IPSJ SIG Technical Report

Vol.2009-ARC-183 No.20
Vol.2009-OS-111 No.20
2009/4/23

Light-weight Slices Employing Linux Containers To Support Flexibility of CoreLab

Ryota Ozaki[†1] and Akihiro Nakao [†2,†1]

Network testbeds are evolving with recent virtualization technology such as virtual machine monitors and resource containers. Isolating resources with such virtualization technology, a network testbed provides a *slice*, a set of resources allocated across the Internet to harness execution of experimental network services. We have been developing CoreLab that aims to enhances PlanetLab, one of the most popular planetary-scale network testbeds, which is designed to provide performance, isolation, scalability of sliced execution environments. CoreLab considers two additional design principles such as flexibility and code-re-usability. It currently leverages a hosted virtual machine monitor to allow an arbitrary operating system in a slice, in such a way that the host environment can keep up with the latest devices and processors, unlike in PlanetLab that relies on the resource container called Linux-VServer that requires patches to the non-latest host kernel. In this paper, we explore the space for another kind of resource container approach that follows our design principles in CoreLab development. We posit that supporting various kinds of virtualization technologies could satisfy arbitrary requirements for experiments, ranging from high performance to flexible execution environments. The paper reports preliminary evaluation and investigation as to whether Linux Containers satisfy our design principles.

1. Introduction

Recent rapid progress in virtualization technology allows us to run each instance of operating system in an *isolated* environment without interference from the other. Isolation has become a key feature in implementing an emerging network test-bed for developing, deploying and experimenting with a new and possibly *disruptive* network services.

We are developing CoreLab[19),20)], an emerging network test-bed that enhances PlanetLab (PL)[7)], the most popular and successful network test-bed to date. PL is designed to provide performance, isolation, scalability of execution environments, and uses the resource container called Linux-VServer (VS)[16)] to harness multiple concurrent experiments. On the other hand, CoreLab considers two additional design principles such as flexibility and code-re-usability. It currently leverages the hosted virtual machine monitor (VMM) called Kernel-based Virtual Machine (KVM)[10)] to accommodate an arbitrary operating system in a *slice*, a set of resources allocated across the Internet to harness execution of experimental network services. We leverage KVM and the other supporting software so that we can keep up with the latest devices and processors, unlike in PlanetLab where the host kernel is limited to the version 2.6.22 due to a large amount of patches to enable VS.

However, virtual machine (VM) techniques usually sacrifice performance and scalability of the execution environments over the flexibility. For example, VM techniques such as KVM consumes more memory than container approaches and can support less number of slices for a given hardware. Although there exist several attempts to address this disadvantage[8),32)], it is generally difficult to overtake containers in this aspect. We address this issue by supporting a variety of virtualization technologies simultaneously, from containers to hosted virtual machines, to compensate for drawbacks of one another.

This paper reports our consideration of adding a container to CoreLab. We investigate Linux Containers[15)], shortly LXC, an implementation of containers in Linux, that leverages only built-in functions of the Linux kernel and does not require any patches to the kernel. Using LXC instead of VS may allow CoreLab to use the latest kernel and support for new devices and processor capabilities, thus satisfies our design principle of code-reusability. The paper also reports preliminary evaluation of network performance and resource consumption in our prototype implementation with LXC, and investigates whether Linux Containers satisfies our design principles except for flexibility.

The rest of this paper is organized as follows. Section 2 describes an overview of CoreLab. Section 3 describes motivations of this research. Section 4 describes common overview of container techniques and Section 5 elaborates on Linux Containers. Section 6 describes our implementation details. Section 7 reports results of primary evaluations. Section 8 and Section 9 discuss further improvements of LXC and related work, respectively. Section 10 briefly concludes the paper.

