call our solution “feather-weight” isolation.

The term “network namespace” refers to a set of named entities in the network stack, including network interfaces, IP addresses, ports, forwarding and routing tables, as well as the sockets. It is one of the most important resources to be isolated but the isolation has not yet supported by any of current commodity operating systems. Early PlanetLab once took the *user id* in the commodity OS as the handle for a slice without network namespace isolated, and later the emphasis on security, scalability and enabling dedicated root environment lead to the choice of VServer as a virtual execution environment, where network namespace is isolated within a VServer container.

In this paper, we re-visit this early idea of user-id-based isolation and try to enhance it for network namespace isolation. The key of the solution is assigning IP addresses and policy-based route tables to *user id*s and enforcing separated usage in interface, address, port space, routing and bandwidth. A caveat is, for this feather-weightness, we may have to tradeoff the rigorous and secure root environment isolation. Fortunately, however, in most cases, root environment may not be necessary.

A parallel work, the Linux network namespace (NetNS) [2], addresses the same requirement for network namespace isolation. It clones the network stack into a container that is identified by a process id of a shell, and thus the shell has an independent set of virtual interface, address, port space and socket space. NetNS has to modify the whole network stack and also other parts of OS kernel, such as filesystem. It is based on the container concept that is advantageous in performance but not so in deployment. It makes the NetNS code hard to maintain and to keep up with the evolution of the commodity OS.

This paper makes three contributions. First, unlike NetNS and any other solutions, this paper suggests the philosophy of enhancing commodity OS with isolation without over-engineering. Second, it puts this design philosophy into practice for the network namespace isolation, realizing the mechanism for isolating IP addresses and route tables within a *user id*. Finally, it demonstrates that our prototype implementation enables a VINI-like environment where we can experiment with new network architectures and services with marginal cost of engineering.

The rest of the paper is organized as follows. Section 2 summarizes the problem in enabling network namespace isolation in commodity OS and motivates our work from the observations on the drawbacks of the container-based NetNS. Section 3 identifies what is required and what is obtained with the proposed solution, especially for the separated routing tables. Section 4 demonstrates typical routing experiments based on our solutions and Section 5 presents benchmark for our prototype. Finally, Section 6 briefly concludes the paper with future work.

2 Related Work

We first identify what the current commodity OS supports for isolation and what it falls short of.

2.1 Resource and Namespace Isolation in Native OS

Even without either full virtualization or containers, native OS, either UNIX or modern Windows, facilitates resource and namespace isolation to some extent.

- *Security Isolation:* file systems protect file and directory access among *user ids*. Currently the security isolation in native OS doesn't hide one's processes and traffics from other users.
- *Performance Isolation:* disk quota in commodity OS controls storage usage per *user id*. CPU scheduling is a basic function of any OS regarding process management. Control Groups (*cgroups*) provides the aggregation for process scheduling. *User id* can be used for the progress grouping definitely. For the network bandwidth issue, traffic reshaping is applied for flows. In Linux operating system, the tool "`tc`" coupled with the `iptables` controls the flow according to its attributes like source or destination addresses, not related to *user id* yet.
- *Network Namespace Isolation:* IP addresses and port space are currently shared among *user ids*, without being isolated. Interfaces, routing tables are the same. Policy-based Routing can support multiple routing tables, but the

separation is not per *user id.* Therefore, the network namespace is what today's commodity OS does not support.

2.2 Linux Network Namespace (NetNS)

NetNS [2] is designed in response to the need of enabling network namespace isolation in OS. It clones the network stack into private sets of network resources to be assigned to one or several processes. The resource set includes device, IP address, port space, route tables, sockets and so forth. Each set is assigned to a shell process, which plays the role of the "container". NetNS can be configured with either Layer-3 router or the Ethernet bridging mode.

NetNS focuses only on network namespace isolation and thus it is much simpler than OpenVZ or VServer. However, it is still a container-based solution and accordingly suffers from the drawback of over-engineering. First, NetNS conflicts with some existing components in the commodity OS. A well-known case is *sysfs*, which must be patched until being able to co-exist with NetNS. Second, NetNS takes a process-id of a shell as the handle for a container. It implies a limitation that a "slice" implemented with such a container cannot run multiple shells with the shared network namespace. Finally, the code complexity causes a lot of troubles and bugs that hurt the stability in operation. For example, our trial of NetNS in Fedora 8 Linux with kernel 2.6.26.8, compiled from natively released source code, ends up with a hangup.

