Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Evaluation of Linux Native Isolation Mechanisms for XtreemOS Flavours
D2.1.6

Due date of deliverable: November 30^{th}, 2008
Actual submission date: January 10^{th}, 2008

Start date of project: June 1^{st} 2006

Type: Deliverable
WP number: WP2.1
Task number: T2.1.10

Responsible institution: INRIA
Editor & and editor's address: Yvon Jégou
IRISA/INRIA
Campus de Beaulieu
35042 RENNES Cedex
France

Version 1.0 / Last edited by INRIA / Jan 9^{th}, 2009

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	$\surd$
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.10	22/10/08	Yvon Jégou	INRIA	Initial document structure
0.11	22/10/08	Yvon Jégou	INRIA	Requirement section
0.12	23/10/08	Jérôme Gallard	INRIA	OpenVZ subsection
0.13	27/10/08	Jérôme Gallard	INRIA	Virtualization (Type-I, Type-II) section
0.14	28/10/08	Yvon Jégou	INRIA	Control groups and namespaces
0.15	30/10/08	Yvon Jégou	INRIA	Section Isolation techniques for XtreemOS
0.16	09/12/08	Jérôme Gallard	INRIA	Section Isolation in SSI and MD flavours of XtreemOS
0.17	16/12/08	Yvon Jégou	INRIA	Section Isolation in MD flavours of XtreemOS, Introduction, Conclusion and Executive Summary
0.20	06/01/09	Yvon Jégou	INRIA	Integrated remarks from internal reviewers
1.00	09/01/09	Yvon Jégou	INRIA	Final version

Reviewers:
Guillaume Pierre (VUA), Jaka Močnik (XLAB)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.1.10	Evaluation of Linux native isolation mechanisms for all XtreemOS flavours	INRIA*, NEC, SAP, TID

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

XtreemOS allows users belonging to different administrative domains to share resources in a secure way through Virtual Organisations. In order to maximize resource usage, applications executed on behalf of the same or different VOs can use the same resources simultaneously. But the impact of each application on each other must be strictly controlled. Each grid application must be *isolated* in order to protect it from the execution environment, in order to protect its execution environment from it and in order to provide a uniform execution environment to its components across the grid.

The Linux kernel community is currently very active in providing basic mechanisms to manage local resources, to improve application sandboxing and to support virtual machines.

The control group system allows for partitioning the whole process space in groups associated to resource subsystems (CPU, memory, network, I/O, ...) and to apply policies to each group (resource limits, ...). These control groups can be directly exploited by XtreemOS: each grid application is run inside its own control group. The policies enforced by the application group can be defined from the job description (min and max on memory, limits on the number of processes, bounds on network throughput, ...).

Recent Linux kernels also provide means to manage the namespace of local application. The PID namespace, for instance, restricts the list of processes seen by the application to those belonging to its namespace. All processes of a PID namespace can be checkpointed and restarted on another node with the same PIDs. A filesystem (or mount) namespace provides a private filesystem tree to the process running in the namespace. The application can *mount* its own filesystems on its own file tree without making this local tree visible to other namespaces. A network namespace defines new network devices visible only from the namespace. It is then possible to apply new IPtable and routing rules to this device. Restricting network accessibility of grid applications allows an administrative domain manager to protect its networking environment. Associating rewriting rules to the virtual device allows to migrate the application without changing IP addresses and routes.

Support for hardware virtualization is now present in Linux. Using these mechanisms, it should be possible to run other operating systems efficiently inside an XtreemOS virtual machine and getting benefits from XtreemOS features. For instance, a non-Linux operating system can provide access to XtreemFS through a virtual device. On the other hand, it is also possible to run XtreemOS inside virtual machines. The dynamic resource management support of XtreemOS allows to create, start and stop these virtual machines “on the fly”.

XtreemOS proposes three operating systems flavours: the LinuxSSI flavour

for clusters, the desktop flavour and the mobile flavour. Exploiting the proposed mechanisms on LinuxSSI necessitates some developments on the Kerrighed kernel: the control group as well as the support for namespaces must be extended to the whole cluster. Exploiting these mechanisms on mobile devices also faces some difficulties as their implementation frequently require some support from the hardware. But the case of mobile devices is not very critical as they are not powerful enough to allow sharing between users.

Contents

Chapter 1

Introduction

XtreemOS allows users belonging to different administrative domains to share resources in a secure way through Virtual Organisations. Each administrative domain provides resources to VOs in a scalable way: no need to configure accounts for VO users in order to let them use the resources.