2. CoreLab

CoreLab[19),20)] is an emerging network testbed enhanced from PL and deployed on

†1 New Generation Network Research Center, National Institute of Information and Communications Technology
†2 Interfaculty Initiative in Information Studies, Graduate School of Interdisciplinary Information Studies, The University of Tokyo

JGN2Plus[13] and SINET networks. Although CoreLab inherits code base from PL[29], it is designed to overcome well-known limitations in PL such as inflexibility in execution environments[19),20].

CoreLab leverages KVM, a hosted VMM (Type-II) utilizing hardware assist, such as Intel VT-x[31] and AMD-V[1] and a user space software piece called QEMU[3] that emulates device I/O operations and provides user interface. It also leverages para-virtualization to boost I/O operations and gains I/O performance[24),25].

As briefly noted in Section 1, following our design principles we employ a combination of established and thus well-debugged pieces of software as well as hardware accelerations, e.g., KVM, MyPLC, libvirt[28], Bittorrent[4], VNC (embedded within QEMU[3]), nagios[18], and Linux networking tools and kernel extensions like netfilter and iptables.

3. Motivations

In this paper, we primarily focus on incorporating a container approach into CoreLab as one of the various virtualization techniques we intend to support in future. We have three motivations to do that.

First, it is not easy to predict demands for testbed features from developers of network services. Although hosted VMMs may provide flexible development environments than containers, containers could achieve high performance sacrificing flexibility. Considering that we do not have one-size-fit-all type of virtualization technology yet at this point, it is best to prepare various kinds of virtualization techniques simultaneously.

Second, it is good to support a variety of hardware platform to run our node software to enable execution environments. One of the most important features of planetary-scale test-beds is to support heterogeneity of platforms.

Finally but most importantly, we must follow one of our design principle of code-reusability. We must keep up with the latest technology for the host environment to fulfill our motivations listed above.

Keeping these in our mind, we decide to explore the space for another container approach called LXC as one candidate for a multitude of virtualization techniques we intend to support in CoreLab.

Table 1 Classes of resources and their examples.

	Sharable	Non-sharable
Separable	files (RO or COW)	files, PID, IPC
Non-separable	CPU, memory	kernel core

4. Containers

4.1 Overview

Containers is known as a kind of virtualization techniques. Like VM techniques, it allows multiple OS instances to run on a physical machine simultaneously. Its ability is fairly similar to VM techniques, however, the internal is very different from VMs'.

A VM technique creates an illusion of a physical (sometimes pseudo) machine to served OSes and by doing so multiple OSes can run on a single machine. On the other hand, a containers technique just divides its OS resources into multiple sub-instances. In fact, the only one OS kernel is running on the top of a physical machine at given time. Nonetheless, from a view of applications or users, they expect that they own their OS and machine resources such as CPU, memory, disk, and even network resources.

Benefit in using containers instead of VMs is a combination of performance and low resource consumption, while drawback is inflexibility of execution environment, since they must inherit the single host environment. Application processes inside a container are almost the same as ordinary processes on an ordinary OS, thus, there is few overhead of virtualization even the case of I/O operations. Furthermore, container techniques can share resources more easily than VM techniques. For example, traditional memory sharing techniques that are usually used in shared libraries and processes are able to be leveraged among containers.

However, container techniques have several limitations due to that running OSes actually shares the same kernel image. Also inside the container, users are usually not allowed to modify parameters of the kernel, insert additional kernel modules, setup network filtering rules and routing tables[⋆1], etc.

4.2 Resource isolation and protection

This subsection describes how OS resources are isolated in containers. Table 1 classifies resources in an OS with the points of sharable and separable among containers.

Most resources are named and structured by the kernel, i.e., the resources are separable. The resources are relatively easy to be isolated by separating namespace of them. For example, a filesystem can be separated by its sub-directory. It is usually conducted by *chroot* system call in ordinary Unix-like OSes. For another example, process ID namespace of each container is separated so that the *init* process of each container can have process ID 1.

Several separable resources are able to be also sharable. Some files can be shared among containers, for example system binaries (e.g., files under /bin, /sbin) and libraries (e.g., files under /lib, /lib64) using read-only bind mount or a whole filesystem using COW capability of a filesystem.