This observation on NetNS has motivated our exploration for a better and lighter-weight solution for the network namespace isolation, which incurs less cost not only in performance overhead but also in implementation and deployment than NetNS.

3 System Design

The objective of our system design is enabling network namespace isolation per *user-id* in a commodity OS, treating the *user-id* as the handle for a slice. The term "network namespace isolation" is defined as enabling following features from the bottom of network stack to its top.

R1 A *user-id* is assigned by the system administrator to one (or more) network interface(s), either physical or virtual, and not allowed to use the other network interfaces.

R2 A *user-id* is assigned by the system administrator to one (or more) IP address(es), either physical or virtual, and not allowed to use the other IP addresses.

R3 A *user-id* is assigned by the system administrator to a forwarding table, and not allowed to use the other forwarding table.

R4 A *user-id* is allowed to update the kernel forwarding table associated with this *user-id.*

We also have two constraints for the system design.

C1 Our solution must be transparent to applications. In other words, the existing applications must work without re-code and re-implementation. Binary-compatibility must be held.
C2 Our solution must involve only the networking kernel of the OS, not to conflict with other features or components, unlike NetNS.
C3 Our solution must enable new features, but leave the other existing features as they are.

The last constraint may need clarification; for example, without our solution, only the privileged user, `root`, can modify the route table in the kernel, while, after our enhancement, regular users are allowed to change their own associated route table but what the `root` can do is totally unchanged.

We design the system with three steps to meet the requirements of our design objective. First, we enable the IP address isolation among *user-id*s. Then, we apply PBR (Policy Based Routing) and traffic shaper with our modified OS kernel to make forwarding tables and bandwidth isolated. Finally, we extend the model through associating route table-id with the *user-id*, and separate the route installation from RIB (routing information base) to FIB (forwarding information base), from the user space to the kernel.

3.1 Isolating IP Addresses among Users

[2] Isolation of IP addresses among users needs two components to be added to the kernel: the data structure describing the assignment of addresses for *user-id*s and the functions that ensure the separated address usage. The data structure and the function are referred by socket system-calls transparently to the callers.

The data structure for address assignment to *user-id* is designed in the form of an association of IP address and *user-id*. An entry (*IP_address*, *user_id*) means the *user-id* is able to use the *IP address*. Considering the scalability, the data structure should be designed accessible with a hash function.

Semantically the user-address association changes the meaning of the "unspecified" (or wild-card) IP address, which is defined with the macro IN_ADDR_ANY for IPv4 and IN6_ADDR_ANY for IPv6, respectively. In a commodity OS, the wild-card means any address configured with any interface of the host. In an OS with user-specific address, however, a wild-card means any addresses among those already assigned to the *user-id* of the process owner that raises the communication.

The interpretation of wild-card complicates the port conflict detection. When two sockets attempt to bind to the same port on the wild-card, the conflict happens when the process owners are associated with at least one common address. Therefore, the port conflict detection routine should also check which addresses

[2] The content of this part is covered by our previous work [11], which focuses on the addressing architecture for slice computing but not other isolation issues for network namespace.

are used by each binding socket, i.e., finding out the user-address association for the *user-id* of the owner of each socket.

Packet dispatch from a protocol stack to a socket is also affected by the change of the semantics of wild-card. The protocol stack searches the socket pool for a socket matching the destination (address, port)-pair of a packet. Previously, if a socket is bound to the wild-card, only port should be check for the dispatching. Now the kernel must identify what the wild-card of a candidate socket mean exactly and determine if this socket's corresponding *user-id* really owns the destination address in the packet to be dispatched.

Based on the above data structures, a variety of functions are added into the socket system-calls and protocol stack.