In order to maximize resource usage, applications executed on behalf of the same or on behalf of different VOs can use the same resources simultaneously. But the impact of each application on each other must be strictly controlled. XtreemOS must guarantee that each application gets its share of the resource (CPU cycles, memory, disk quota, ...). This is a basic requirement for running interactive applications as well as complex applications spanning multiple resource nodes. Enforcing limits on resource usage and accounting resource usage is also necessary. As compute nodes from an administrative domain run applications from users belonging to other administrative domains, XtreemOS must protect other applications running on the same node as well as the environment of the node from malicious actions. During its execution, an application creates various objects (processes, files, sockets, ...) on a resource node. XtreemOS must keep track of the owner (user or job) of each object for billing purposes and in order to clean all objects of the application when it is terminated. In order to enable seamless execution of applications on the grid, XtreemOS must provide coherent development and execution environments to VO users. As a computation node may be shared by different VOs having different software environment requirements, XtreemOS must support the provision of different execution environments to its users.

The implementation of these requirements in XtreemOS necessitates some support from the Linux kernel in order to control and limit resource usage, protect applications against each other and provide stable execution environments to VO users. This document analyses various mechanisms that have been implemented in recent Linux kernels and evaluates how these new mechanisms can be exploited

in order to implement these requirements.

Chapter 2 analyses XtreemOS requirements about resource management, confidentiality, accounting, execution environments as well as application life cycle. Chapter 3 evaluates various mechanisms of Linux kernel which can be used to monitor, limit and account resource usage, protect applications, provide an execution environment to the applications and manage the life cycle of system objects created by the applications during their execution. Chapter 4 proposes solutions to the requirements of chapter 2 mainly based on the mechanisms evaluated in chapter 3. Chapter 5 discusses the possibility to exploit these mechanisms in the LinuxSSI and mobile device flavours of XtreemOS. Finally, chapter 6 concludes this evaluation.

Chapter 2

Isolation Requirements for XtreemOS

When an application is submitted to XtreemOS for execution, XtreemOS first selects a set of available resources and then starts the execution on these resources. In the general case, these resources are not located in the user's administrative domain. Moreover, a single resource may simultaneously support the execution of multiple applications running on behalf of different Virtual Organisations. XtreemOS must protect the applications from their execution environment, protect the execution environment (local processes and data, network environment) from the applications, provide means to protect applications from failures, provide an acceptable execution environment to the applications and finally control effective resource usage.

2.1 Performance Isolation/Resource Usage Management

Resource nodes are selected in XtreemOS depending on their static characteristics (number of CPUs, memory size, performance, ...) as well as dynamic values corresponding to their current state (CPU load, free memory, networking load, ...). XtreemOS applications can request that the selected resources provide a minimal level of internal resource (CPU cycles, memory, disk quotas) during execution. Respecting minimal performance is important as XtreemOS is not a basic batch system: applications can be interactive (a minimal responsiveness is expected), the processes running on a single node can be part of a complex application spanning multiple nodes, etc. XtreemOS must be able to guarantee the minimal performance requested by an application and protect the allocated internal resources from other applications or processes running on the same node.

In order to fairly share resources between applications, XtreemOS must provide resource management at the application level. Standard Unix systems control resource usage (CPU, memory, ...) at the process level. However, complex applications are frequently multi-process applications and bounding the whole resource consumption from the individual process consumption in real time is not a simple task. As applications can specify maximal bounds for resource usage (total CPU time, maximum physical or virtual memory, maximum storage space, ...), XtreemOS must provide some means to control in real time the resource usage of whole applications. Enforcing resource usage by some application is necessary in order to guarantee minimal performance for other applications.

2.2 Confidentiality/Sandboxing

Some XtreemOS requirements request that *"It must not be possible for parties in different VOs to recognize that they are sharing resources nor to gain knowledge of what other parties are doing with those resources"*. As computing resources are nowadays more and more powerful (multi-cores, large memories, ...), sharing the computation power of these resources between multiple applications is necessary. Providing confidentiality on basic Unix system is difficult: all processes run inside common namespaces: user namespace, process namespace, filesystem namespace, network namespace, etc. Ideally, each XtreemOS application should be allowed to run inside its own exclusive environment as if it were the exclusive user of the resource node.

Sandboxing techniques can limit the system object space visible by processes (processes, files, ...) and can provide some means to enforce confidentiality and to limit the visible objects. But, as XtreemOS allows applications to be run outside their owner's administrative domain, it should also be possible to control the access rights of the applications to external objects, internet sites, for instance.

2.3 Accounting, Monitoring, Logging

Auditing, logging, accounting mechanisms must be implemented at node level to provide the information needed by the VO management and security services. As VO management and security services do not have any knowledge of application internals (processes, threads, ...), but only some application reference point (job ID, security token), this application reference point must be associated to all processes generated by the application.

2.4 Execution Environment

XtreemFS provides a uniform filesystem space to users across the resources managed by a VO. User applications can access XtreemFS files in a uniform way independently of the resources selected for execution. However, the uniform space provided by XtreemFS cannot hold all files and directories accessed by the application. This is especially the case of some program libraries which depend on the node configuration. Maintaining a uniform operating system configuration across a large grid is a difficult task: the configuration depends on the resource installation, synchronising updates on different administration domains is complex, some administration domains need to provide different configurations for independent VOs. It is possible to specify execution environment constraints (operating system) during resource selection. Although this possibility can provide uniform environments to users, it does not facilitate the administration of resources shared by multiple Virtual Organisations. XtreemOS resource nodes must support multiple compilation and execution environments.