On the other hand, in the case of non-separable resources, we need a mechanism to share them or protect from containers. Several resources are shared through time or space division. The resources should be given to each container fairly through accounting resource consumptions of each container preciously. For example, CPU consumption is accounted by a set of processes in a container not by a process.

Remaining resources are non-separable and shared among containers, thus, they should not be accessed by containers. For example, kernel parameters and kernel modules inside the kernel should be protected from containers. Even an administrative user in a container should not be allowed to access the protected resources inside the kernel.

5. Linux Containers

5.1 Overview

Linux Containers, shortly LXC[15)], is an implementation of container in Linux. Unlike the other containers, VS and OpenVZ[23)], LXC leverages the capabilities of vanilla Linux instead of additional code that requires patches to the kernel. The advantage of the approach is that it can leverage capabilities of latest kernels, e.g., new features of the kernel, applications and libraries for the features, and device drivers for new devices.

⋆1 It depends on a setting of containers. OpenVZ allows it with a normal setting, however, PlanetLab does not allow.

LXC can be divided into user space tools and kernel components. There are two user space tools, lxc[15)] and libvirt[28)]. Both tools enables to manage containers with formal configuration files, command-line tools, or a special shell. The capability inside the kernel for LXC is implemented with the combination use of several components.

5.2 Components of LXC

LXC utilizes capabilities of namespace separation, control groups (cgroups)[14)], and a security module in the Linux kernel.

Namespace separation: The latest Linux, version 2.6.29 at this time, can separate the following resources.

- hostname
- PID
- IPC
- User (UID/GID)
- Network
- pseudo filesystems (procfs, sysfs, and devpts)
- filesystems (mount points)

Since network resources can be separated, each container has individual network resources, such as network interfaces, routing tables, etc.

Cgroups: It is a general facility to control resources in Linux. It allows the administrator to group processes with hierarchical structures and control several resources assignments, allocations, and limitations according to the grouped processes. Device files, a set of CPUs, and amounts of memory are controlled using cgroups.

The above two are must and actually used in both lxc and libvirt, however, there remains some critical resources faced on unexpected accesses by containers as mentioned in Section 4.2. Therefore, we need to protect the critical resources in the kernel with somewhat additional facilities. There are several candidates of security modules to address the problem, for example *POSIX capabilities*[11)], *SELinux*[26)], *SMACK*[6)], etc.

6. Implementation

In this section, we describe a candidate implementation of LXC-based slices.

6.1 LXC management

We use libvirt to manage LXCs as well as KVM that CoreLab currently supports. Libvirt provides a common API set to manage several virtualization techniques such as

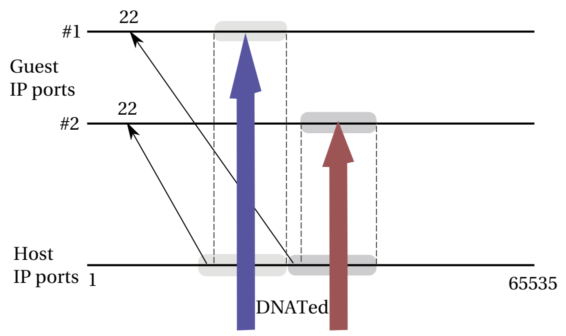

Fig. 1 IP ports assignments.

KVM, LXC, Xen, and OpenVZ. As a result we could mostly reuse the code base for KVM.

6.2 Network configuration

A container of LXC is connected to the host with *veth (Virtual ETHernet)* that is a pair of ethernet devices connected each other in point-to-point where each ethernet device has MAC and IP addresses. We configure both with private IP addresses and make NAT on the host to allow the container to access outside networks. On the other hand, the case of accesses from outside, we assign a range of IP ports to each container and forward packets within the range to the corresponding container using DNAT[20] where the lowest port is reserved to ssh (Fig. 1).