- *Sender address availability check:* when a process tries to bind a specific address for connection to send out packets directly with the specific source address, the kernel checks if the process owner *user-id* is assigned with the address.
- *Listener address availability check:* when a process tries to bind a specific address for a daemon, the kernel checks if the listener process owner *user-id* is assigned with the address.
- *Address selection:* when a process does not specify a source address but lets the kernel choose one for itself in either of the above cases, the kernel filters only those addresses assigned to the process owner *user-id* as the candidate for the selection. More importantly, the commodity system also tries to select the source address "closest" to the destination. Our modified kernel should apply this algorithm with the addresses available for the process owner *user-id*. The address selector also helps protocol stack to identify proper socket that a connectionless datagram or a connection request is targeting.

Per-user address assignment directly results in the elimination of port conflict among *user-id*s as in a commodity OS where users can share the use of a single IP address, since the full port range of the dedicated IP address is available to each user. We may change the minimum available port number for a regular *user-id* from 1024 to 0. This is especially beneficial when each user wants to use the well-known port number of the dedicated IP address, for example, to run BGP speakers with a well-known port number such as 179.

Per-user address assignment is suitable for the circumstances where IP addresses are abundant, either in IPv4 or in IPv6, or in private IP addresses. Note that our proposed system can assign even private IP addresses to each *user-id*. Most virtual networks are using tunnels through private IPv4 addresses. Considering this, our system is especially useful for network virtualization, as demonstrated in Section 4.

3.2 Combining Address Separation with Other Features

Network namespace includes link interface, IP address, ports, routes and bandwidth. After IP addresses have been isolated among *user-id*s, we can also isolate some of other resources among users by configuring those resources with the user-specific addresses.

- *Interface:* interface is configured with one or more IP addresses. Therefore, the use of each interface is isolated per *user-id*. However, only the *use* of the interface is isolated, not its *visibility*—each user can see all the physical and logical network interfaces on the host, knowing their addresses with commands such as `ifconfig` and `ip link`.
- *Routing:* Policy-based Routing (PBR) can be used to define a separate route (more precisely, forwarding) table for packets from a specific source address. Therefore, we can let each *user-id* own and use a separate forwarding table for packets originating from the address the *user-id* is assigned to, rather than share the default table with others. RIB (routing information base) is managed at the application layer. Therefore, it is naturally isolated when being run by specific *user-id*. However, today's commodity OS doesn't allow unprivileged *user-id* to run routing tools and to install RIB to the system FIB.
- *Bandwidth:* Traffic control tools in commodity OS (e.g., `tc` in Linux) support shaping policies with respect to a source or a destination address. Configuring `tc` with the rule specified by user-specific source or destination addresses, we can schedule bandwidth utilization among *user-id*s.
- *Raw socket:* A commodity OS allows only processes running with the set-user-id flag `suid = 0` to open a raw socket. The purpose of this design is to prevent users from applying raw socket to generate arbitrary malicious packets. A program that needs a raw socket but to be run by a non-privileged user should set the set-user-id flag correctly on its binary file. After IP addresses being isolated among *user-id*s, we do not have to prevent a regular user from using a raw socket any more. One who attempts to abuse a raw socket to forge a packet can only stick one's own IP address in the packet, since our system prohibits the usage of the other IP addresses. Enabling a user's program to open a raw socket doesn't need any further modification in the kernel. Once a user writes such a program, the privileged user can set the set-user-id flag of the executable correctly.

3.3 Routing Isolation

Isolating routing is extremely important for network experiments on virtual networks [9]. Although we mention in the discussed above that crafting PBR rules with user-specific address can achieve routing isolation, running separate routing protocols in different slices will not achieve this goal. The problem is that in a commodity OS only the privileged user can install a route to the system FIB. For this reason, we need to enable a regular user to update the FIB of the PBR associated with his or her *user-id* in the following steps.

First, to enable the route installation from a regular *user-id*, it is necessary to ask the kernel to accept the route update request from a the (non-privileged) *user-id* through either `ioctl` commands or `RTNETLINK` socket messages.

Second, once a regular *user-id* is allowed with route updates, it must be ensured that a certain user may not interfere with the routes of the others. Note that the privileged user can manipulate any tables.

When a table-id is specified in a route update request, the kernel must ensure that the table-id is associated with the *user-id*. When the table-id is not specified, then the kernel should find out which table is available for the request sender.

Therefore, to enable the comprehensive routing isolation, we have to add the route table association into the kernel as well.