A process identifier (PID), a user identifier (UID), a list of group identifiers (GID) and, in some cases, extra user identifier (during UID changes) are associated to each Unix process. An application can retrieve these identifiers and store them in its local memory. For instance, the application can fork a new process, store its PID in the memory in order to later send signals to it. Other execution attributes such as the internet address or the hostname of the resource node, the port number bound to a socket can also be fetched by the application. These attributes are linked to the resource selected for the execution. The UIDs and GIDs are allocated dynamically and change for each run. The host name and host internet address change for each job as a new pool of resources is selected for each execution. Even worse, these attribute might change during the execution if the application is migrated or restarted after a checkpoint. XtreemOS must offer some means to provide stable execution attributes for the applications.

2.5 Application Boundary

A typical XtreemOS application running on a resource node is made of multiple processes and threads. In order to cleanup, checkpoint, migrate and control resource usage of an application, the local operating system must permanently keep track of all processes/threads of the application. Standard Unix operating systems manage login sessions in order to globally evaluate local resource usage from login to logout. But this session management in Unix does not cover all XtreemOS needs:

- Resource usage limits are not controlled globally.

- A process can leave the login session (so that the user can logout while background processes continue to run). In order to allow scalable management of users and resources in XtreemOS, the user environment is dynamically created when the session is started and freed at the end of the session. No user process should remain active after the session ends.

- It is not possible to create new processes inside an existing session from outside the session. In XtreemOS, a session is a dynamic execution context created after negotiation with the resource manager. To each session correspond user credentials, resource usage limits as well as an accounting/monitoring context. It should be possible to launch new processes controlled by the same credentials inside an active session from outside, the resource usage of the process being controlled/accounted/monitored on behalf of the session. Two typical use cases are: a user can login into an existing session in order to debug/monitor his application; a complex application spanning multiple resources can remotely create new processes on the allocated resource nodes.

2.6 Virtual Machines

Some XtreemOS deliverables specify requirements about virtual machines. For instance, consider requirement **R16** of deliverable D4.2.3 [42].

> ***R16:*** *XtreemOS must support virtual machines (e.g. XEN, VMWare, OpenVZ/ Virtuozzo). This requirement is actually twofold.*
>
> 1. *XtreemOS should be able to run inside a virtual machine.*
> 2. *XtreemOS should be able to run virtual machines (i.e. act as the host operating system).*
>
> *The second point is important for business applications with strict isolation requirements between different VOs. This is especially true for the case of multiple VOs sharing the same physical hardware. For such scenarios with strong isolation requirements it must therefore be possible to execute applications in VMs or virtual containers/compartments, and a VO is created across a group of selected VMs/containers.*

Chapter 3

Isolation Mechanisms in Linux

Several kinds of mechanisms exist in Linux to provide Isolation. Containers or virtual machines are more and more used in this context. After giving general properties concerning this kind of system, we present the Type-I, Type-II and container virtualization system.

3.1 Virtualization Properties

A virtualization system could provide several kind of functionality.

Suspend/Restart This functionality allows one to suspend and restart a VM. This functionality is transparent to applications running inside the VM, although some network problems have been identified (lost of packets...).

Migration This functionality allows a VM to migrate from one node to another node in a transparent way for the running application inside the VM. The basic mechanism consist in suspending the VM, transfering all necessary files to the destination node, and restarting the VM.

However, this technique implies a period of unavailability of the VM. To solve this issue, the technique of Live Migration was created [9]. This technique migrates the VM on the fly. More information about this technique is provided in Section 3.2.2.

3.2 Virtualization (Type-I, Type-II)

In 1973, Goldberg proposed a formal definition of virtualization [21]. This definition is based on two types: Type-I and Type-II.

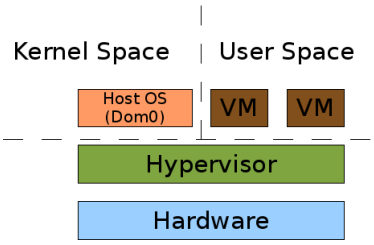

Figure 3.1: Type-I virtualization

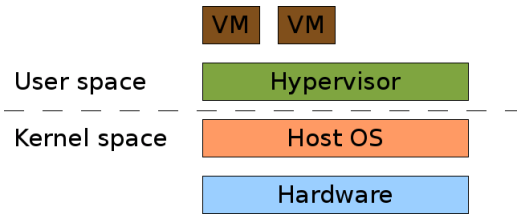

Figure 3.2: Type-II virtualization

3.2.1 Type-I, Type-II: System-level Virtualization

This approach aims to provide the virtualization of a complete system: virtual resources are exposed to a complete OS inside a VM. The system running on the VM is called guest OS. The guest OS can't run privileged instructions at the CPU level. To solve this problem, the VM forwards their requests to a host OS. The host OS has the right to execute privileged instructions on the CPU. A special piece of code (the hypervisor) is in charge of forwarding the requests from one VM to the host OS, and of scheduling multiple VMs on a physical node.