In this configuration, an individual ssh server run in each container where PL runs a ssh server that is modified to redirect a login user for the corresponding container with a trick in the ssh server and the login shell.

6.3 Base Filesystems

To make a container an individual OS instance, the container needs to have its own filesystem that includes binaries, libraries, and configuration files. Although other container implementations have their own tool to build a filesystem for a container, LXC does not have such a tool yet.

Instead of using a special tool, we reuse VM disk images for KVM-based slices. We disseminate VM disk images using BitTorrent[4] to CoreLab nodes[20]. We extract contained files from a downloaded VM disk image and use the files as a *base filesystem*.

As mentioned above, each container requires an individual filesystem, however, copying a whole base filesystem for every containers is clearly inefficient. We leverage the COW feature of *btrfs*[5]. Btrfs can have sub-volumes in a btrfs volume and take a snapshot of a sub-volume at any time. The snapshot can share data with the original sub-volume in the disk until data is modified. The modified data is duplicated and stored as another data in the disk. Assigning each container to a snapshot inherited from a sub-volume that contains a base filesystem can reduce disk usage (Fig. 2).

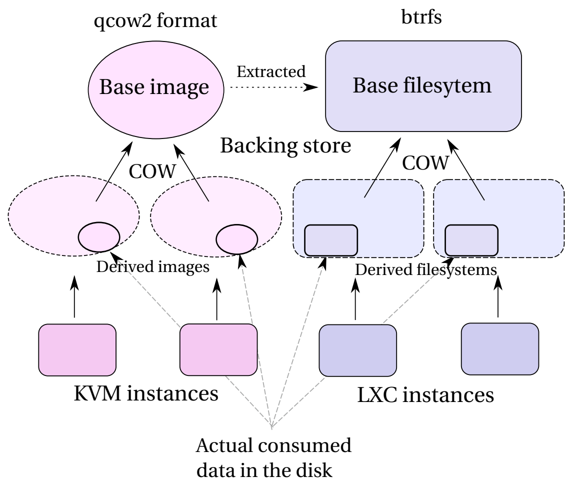

Fig. 2 A base filesystem and derived filesystems.

6.4 Privileges of containers

We currently prevent accesses of containers to the critical resources in the kernel using *POSIX capabilities*. It is a kind of functionalities to realize *least privilege*[9] for administrative users and processes. It divides the root privilege into small pieces and allows to assign a set of them (or all of them, i.e., the case of ordinary root user) to a process. The privilege assigned to a process is inherited to descendants of the process[⋆1].

We drop several capabilities of the init process in every containers before its ex-

⋆1 To be precise, the privilege is decided with capabilities assigned to a process *and* the execution file of the process. However, if no capability is assigned to any files in a container and CAP_SETFCAP is dropped in the init process in the container, it is guaranteed that any processes in the container never gain dropped capabilities again.

ecution, for example CAP_SYS_BOOT, CAP_SYS_ADMIN, CAP_SYS_MODULE, CAP_MKNOD are dropped for safe.

6.5 Miscellaneous

An ordinary OS starts the *init* process as the first user process and then the init initializes environments and runs system services. However, the init is overkill in the case of a container. For example, mounting pseudo filesystems and creating device files that are usually responsible for the init to do are conducted by libvirt. Instead of using the init, we use a simple shell script that just starts several services such as dhclient, rsyslogd, and sshd and then transforms a shell to serve accesses through a console from the host, which is actually not used in usual operations except for debugging. They are minimal and enough to make a container usable in CoreLab.

We need a trick to run 32-bit programs in a container of LXC on a machine running a 64-bit kernel, even the kernel enables 32-bit emulation mode. Because Linux cannot fake the architecture of processors, i.e., `uname` syscall returns the architecture of the kernel straightforward. Although most 32-bit programs can run without knowing about the underlying architecture, some 32-bit programs are confused, for example `yum`, a package management tool, that depends on the return value of uname syscall. We avoid this defect by using a linker technique. Setting a library that define a function named uname in LD_PRELOAD, the function is called ahead over the original function in libc. Replacing 'x86_64' with 'i386' in the return value of original uname by using the technique can cheat on yum as the underlying architecture is 32-bit. Nonetheless, we think the capability to fake the underlying architecture is needed in the kernel primitives to make LXC more useful.