When any route table-id is not explicitly associated with a certain *user-id*, it means this user wouldn't like to change the route by itself and it doesn't need a separate route table at all. Accordingly it just shares (and only reads) the default route table of the host, which is able to be update only by `root`. If a non-privileged user, associated with the default table, sends a route update request to the kernel, the kernel should reject the request and return the error, "operation not permitted".

As a summary of the system design, in Table 1, we list the features of isolation enabled (marked with "$\surd$") and not enabled (marked with "–") by our solution, compared with the container-based NetNS. We trade off only the privacy of slices for gaining the simplicity in coding, maintenance and deployment.

Table 1. Network Namespace Isolation Supports

Isolation	NetNS	Our Solution
Interface	$\surd$	$\surd$
Address/port	$\surd$	$\surd$
Forwarding	$\surd$	$\surd$ (w/ PBR)
Routing	$\surd$	$\surd$ (w/ PBR)
Traffic Reshaping	$\surd$	$\surd$ (w/ `tc`)
Raw socket	$\surd$	$\surd$ (authorized by `root`)
Visibility/privacy	$\surd$	–

We have implemented the Linux kernel patches and tools for the user-id-based network namespace isolation, with both IPv4 and IPv6. Readers are welcome to try our code which is open-source and downloadable from online[3].

The change involves three major parts: 1) new data structures associating IP address, *user-id* and route *table-id*, and their manipulation functions, which are used in new `ioctl` command as well as `procfs` parsers for the administrator doing management over the user-address-table associations; 2) modifications in socket system call instances, augmented with the *user-id* related behaviors; and 3) modifications in RTNETLINK message processing functions, enabling regular users to update route tables and dispatching the updates into proper table corresponding to *user-id*.

[3] See http://sourceforge.net/projects/uoa/ for details.

4 Demonstration: A VINI Experiment

Our proposed user-id-based namespace isolation implemented in the Linux kernel is minimal compared to the other container approach and can be easily apply to any version of Linux kernel release. This is mainly because the modified pieces are concentrated in the networking stack implementation of the Linux kernel. In this section, we demonstrate that our proposed solution can serve as a slice computing platform with network namespace isolated through achieving a similar platform to VINI [9], which also aims to provide network namespace isolation for a slice but using Linux VServer, a resource container virtualization technique.

We arrange the demonstration with a simple configuration. As is shown in Fig. 1, three computers running Linux with the modified kernel and physically connected to the same network. In the implementation of VINI, nodes are connected over the Internet, not necessarily within the same network. We separate the virtual links via tunneling, to mimic the real environment.

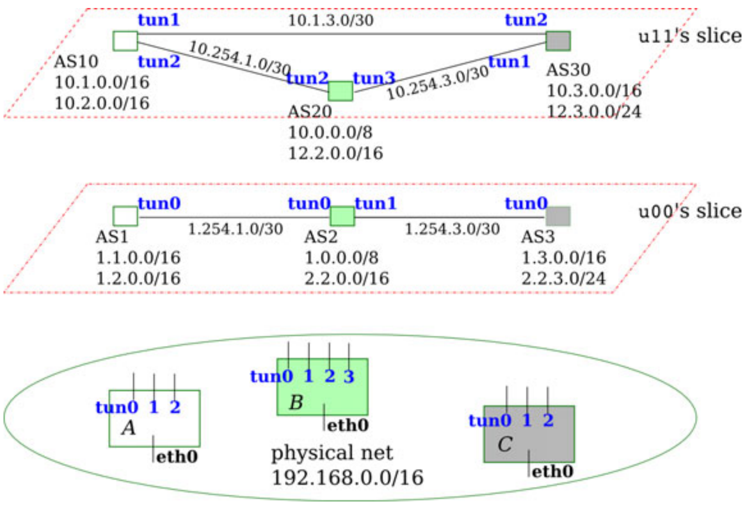

Fig. 1. Nodes, networks and slices for the VINI experiment

Two users are running their own slices with *user-ids*, `u00` and `u11`, respectively[4] . Assigning proper addresses of tunnel interfaces to the users, we have the topologies of the networks of the two slices separately defined. Fig. 1 depicts that the networks on the two slices are configured with separated address blocks.