Based on this general status, Goldberg defines the Type-I (see Figure 3.1), where the hypervisor is executed directly on the bare hardware (in this manner, the host OS is installed in a special VM generally called Domain0), and the Type-II (see Figure 3.2) where the hypervisor is executed on the host OS (in this manner, the hypervisor is treated by the host OS as a simple process).

Xen [1] is an example of Type-I virtualization system. VMWare Server [40] or QEMU [2] are examples of Type-II virtualization systems.

3.2.2 Type-I Virtualization: XEN

The Type-I virtualization could be declined in two categories. The first one (the oldest) is *paravirtualization*. With paravirtualization, the guest OS code must be modified in order to execute it on a VM. The second one is *full virtualization* where, thanks to the hardware support, no guest OS code modification is necessary. Xen is a virtualization system exploiting paravirtualization (for XEN < 3.0) and full virtualization (for XEN ≥ 3.0 with the use of hardware virtualization).

In both versions, Xen is composed of domains: one domain0 and several domaineU (VMs). The domain0 is a special domain in which the kernel can execute privileged instructions. When a VM executes a privileged instruction, the request is forwarded to the domain0. This is done by synchronous or asynchronous calls (hypercall and event).

Each VM gets its own exclusive part of physical memory. The guest OS has direct access to hardware page tables, but updates are batched and validated by

the hypervisor. A domain may be allocated non-contiguous machine pages.

From the CPU point of view, the domain0 is the only one that can perform privileged instructions. However, the hypervisor (directly installed on the bare hardware) runs in the most privileged level of the CPU (generally, the level 0). In addition, traditional guest OSs also need to run on the most privileged level of instructions (generally, the level 0). This is why, in the case of paravirtualization, it's necessary to modify the guest OS to give it the illusion that it is performed on the highest level of privileged instruction (generally, level 1). With full virtualization (use of Xen $\geq$ 3.0 with the use of hardware virtualization), this problem is solved. The hypervisor runs in a level -1, whereas the guest OS runs in the level 0.

Xen does not provide hardware emulation. I/O are directly treated with the bare devices.

(i) Isolation and Security VMs are independent from each other. For instance, it is possible to reboot one VM without impacting others. In addition a VM is not visible from other ones.

(ii) Server Consolidation Generally, in a cluster (and even more in a grid!), the majority of nodes are not used at 100% of their capacity. It should be really interesting to have mechanisms to balance workload on certain nodes (to use them at 100% of their capacity) and to turn off others, before turning them on again if necessary. Server consolidation is supported by Xen.

(iii) Billing and Application Boundary Concerning billing issues, these are on the roadmap of the Xen development. However, to our knowledge, nothing is done at the time of this writing.

Concerning the VM boundary, it is possible to specify the size of the memory, the CPUs to use, etc. These parameters can be modified in real time.

(iv) Configurability and Portability Each VM could have their own guest OS. However, Xen does not provide CPU emulation, that is, it is not possible to run on a VM a guest OS or an application not compiled for the physical CPU architecture. In addition, it is not possible to migrate a VM from one physical architecture to another.

Live Migration A VM is composed of a disk image file, several configuration files and a memory file. To transfer a VM from one node to another, we have to transfer all of these files. However, this could take a lot of time according to the size of the VM and to the network bandwidth. To solve this, the concept of

live migration were invented, ie, the migration of a VM from one node to another without (or with the smallest) interruption time and transparently for all clients connected to the VM. This implies two main problems: the first one is about the migration of the VM itself, the second one is about the network connectivity of the VM.

For instance, Xen is able to do Live Migration [9]. We assume that the disk image file and all the configuration files are shared on a central repository. In that case, to do a Live Migration, we only have to transfer the memory from the originating node to the destination node. To do this, the memory of the VM is copied to the distant node (during this time, the VM is running). The modified pages of the memory are continously transfered to the distant node until only frequently modified pages remain. Then the VM is stopped and the remaining pages are transferred to the distant node. The problem is if the VM executes an application which makes lot of change on the memory. In that case, the number of pages to transfer when the VM is stopped will be important, that is, the interruption time won't be negligible.

Concerning the problem of the network connectivity, the VM is migrated with its own IP. To avoid a loss of connectivity, when the VM is restarted on the distant node, it sends "false" ARP-Reply packets to signal to other connected machines (virtual or not) that the VM has been migrated. Otherwise, it is possible to rely on the switch to detect the migration of the VM. This solution works only when we migrate a VM on the same network, otherwise, it is necessary to use other solutions to virtualize the network (VPN, VON, Distributed servers [43]...).