7. Evaluations

This section evaluates memory consumption and network performance both in KVM and LXC. We set up a set of two nodes in private CoreLab deployed in a closed LAN. Each node has one 2.66 GHz dual core processor (Intel Xeon x3070) and 4 GBytes memory and connects over Gigabit Ethernet network with each other. We use linux-2.6.29-rc8 as the kernel with kvm-83 loaded. Note that VMs in CoreLab are equipped with a para-virtualized driver, called virtio[25].

7.1 Resource consumption

We compare memory and disk consumption on LXC and KVM. We increase the number of containers/VMs ranging from 0 to 200 by 5 and estimate increasing amount of disk usage and ratio of used memory and swapped memory to the available memory.

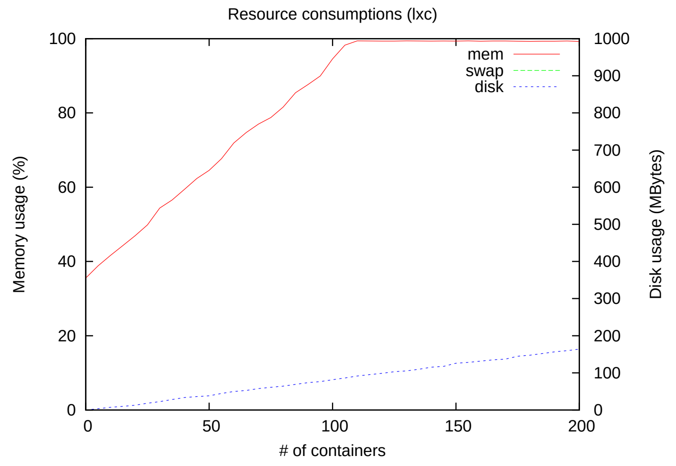

Fig. 3 Resource consumption on LXC.

Figure 3 and 4 show the results on LXC and KVM, respectively. In both graphs, the x axis indicates the number of containers/VMs, the left y axis indicates percentages of memory usages, and the right y axis indicates increasing amounts of disk usage.

Although not surprising, the result shows that resource consumption of LXC is lower than those of KVM. LXC leaves free memory until the number of containers increases up to 110 where KVM consumes entire memory up by the number of VMs reached 30. Furthermore, after consuming the entire memory, LXC does not cause swapping mostly where KVM causes it when 40 of VMs launches. This is thanks to the facility of memory sharing in the kernel and the COW feature of btrfs. The former enables to share the memories used by processes in every containers and the latter suppresses increasing amount of disk usage only meta data that are not shared in btrfs. As a result, LXC could scale up hundreds of containers. On the other hand, KVM ends up booting VMs anymore after swapped memory has reached the maximum size of the swap.

Note that the resource consumption in this evaluation is absolute minimum and running process inside containers/VMs will increase the consumptions.

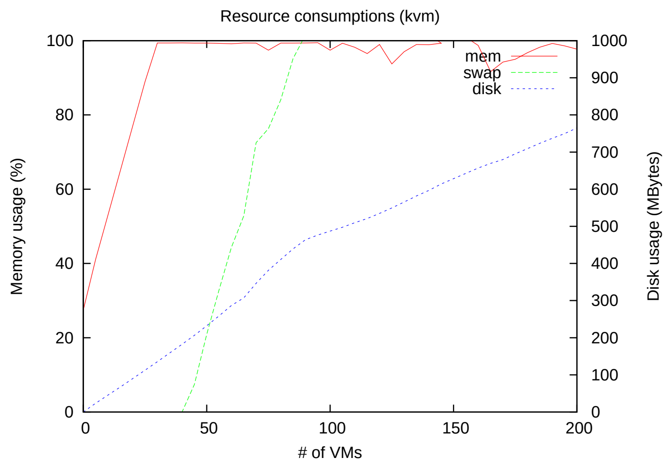

Fig. 4 Resource consumption on KVM.