[4] To be precise, *user-ids* of the same slice on different nodes could be vary. To simplify the discussion, we specify the *user names* on all nodes identical to the name of the slice and don't discuss the detailed number of *user-id* in the operating system.

4.1 Configurations

We suppose that address blocks 1.0.0.0/8 and 2.0.0.0/8 are assigned to `u00` while blocks 10.0.0.0/8 and 12.0.0.0/8 to `u11`. The two users would run their own networks with their own addresses, respectively. How do they arrange the addressing and networking depends upon their needs.

Addresses and route tables: We assume that after a user has made the layout of its in-slice network, the user informs the administrator of the platform to configure the virtual interfaces, enabling virtual links. Here we apply the GRE tunneling for this purpose and the `ip tunnel/addr/link` tools are utilized.

The administrative tool, `uoamap`, is used to configure the association of IP address, *user-id* and route table-id. Table 2 presents the configurations that we make for the nodes A, B and C, respectively.

Table 2. Address and route table associations

Config.	Node	u00	u11
IP addr. (iface)	A	1.254.1.2 (tun0)	10.254.1.2 (tun2)
			10.1.3.1 (tun1)
	B	1.254.1.1 (tun0)	10.254.1.1 (tun2)
		1.254.3.1 (tun1)	10.254.3.1 (tun3)
	C	1.254.3.2 (tun0)	10.254.3.2 (tun1)
			10.1.3.2 (tun2)
table-id	A B C	32800	32811
(rules)		(from 1.0.0.0/8, from 2.0.0.0/8)	(from 10.0.0.0/8, from 12.0.0.0/8)

Routing policies: data plane Rules for routing policies are configured directly with the `ip rule` tool. According to the address block assignment, we apply the policy that allow each slice's network can transfer packets with the source addresses within the blocks assigned to the slice. For our demonstration, we have shown these rules in the Table 2.

Note that the VINI also supports interaction of real networks and in-slice overlays. In such a case, source addresses of packets running over one's slice may not be restricted in the assigned address blocks. This can be supported with applying rules regarding the incoming interface, e.g., `ip rule add dev tun0 table 32800` on node A indicates that the packets coming through the device `tun0`, which is configured with an address in `u00`'s slice, can traverse the node A according to the route table with id 32800.

Links, addresses, tables and rules are all configured by the `root` on each node. Afterwards, users have the freedom of manipulate their own route table by either the manual configuration with `ip route` commands or the automatic update with routing protocol daemons.

Routing protocols: control plane In our demonstration, a routing protocol runs as an application of regular *user-id*. Unlike the other container approaches, in our solution, there are several minor limitations such as Quagga routing suite has to be re-compiled by each user, because the installation path must be set to the user's home directory rather than the default `/usr/local`, and the daemons must run with the context of the *user-id* itself rather than the default "`zebra`" or "`daemon`". In addition, we must disable Quagga's check on the security capability. All of these can be specified with the `configure` tool in the Quagga source.

After the Quagga suite is installed, a user has to edit the configuration files, `zebra.conf` and `bgpd.conf` to define BGP peers. Finally, the user can start both `zebra` and `bgpd` to start routing in its own slice. Since the BGP peering in our demonstration is simple so for brevity's sake, it is not shown here.

To meet the design goal of transparency to slice users, the table-id of a slice is known only to the `root` while it is hidden to a regular user. The latter only sees it as if it were the main table of the system and cannot see other users' tables. Therefore, it is not necessary to assign table-id in the `zebra` configuration.

4.2 RIB and FIB

BGP sessions are established between two application-layer peers, therefore their RIB is naturally isolated. Our demonstration shows that the sessions are established and not interfere each other. For example, for node A, we check both the BGP session advertisement as well as the corresponding route table to verify our design and implementation.

BGP tables: Each slice user applies `telnet` to the Quagga BGPd user interface of any of its own sliver. On node A, `u00`'s BGP table can be accessed through 1.254.1.2 port 2605, while `u11`'s BGP session can be completely isolated of that of `u10`.

Fig. 3 illustrates what we can achieve. The success of the BGP peer establishment verifies our PBR-based source address selection is working correctly.

Isolated FIB installation: The data plane of in-slice routing is isolated through the PBR facility. Our design further realizes the isolation of route updates from RIB to the user-specific FIB. We login as each user and run the `ip route` command to check the installation of the routes.