3.2.3 Type-II Virtualization

VMWare Server VMWare Server is a commercial product [40] of Type-II virtualization system type. This product is developed to be executed on x86 CPU architecture. In addition it can take advantage of the latest hardware virtualization technologies such as Intel-VT and AMD-Pacifica.

VMWare Server is compatible with a lot of guest and host OSs (Linux and Windows). Moreover, it provides the Virtual SMP technology, allowing one VM to have multiple CPUs. VMWare Server is able to do snapshotting, suspend/restart, and migration of VMs (Live Migration is possible in the commercial version of VMWare Server).

In addition, VMWare Server defines a *Virtual Machine Communication Interface* (VMCI) allowing fast communications between VMs, or between VMs and the host OS on the same node. Without VMCI, VMs communicate with the host OS using the network layer. Using the network layer adds overhead to the communication. With VMCI communication overhead is minimal and different tasks that require communication can be optimized.

Moreover, VMWare Server proposes a VIX API, allowing users to manipulate their VMs via scripts or programs. The VIX API runs on the Microsoft Windows and Linux platforms. Client codes can be written in C, Perl and COM.

VMWare Server does not provide emulation of the CPU. In this way, from a CPU point of view, performances of VMware Server are close to the native ones, however it is not possible to migrate a VM from one physical architecture to a different physical architecture.

(i) Isolation and Security Like Xen, VMware VMs are independent from each other and a VM is not visible from other ones.

(ii) Server Consolidation VMWare Server is able to do server consolidation.

(iii) Billing and Application Boundary Some commercial versions of VMWare seem to allow billing.

Concerning the boundary, it is possible to limit the use of memory etc.

(iv) Configurability and Portability Each VM could have their own guest OS. However, VMWare Server does not provide CPU emulation, that is, on a VM, it is not possible to run a guest OS or an application not compiled for the physical CPU architecture. In addition, it is not possible to migrate a VM from one physical architecture to another.

The commercial version of VMWare can provide Live Migration.

QEMU QEMU [2, 3] is a generic and open source machine emulator and virtualizer.

When used as a machine emulator, QEMU can run OSes and programs compiled for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves good performances. QEMU supports a lot of host CPUs and can emulate a lot of CPU architectures.

When used as a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU. A host driver called the QEMU accelerator (also known as KQEMU) is needed in this case. The virtualizer mode requires that both the host and guest machine use x86 compatible processors (same as VMWare Server).

(i) Isolation and Security Like VMWare Server, VMs are independent from each other and a VM is not visible from other ones.

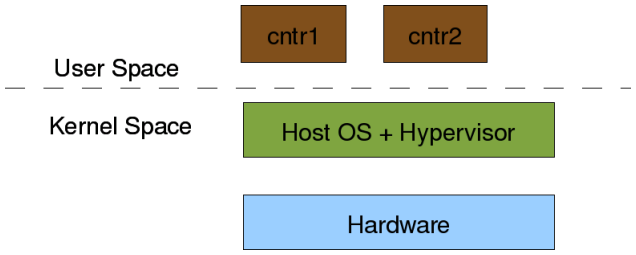

Figure 3.3: Containers virtualization

(ii) Server Consolidation QEMU is able to do server consolidation.

(iii) Billing and Application Boundary To our knowledge, nothing has be done concerning billing.

Concerning the boundary, it is possible to limit the use of memory etc.

(iv) Configurability and Portability Each VM can have its own guest OS. In addition, QEMU provides emulation of the CPU, that is, it is possible to run on a VM a guest OS or an application not compiled for the physical CPU architecture. In that way, it is possible to migrate a VM from one physical architecture to another in a transparent way for the guest OS running inside the VM. However, QEMU does not provide Live Migration.

3.3 Containers

3.3.1 Process-level Virtualization

This technique is not addressed by the Goldberg theory. However, it is used increasingly nowadays. It consists of executing several processes simultaneously on several containers. In that case, there is no guest OS on the containers: all containers run the kernel of the host OS and can execute privileged instructions (see Figure 3.3).

3.3.2 OpenVZ

OpenVZ [33] is a container virtualization system, ie, there is one special kernel on the host OS that executes privileged instructions of all the containers. This special kernel is OpenVZ (normal kernel + OpenVZ patch) which runs directly on the bare hardware. OpenVZ works with the concept of Virtual Environment

(VE). A VE has its own configuration (IP address, users, specific packages, ...). In this way, OpenVZ allows to create VEs with different distributions in a easy manner.

OpenVZ is able to do Live Migration of containers. In addition, OpenVZ provides container checkpointing and it is highly scalable (for instance, it is possible to run 120 VEs on top of nodes with 750MB of RAM memory).

OpenVZ provides several properties like: (i) isolation and security, (ii) server consolidation, (iii) accounting and boundary, (iv) Configurability and portability of the virtual environment.