Table 2 Throughput (Mbps)

	Average	Minimum	Maximum
Vanilla Linux	940	940	941
PlanetLab	941	941	942
LXC	931	901	934
KVM (virtio)	604	492	702

7.2 Throughput

We compare the throughput on KVM and LXC as follows. We run `iperf` server and client in each VM and container. In this evaluation, we run iperf during seven minutes and do not account the first three minutes to get rid of the effect of TCP slow-start. We also sample the throughput every 10 seconds to be used for estimating the minimum and the maximum in addition to the average.

Table 2 shows the comparison among vanilla Linux, PL, LXC and KVM in terms of TCP throughput between two nodes measured by iperf. The case for the vanilla Linux is added to show the baseline throughput. It is clear that PL and LXC gains the comparable throughput with vanilla Linux, while KVM achieves about 60 percent of the performance on average. The reason that the throughput of LXC is a bit lower than PL is difference in the network configuration. In PL, all container shares a physical network interface where LXC's setting assigns an individual network interface to each container and routes packets from/to corresponding containers. Thus, we currently believe that the overhead of routing is appeared in the difference between two results.

8. Discussion

We believe that LXC is a potential implementation of containers, however, it is not practically mature at this time and requires further improvement. The most important concern to incorporate LXC into CoreLab is security isolation. Unlike VMs offered by KVM, containers have potential issues in security isolation due to its architecture. We must not disclose a way to jail-break to users in any containers.

We summarize behaviors inside a container and functionalities of each container implementation in Appendix A.1. This investigation shows that LXC remains several holes to be fixed even though most of them are not critical defections.

Furthermore, there remains another concern about performance isolation. One concern is that LXC does not have quota supports of disk usage by a container in LXC itself. We need to cap disk space to protect malicious uses in a container. There are two possible solutions to this issue; one is to use Logical Volume Manager (LVM) and the other is to use btrfs that is actually not available at this time and will be available in near future. Using LVM as a storage pool and a fixed partition as a storage volume for a container, we can set the upper limit of a disk usage by a container. However, this method cannot leverage the COW capability. On the other hand, btrfs has a plan to implement quota per sub-volume/snapshot [⋆1]. We believe that this method will be one of most dominant solutions for the issue.

Yet another concern is that capability is missing to limit disk I/O bandwidth by a container. Linux already has a capability to control the bandwidth using *ionice*, however, it just sets a priority of I/O requests. Furthermore, we also need to cap network bandwidths as well.

9. Related Work

Linux-VServer (VS) is one of the earliest implementations of the container for Linux. It supports minimal capabilities for a container, such as namespace separations, pretty-

⋆1 `http://btrfs.wiki.kernel.org/index.php/Development_timeline`

Vol.2009-ARC-183 No.20
Vol.2009-OS-111 No.20
2009/4/23

enhanced POSIX capabilities, etc. It has been used in PlanetLab and thus demonstrated its capabilities and effectiveness.

OpenVZ is an implementation of containers newer than VS. It enhances virtual networking and resource controls than VS. Additionally, live migrations of containers are supported and the feature makes OpenVZ unique among container implementations.

Both implementations have not been merged in the mainline Linux kernel[⋆1], therefore, it requires patches for the kernel. This drawback is against our design principles of CoreLab. We must avoid the case where the latest development in the kernel may not be hindered by some underdeveloped features. For example, KVM enhances its memory swapping of guests by the kernel 2.6.27, however, VS and OpenVZ for that version are not available.

Xen is a hypervisor that serves multiple VM instances on a physical machine[2)]. It fully leverages para-virtualization techniques to boost performance, and gains comparable performance to native OSes. Although it might realize both flexibility and high performance, it is not keeping up with the latest host kernel but with only the version 2.6.18.