In displaying the routes, we don't specify the table-id and let the kernel pick up the user-specific table-id automatically. Therefore, the separated route display is also transparent to slices since it is not required that the slice user get to know the detailed number of its table-id, which is not decided by itself but decided by the local administrator (the `root`).

The successful establishment of BGP peers and the updates from BGP RIB to kernel FIB validates that our solution has the ability to support comprehensive routing isolation.

```
bgpdA> show ip bgp
BGP table version is 0, local router ID is 1.254.1.2

   Network          Next Hop    Metric LocPrf Weight Path
*> 1.0.0.0          1.254.1.1        0             0 2 i
*> 1.1.0.0/16       0.0.0.0          0         32768 i
*> 1.2.0.0/16       0.0.0.0          0         32768 i
*> 1.3.0.0/16       1.254.1.1                      0 2 3 i
*> 2.2.0.0/16       1.254.1.1        0             0 2 i
*> 2.2.3.0/24       1.254.1.1                      0 2 3 i

Total number of prefixes 6
```

(a) The RIB of `u00`'s slice at node *A*

```
bgpA> show ip bgp
BGP table version is 0, local router ID is 10.254.1.2

   Network          Next Hop    Metric LocPrf Weight Path
*  10.0.0.0         10.1.3.2                       0 30 20 i
*>                  10.254.1.1       0             0 20 i
*> 10.1.0.0/16      0.0.0.0          0         32768 i
*> 10.2.0.0/16      0.0.0.0          0         32768 i
*  10.3.0.0/16      10.254.1.1                     0 20 30 i
*>                  10.1.3.2         0             0 30 i
*  12.2.0.0/16      10.1.3.2                       0 30 20 i
*>                  10.254.1.1       0             0 20 i
*  12.3.0.0/24      10.254.1.1                     0 20 30 i
*>                  10.1.3.2         0             0 30 i

Total number of prefixes 6
```

(b) The RIB of `u11`'s slice at node *A*

Fig. 2. Success of in-slice BGP peering

5 Performance Evaluation

This section describes our qualitative and quantitative evaluations on the the proposed solution, showing it is light in both engineering and performance overhead.

5.1 Qualitative Evaluation

The development needs only a small amount of coding for the kernel patch and the management toolset. Totally, the kernel patch includes adding 10 and modifying 3 header files in `include/net`, adding 8 and modifying 20 C source files in `net/ipv4` and `net/ipv6` subdirectories. The kernel patch (`diff` result

```
u00@uor: ~$ /sbin/ip route list
1.254.1.1 via 1.254.1.2 dev tun0
2.2.3.0/24 via 1.254.1.1 dev tun0  proto zebra
2.2.0.0/16 via 1.254.1.1 dev tun0  proto zebra
1.3.0.0/16 via 1.254.1.1 dev tun0  proto zebra
1.0.0.0/8 via 1.254.1.1 dev tun0  proto zebra
```

(a) The FIB of `u00`'s slice at node A

```
u11@uor: ~$ /sbin/ip route list
10.1.3.2 via 10.1.3.1 dev tun1
10.254.1.1 via 10.254.1.2 dev tun2
12.3.0.0/24 via 10.1.3.2 dev tun1  proto zebra
10.3.0.0/16 via 10.1.3.2 dev tun1  proto zebra
12.2.0.0/16 via 10.254.1.1 dev tun2  proto zebra
10.0.0.0/8 via 10.254.1.1 dev tun2  proto zebra
```

(b) The FIB of `u11`'s slice at node A

Fig. 3. Success of in-slice route installation from RIB to FIB

from stock to our kernel code) has only 3707 lines, 99KBytes, including inline comments. The code involves only the networking kernel and is well decoupled from other parts, and therefore it is easy to maintain and upgrade. In contrast, either VServer or NetNS needs a big amount of coding, and the coding heaviness is also the major reason causing the poor compatibility, resulting in troubles that we have mentioned in Section 2.