(i) Isolation and Security VEs are independent from each other, for instance, it is possible to reboot one VE without impacting others. In addition processes of one container are not visible from other ones. However, the same kernel is used to execute privileged instructions of all containers. In this manner, a malicious user could hijack the system to take its control.

(ii) Server Consolidation With OpenVZ, it is possible to consolidate servers by migrating VEs (in order to reduce the power consumption. for instance).

In addition it is possible to dynamically reallocate resources to a VE. For example, it is possible to dynamically add more memory to a VE.

(iii) Billing and Application Boundary The resource management provided by OpenVZ is composed of three elements: (i) disk quotas, (ii) fair CPU scheduler, (iii) user beancounters. This approach gives more dynamicity and flexibility in the management of resources.

- Disk quotas: they are defined for each VE by the administrator of the host OS (quotas are adjustable in real time).
- Fair CPU scheduler: it allows the administrator of the system to specify a percentage of time of the processor to each VE.
- User beancounters: it is a set of counters, boundaries and guarantees attributed to a VE. There are approximately 20 parameters available to configure a VE. This ensures that no VE can take all resources to the detriment of other ones.

(iv) Configurability and Portability of the Virtual Environment According to the type of users and usage (developers, testers, host providers, students, ...) it is often necessary to deploy on a same physical hardware several Linux distributions. This operation can be very expensive in term of time, and very tiresome. OpenVZ

gives an element of solution to this issue by allowing users to deploy several VEs on a same node. A VE can be created in 60 seconds, and it is easy to clone a VE with a negligible degradation of performances.

Concerning application portability, it is not necessary to modify them before their execution on a VE, however, it is not possible to migrate one container from a physical architecture to another.

3.4 Summary Type-I, Type-II, Containers

Prop. / Virt. Sys.[1]	XEN > 3.0	VMWare Server	QEMU	OpenVZ
Isolation & Security	YES[2]	YES	YES	0.5 YES[3]
Server Consolidation	YES	YES	YES	YES
Accounting & Boundary	-[4]	-	-	YES
Conf. & Port. of App.[5]	0.5 YES	0.5 YES	YES	0.5 YES
Live Migration	YES	NO[6]	NO	YES

3.5 Control Groups

This description has been extracted from LWN article [13] and Linux kernel documentation [30].

Control Groups (`cgroup`) provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.

- A *cgroup* associates a set of tasks with a set of parameters for one or more subsystems.

- A *subsystem* is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualization subsystem.

[2]Properties / Virtualization System
[3]YES: the virtualization system provides the given property
[4]0.5 YES: the virtualization system does not provide completely the given property
[5]-: Not known
[6]Configurability and portability of applications
[7]NO: the virtualization system does not provide the given property

- A *hierarchy* is a set of cgroups arranged in a tree, such that every task in the system is in exactly one of the cgroups in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each cgroup in the hierarchy. Each hierarchy has an instance of the cgroup virtual filesystem associated with it.

At any time there may be multiple active hierarchies of task cgroups. Each hierarchy is a partition of all tasks in the system. A `cgroup` hierarchy filesystem can be mounted for browsing and manipulation from user space. For instance, the following command mounts the `cpuacct` and `cpuset` subsystems, creates two subgroups and forks a new shell inside each subgroup.

```
 (1) mkdir /xos-groups
 (2) mount -t cgroup -o cpuacct,cpuset none /xos-groups/
 (3) mkdir /xos-groups/app1
 (4) mkdir /xos-groups/app2
 (5) echo 0 > /xos-groups/app1/cpuset.cpus
 (6) echo 1 > /xos-groups/app2/cpuset.cpus
 (7) echo 0 > /xos-groups/app1/cpuset.mems
 (8) echo 0 > /xos-groups/app2/cpuset.mems
 (9) xterm &
(10) echo $! > /xos-groups/app1/tasks
(11) xterm &
(12) echo $! > /xos-groups/app2/tasks
```

Instruction `(2)` mounts subsystems `cpuacct` for cpu accounting and `cpuset` for cpu scheduling on directory `/xos-groups`. This root control group contains all system processes. Instructions `(3)` and `(4)` create two new subgroups in the hierarchy, initialised in `(5-8)`. The new task created in `(9)` is moved to subgroup `app1` in `(10)`. The task created in `(11)` is moved to subgroup `app2` in `(12)`. CPU allocation for these tasks is controlled by their attached subgroup: CPU 0 for `app1` and CPU 1 for `app2`. Effective CPU usage of each subgroup can be obtained from file `cpuacct.usage` of each subgroup.

```
# cat app1/cpuacct.usage
22740167
```

User level code may create and destroy cgroups by name in an instance of the cgroup virtual file system, specify and query to which cgroup a task is assigned, and list the task pids assigned to a cgroup. Those creations and assignments only affect the hierarchy associated with that instance of the cgroup file system.