OpenSolaris natively supports containers called *Solaris Containers (SC)*[22)]. SC is superior than LXC on several capabilities at this time. SC has sophisticated management tools and a mature COW-capable filesystem called zfs. OpenSolaris also supports a hosted VMM, Sun xVM VirtualBox[27)]. In theory, OpenSolaris may become one of the candidates to support in CoreLab. However, our goal is to support a variety of virtualization techniques to achieve flexibility, not just a container approach. We must also consider compatibility of the management software to federate with the other PlanetLab based test-bed such as OneLab[21)], EverLab[12)] and G-Lab[30)]. That said, we may consider SC as a candidate virtualization technology to support in CoreLab in near future.

10. Conclusion

Network testbeds have evolved with recent virtualization technologies such as virtual machines and containers.

In this paper, we evaluate LXC as a candidate container approach to be supported in CoreLab. Although LXC still leaves room for improvements, it seems to satisfy our design principles in CoreLab and to let CoreLab keep up with the latest host kernel.

The contributions of this paper are two-fold. First, we have evaluated a container technology that follows our design principles in CoreLab development. If this approach becomes mature, we can support two type of virtualization techniques, KVM and LXC, thus increases the flexibility of execution environment. Second, we have implemented a prototype of container based slice using LXC and have conducted preliminary evaluations.

⋆1 Some capabilities in OpenVZ such as veth have been merged in the mainline kernel and LXC leverages them.

References

1) AMD: AMD64 Virtualization Codenamed "Pacifica" Technology, Secure Virtual Machine Architecture Reference (2005).
2) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, Proc. *the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp.164–177 (2003).
3) Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, Proc. *USENIX 2005 Annual Technical Conference, FREENIX Track*, pp.41–46 (2005).
4) BitTorrent, Inc.: BitTorrent. `http://www.bittorrent.com/`
5) Btrfs. `http://btrfs.wiki.kernel.org/`
6) Casey Schaufler of California: SMACK: The Simplified Mandatory Access Control Kernel for Linux. `http://schaufler-ca.com/`
7) Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M. and Bowman, M.: PlanetLab: an Overlay Testbed for Broad-coverage Services, *ACM SIGCOMM Computer Communication Review*, Vol.33, No.3, pp.3–12 (2003).
8) Diwaker, G., Sangmin, L., Michael, V., Stefan, S., Alex, S.C., George, V., Geoffrey, V.M. and Amin, V.: Difference Engine: Harnessing Memory Redundancy in Virtual Machines, Proc. of *8th USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, pp.309–322 (2008).
9) Fred, S.B.: Enforceable security policies, *ACM Transactions on Information and System Security (TISSEC)*, Vol.3, No.1, pp.30–50 (2000).
10) Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux Virtual Machine Monitor, Proc. *Ottawa Linux Symposium 2007 (OLS '07)*, pp.225–230 (2007).
11) Hallyn, S.E. and Morgan, A.G.: Linux Capabilities: making them work, Proc. *Ottawa Linux Symposium 2008 (OLS '08)*, pp.163–172 (2008).
12) Jaffe, E., Bickson, D. and Kirkpatrick, S.: Everlab-A Production Platform for Research in Network Experimentation and Computation, Proc. of *the 21st Conference on 21st Large Installation System Administration Conference*, pp.203–213 (2007).
13) JGN2plus. `http://www.jgn.nict.go.jp/`
14) libcg: Control Group Configuration Library. `http://libcg.sourceforge.net/`