In the deployment perspective, users of a node can directly use the proposed solution as long as the `root` has configured its available address, virtual links (if necessary) and routing table. Section 4 has demonstrated the simplicity of our user-specific addressing and routing without any containers. In VServer solution, it is necessary to establish containers and install frequently-used applications in the containers one by one. In NetNS solution, the root needs to detach the default namespace from a new shell process and then assign a virtual interface to it and establish the new namespace. It is obvious that the workload to introduce our user-specific addressing-routing approach is lower than them two.

Therefore, we are confident of the lightness of our solution in the terms of the engineering for code maintenance and deployment. We need also verify the performance scalability of the proposed solution. We conducted the following benchmark with our prototype implementation and compare it to the performance of the stock kernel. In [10], one can find benchmark results for the container-based solutions, where either VServer or NetNS is significantly poorer than the stock kernel.

5.2 Benchmarking

For the benchmark, two computers equipped with Intel Core2Quad CPU Q6700@ 2.66GHz and 3.2GB memory are connected over a 1000 Mbps cross-wire Ethernet link. One of them, (A), is installed the native Linux kernel 2.6.32 for Debian 5.0.3 and our patched kernel. It can be booted up with either. The other machine, (B), is running with the native kernel. The benchmarking tool *netperf* [4] version 2.4.4 is used. *netserver* is running at (B) and (A) starts `netperf`.

Working with the kernel enabled with user-specific address and route table, (A) is running with a certain number of IPv6 addresses, one-to-one associated to the same number of *user-id*s. For a fair comparison, when the machine (A) runs the stock kernel, i.e., users are not associated with specific addresses but share them all, we also *configure* the same number of addresses to the interface, even though processes always pick up the same one. Further, we run each test twice, one round with source address unspecified while the other round with source address specified. All the tests are conducted with source port unspecified at the sender side, i.e., the kernel will perform port selection for connection or message sending.

The *netperf* benchmarking is conducted with four types of test: TCP connection/close, UDP request/response, TCP streaming and UDP streaming. Tests are run by one user among a group, every user of which is assigned with a dedicated IPv6 address. Performance impact of the user group size is depicted in Fig. 4.

TCP streaming and UDP request/response are the most popular types of applications applying TCP and UDP, respectively. Our benchmark shows that the performance of TCP streaming and UDP request/response are not impacted by the patch for the user-specific addressing and routing. Actually, for the TCP streaming, either the address availability check or the port conflict detection is done only once for a session at the connection establishment stage. The check for the address availability must consume some CPU time but this could be ignored in the macro observation for the end-to-end throughput. For UDP request/response, the check and detection is done for every request, and the address availability check consumes also the ignorable amount of CPU time.

The TCP connection/close and UDP streaming benchmarking, however, present an unexpected behavior, where the patched kernel performs even better than the stock kernel when the user group is getting quite large. This can be explained with the principle of port selection. Because the user-specific addressing decreases the probability of the port conflict, when the number of concurrent sessions gets high, stock kernel suffers higher port conflict rate and it must take more time to find an unoccupied port number for a new socket. TCP connection/close leaves a lot of sockets in the TCP TIME_WAIT state and they hold the use of their port for a quite long period in comparison to the action of connection establishment, while UDP streaming also causes a lot of long-living sockets occupying port numbers in use. Therefore, the affect of port conflict is apparent in these two types of benchmark.