The main purpose of control groups is to provide a generic framework where several "resource controllers" can plug in and manage different resources of the

system used by whole process groups. Typical examples of resources which can be managed at group level (whole applications) are memory, CPU, I/O. Cgroups also offers a unified user interface, based on a virtual filesystem where administrators can assign arbitrary resource constraints to a group of chosen tasks. For example, Cpusets and Group Scheduling have been merged in 2.6.24. The first one allows to bind CPU and Memory nodes to the arbitrarily chosen group of tasks and the second one allows to bind a CPU allocation policy to the cgroup.

The following control groups have been developed so far: memory resource controller, swap subsystem, fair scheduler, CPUsets, accounting, container freezer and block I/O bandwidth tracker.

3.5.1 Memory Control Group

This description has been extracted from LWN article [11] and Linux kernel community page [28].

The memory resource controller in 2.6.25 is a cgroups-based feature. The memory resource controller isolates the memory behavior of a group of tasks from the rest of the system. It can be used to:

- Isolate an application or a group of applications. Memory hungry applications can be isolated and limited to a smaller amount of memory.
- Create a cgroup with limited amount of memory. This can be used as a good alternative to booting with `mem=XXXX`.
- Control the amount of memory that virtualization solutions want to assign to a virtual machine instance.
- Allow a CD/DVD burner to limit the amount of memory used by the rest of the system to ensure that burning does not fail due to lack of available memory.

Configuration, like all cgroups, is done by mounting the `cgroup` filesystem with the "`-o memory`" option, creating a randomly-named directory (the cgroup), adding tasks to the cgroup by '`cat`'ting its PID to the '`task`' file inside the cgroup directory. The behaviour of each memory control group is controlled using the following files inside the cgroup directory:

`memory.limit_in_bytes`,

`memory.usage_in_bytes` (memory statistic for the cgroup),

`memory.stats` (more statistics: RSS, caches, inactive/active pages),

`memory.failcnt` (number of times that the cgroup exceeded the limit),

`memory.mem_control_type` (type of memory pages managed by the control group).

Out of memory conditions (OOM) are also handled in a per-cgroup manner: when the tasks in the cgroup surpass the limits, OOM will be called to kill a task between all the tasks involved in that specific cgroup.

3.5.2 Swap Control Group

This description has been extracted from Linux container mailing-list [32].

Even when memory usage is limited by a cgroup memory subsystem or isolated using CPUSet, swap space is still shared by all processes. If one application or process uses all the swap space, it can affect other applications.

The swap subsystem cgroup adds swap space management (limit, charge) to the memory subsystem.

3.5.3 CPUSets Control Group

This description has been extracted from Linux kernel documentation [15].

Cpusets provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks. A CPUSet constrains the CPU and Memory placement of tasks to only the resources within a task's current CPUSet. They form a nested hierarchy visible in a `cgroup` virtual file system.

Requests by a task, using the `sched_setaffinity(2)` system call to include CPUs in its CPU affinity mask, and using the `mbind(2)` and `set_mempolicy(2)` system calls to include Memory Nodes in its memory policy, are both filtered through that task's cpuset, filtering out any CPUs or Memory Nodes not in that cpuset. The scheduler will not schedule a task on a CPU that is not allowed in its `cpus_allowed` vector, and the kernel page allocator will not allocate a page on a node that is not allowed in the requesting tasks `mems_allowed` vector.

User level code may create and destroy cpusets by name in the cgroup virtual file system, manage the attributes and permissions of these cpusets and which CPUs and Memory Nodes are assigned to each cpuset, specify and query to which cpuset a task is assigned to, and list the task pids assigned to a cpuset.

3.5.4 Accounting Control Groups

This description has been extracted from Linux kernel documentation [8] for `cpuacct` and [7] for `cgroupstats`.

The `cpuacct` control group allows to count the CPU cycles consumed by group of processes. The `cgroupstats` control group extends the taskstats systems to record resource usage and statistics from groups of tasks. Using `cgroupstats`, it is possible to get the CPU usage (user, system), the number of major/minor page faults, the memory (physical and virtual) usage as well as I/O statistics about a group of processes in real time.

3.5.5 Completely Fair Scheduler Control Group

This description has been extracted from Linux kernel documentation [6].

The CFS Linux scheduler can be configured (at kernel compile-time) to divide CPU time fairly among arbitrary groups of tasks managed by the control group system. The ratio of cpu cycles allocated to a task group is defined in file `cpu.shares` of the cgroup directory. Here is an example from [6] showing how CPU bandwidth can be shared between two applications.

```
# mkdir /dev/cpuctl
# mount -t cgroup -ocpu none /dev/cpuctl
# cd /dev/cpuctl

# mkdir multimedia # create "multimedia" group of tasks
# mkdir browser    # create "browser" group of tasks

# #Configure the multimedia group to receive twice the
# # CPU bandwidth that of browser group

# echo 2048 > multimedia/cpu.shares
# echo 1024 > browser/cpu.shares

# firefox &         # Launch firefox and move it to
#                   # "browser" group
# echo <firefox_pid> > browser/tasks

# #Launch gmplayer (or your favourite movie player)
# echo <movie_player_pid> > multimedia/tasks
```

3.5.6 Container Freezer Control Group

This description has been extracted from Linux container mailing-list [26] and LWN article [27]. The `swsusp` modules is described in LWN article [41].