15) Linux Containers. http://lxc.sourceforge.net/
16) Linux-VServer. http://linux-vserver.org/
17) Menon, A., Cox, A.L. and Zwaenepoel, W.: Optimizing Network Virtualization in Xen, Proc. *2006 USENIX Annual Technical Conference (USENIX '06)*, pp.15–28 (2006).
18) Nagios Enterprises, LLC.: Nagios: The Leader and Industry Standard in Enterprise System, Network, and Application Monitoring. http://www.nagios.org/
19) Nakao, A., Ozaki, R. and Nishida, Y.: Building a Flexible Overlay Network Testbed with a Hosted Virtual Machine Monitor, Proc. of *20th Computer System Symposium 2008 (ComSys '08)*, pp.13–22 (2008).
20) Nakao, A., Ozaki, R. and Nishida, Y.: CoreLab: An Emerging Network Testbed Employing Hosted Virtual Machine Monitor, Proc. of *ACM 3rd International Workshop on Real Overlays & Distributed Systems (ROADS '08)* (2008).
21) OneLab. http://www.one-lab.org/
22) Price, D. and Tucker, A.: Solaris Zones: Operating System Support for Consolidating Commercial Workloads, Proc. *18th Large Installation System Administration Conference (LISA '04)*, pp.241–254 (2004).
23) OpenVZ. http://openvz.org/
24) Ozaki, R. and Nakao, A.: Analysis of Network I/O Performance in KVM, Proc. of *IPSJ SIG Technical report*, 2008-OS-107, pp.111–118 (2008).
25) Rusty Russell: virtio: towards a de-facto standard for virtual I/O devices, ACM SIGOPS Operating Systems Review, Vol.42, No.5, pp.95–103 (2008).
26) Smalley, S., Vance, C. and Salamon, W., Implementing SELinux as a Linux security module, *NAI Labs Report*, Vol.43, No.1 (2001).
27) Sun Microsystems, Inc.: VirtualBox. http://www.virtualbox.org/
28) The Virtualization API. http://libvirt.org/
29) The Trustees of Princeton University: MyPLC. http://www.planet-lab.org/doc/myplc
30) Tran-Gia, P.:G-Lab: A Future Generation Internet Research Platform. http://www.future-internet.eu/
31) Uhlig, R., Neiger, G., Rodgers, D., Santoni, A.L., Martins, F.C.M., Anderson, A.V., Bennett, S.M., Kagi, A., Leung, F.H. and Smith, L.: Intel Virtualization Technology, *IEEE Computer*, Vol.38, No.5, pp.48–56 (2005).
32) Waldspurger, C.A.: Memory Resource Management in VMware ESX Server, Proc. of *the 5th Symposium on Operating Systems Design and Implementation*, pp.181–194 (2002).

Appendix

A.1 Detailed comparison among container implementations

(1) machine: The architecture type cannot be changed.
(2) syslog: LXC discloses dmesg.
(3) network: This indicates network traffics can be isolated.

Table 3 Detailed comparison among container implementations

LXC: Linux Containers
VZ : OpenVZ
VS : Linux-VServer

	LXC	VZ	VS	Notes
hostname	o	o	o	
machine	x	x	x	(1)
syslog	=	o	o	(2)
network	o	o	o	(3)
kernel version	=	=	=	(4)
disk size	=	o	o	(5)
quota	=	o	o	(6)
mount points	o	o	o	
memory	=	o	=	(7)
sysfs	=	o	–	(8)
procfs	x	o	=	(9)
devpts	o	o	o	(10)
tools	=	o	o	
freeze	o	o	x	(11)
Checkpoint/Restart	x	o	x	(12)
live migration	x	o	x	
PID	o	o	x	(13)

(4) kernel version: VZ and VS need kernel patches. LXC needs latest kernel.
(5) disk size: LXC discloses host disk sizes.
(6) quota: LXC depends on the underlying filesystem.
(7) memory: LXC and VS discloses the host memory size although it is capped.
(8) sysfs: LXC and VS discloses host's resources although they are protected.
(9) procfs: LXC discloses host information and some parameters are writable although we can make whole procfs read-only.
(10) devpts: LXC supports devpts separation from kernel 2.6.29
(11) freeze: This indicates whether be able to stop a container that includes all process inside the container.
(12) Checkpoint/Restart: The feature is being prepared and might be available in LXC in near future
(13) PID: VS is shared PID namespace among containers and the host