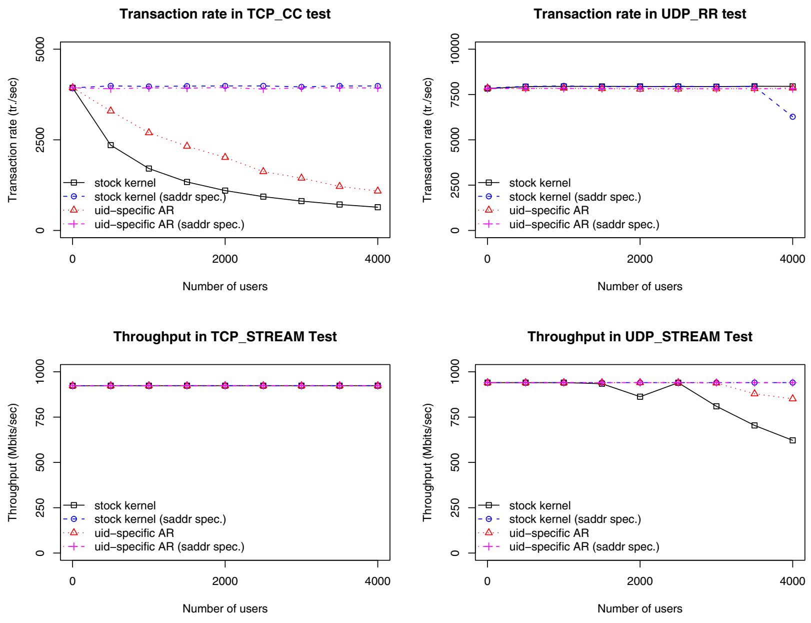

Fig. 4. Performance comparison: user-specific addressing vs. stock kernel

To summarize the benchmark, we state that even the current full-fledged prototype of our user-specific addressing and routing can achieve the end-to-end performance not worse (but sometimes better) than the commodity OS, for most cases of Internet applications.

The lightness in run-time overhead makes our user-specific addressing-routing approach competing with a container based solution like VServer. Without container, our solution can easily support up to 4K slices (actually, 4K is the maximum number of IP addresses that are able to be configured with a network interface card in Linux) over a PC and have good performance. While a 10^2 GB-capacity disk is hard to support more than hundreds of VServer guests because each guest has to consume at least tens of MB for the basic container facility.

What we trade off for the scalability is the `root` environment for a slice. On the other hand, in the privacy-sensitive circumstances, we cannot shield a slice's activity from being viewed by another.

6 Conclusions

This paper explores the feasibility of supporting isolation for slices in a commodity OS with as few modifications as possible and only within the networking kernel. For this purpose, we design and implement the idea of isolating IP addresses and routing tables among *user-id*s on top of a commodity OS. Our minimal modifications to the stock kernel together with the well-developed policy-based routing (PBR) facility successfully validate that the philosophy of minimal engineering can provide comprehensive isolation for network namespace. Meanwhile, the performance overhead is light enough to ensure the performance of the modified kernel not worse than the commodity OS stock kernel. The number of slices supported by the user-specific addressing and routing can reach the limit of the maximum IP addresses that a commodity OS can configure with a network interface.

Acknowledgment

We thank Taoyu Li for his early design and implementation, and Yuncheng Zhu for the discussion and help in performance evaluation.

References

1. Freebsd architecture handbook, `http://www.freebsd.org/doc/en/books/archhandbook/jail.html`
2. Linux network namespace (NetNS), `http://lxc.sourceforge.net/network.php`
3. Linux VServer, `http://www.linux-vserver.org/`
4. Netperf, `http://www.netperf.org/`
5. OpenVZ, `http://wiki.openvz.org/Main_Page`
6. VMWare, `http://www.vmware.com/`
7. Xen, `http://www.xen.org/`
8. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP 2003: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 164–177. ACM, New York (2003)
9. Bavier, A., Feamster, N., Huang, M., Peterson, L., Rexford, J.: In vini veritas: realistic and controlled network experimentation. SIGCOMM Comput. Commun. Rev. 36(4), 3–14 (2006)
10. Bhatia, S., Motiwala, M., Mühlbauer, W., Mundada, Y., Valancius, V., Bavier, A., Feamster, N., Peterson, L., Rexford, J.: Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In: ACM ROADS Workshop 2008, Madrid, Spain (December 2008)
11. Chen, M., Nakao, A., Bonaventure, O., Li, T.: UOA: Useroriented addressing for slice computing. In: Proceedings of ITC Specialist Seminar on Network Virtualization, Hoi An, Vietnam (May 2009)
12. Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., Lepreau, J.: Large-scale virtualization in the emulab network testbed. In: ATC 2008: USENIX, Annual Technical Conference, pp. 113–128. USENIX Association, Berkeley (2008)
13. Peterson, L., Anderson, T., Culler, D., Roscoe, T.: A Blueprint for Introducing Disruptive Technology into the Internet. In: Proceedings of the 1st Workshop on Hot Topics in Networks (HotNetsI), Princeton, New Jersey (October 2002)
14. Peterson, L., Muir, S., Roscoe, T., Klingaman, A.: PlanetLab Architecture: An Overview. Technical Report PDN–06–031, PlanetLab Consortium (May 2006)