The freezer subsystem in the container filesystem defines a cgroup file named `freezer.state`. Reading `freezer.state` will return the current state of the cgroup. Writing "`FROZEN`" to the state file will freeze all tasks in the cgroup. Subsequently writing "`RUNNING`" will unfreeze the tasks in the cgroup.

The cgroup freezer is useful for batch job management systems that start and stop sets of tasks in order to schedule the resources of a machine according to the wishes of a system administrator. This sort of program is often used in HPC clusters to schedule access to the cluster as a whole. The cgroup freezer uses cgroups to describe the set of tasks to be started/stopped by the batch job management system. It also provides a means to start and stop the tasks composing the job.

The cgroup freezer can also be useful for checkpointing groups of tasks. The freezer allows the checkpoint code to obtain a consistent image of the tasks by attempting to force the tasks in a cgroup into a quiescent state. Once the tasks are quiescent another task can walk `/proc` or invoke a kernel interface to gather information about the quiesced tasks. Checkpointed tasks can be restarted later should a recoverable error occur. This also allows the checkpointed tasks to be migrated between nodes in a cluster by copying the gathered information to another node and restarting the tasks there.

Example of usage:

```
# mkdir /containers/freezer
# mount -t cgroup -ofreezer freezer /containers/freezer
# mkdir /containers/freezer/0
# echo $some_pid > /containers/freezer/0/tasks
```

to get status of the freezer subsystem :

```
# cat /containers/freezer/0/freezer.state
RUNNING
```

to freeze all tasks in the container :

```
# echo FROZEN > /containers/freezer/0/freezer.state
# cat /containers/freezer/0/freezer.state
FREEZING
# cat /containers/freezer/0/freezer.state
FROZEN
```

to unfreeze all tasks in the container :

```
# echo RUNNING > /containers/freezer/0/freezer.state
# cat /containers/freezer/0/freezer.state
RUNNING
```

to kill all tasks in the container :

```
# echo 9 > /containers/freezer/0/signal.kill
```

The container freezer cgroup subsystem utilizes the swsusp freezer to freeze a group of tasks [41].

3.5.7 Block I/O Bandwidth Tracking Control Group

Different block I/O bandwidth tracking and management systems have been proposed for Linux, for instance: dm-ioband (v1.7.0) described in LWN article [37], 2-Layer CFQ described in Linux kernel mailing-list [38] and io-throttle (v11) discussed in mailing-list [34].

Bio-cgroup is an I/O tracking mechanism, implemented in the cgroup memory subsystem [36]. With this mechanism, it is able to determine to which cgroup eachI/O belongs to, even when the I/O is one of delayed-write requests issued from a kernel thread such as `pdflush`.

Dm-ioband is an I/O bandwidth controller implemented as a device-mapper driver, which gives specified bandwidth to each job running on the same block device [37]. A job is defined as a group of processes with the same `PID` or `PGRP` or `UID` or a virtual machine such as KVM or Xen. A job can also be a cgroup managed by the bio-cgroup system.

3.6 Namespaces

A Unix process runs inside a set of namespaces: file namespace (file system tree), process namespace (PIDs of the active processes), user namespace (user/group UID), tty namespace, device namespace, networking namespace (network interfaces with associated routing tables), IPC and shared segments namespace, etc. These namespaces are common and shared by all processes on traditional systems.

In order to increase security, facilitate application management, and implement application checkpoint/restart/migration, recent Unix operating system (BSD, Linux, Solaris, etc.) have introduced means to manage namespaces at the application level. The current Linux kernel (2.6.24) manages six different namespaces at task level: UTS (the system names), PID, USER (UIDs), NS (mount), IPC and NET (network). Other namespaces, such as device and/or PTY namespace might be integrated in the future.

3.6.1 Network Namespace

This description has been extracted from LWN articles [5] and [12] and from Sourceforge documentation [31].

A network namespace is a private set of network resources assigned to one or several processes. These have their own set of network devices, IP addresses, routes, sockets etc. Other processes outside of the namespace cannot access these network resources, nor even know they exist. That allows:

- virtualization : processes inside the network namespaces do not know anything about the network resources outside the namespace and use the resources without conflicting with other network namespaces.
  - For examples: several network namespaces can have `eth0` and `lo` network devices.
  - several apache servers listening on *:80 can be launched into different network namespaces.
- isolation : processes cannot access the network resources that are outside the namespace.
  - For examples: a process cannot sniff traffic related to another network namespace.
  - a process cannot shutdown an interface belonging to another network namespace.

The virtualization/isolation allows to implement different interesting features:

- security : a server can be run into a network namespace which ensures, if the server is hacked, that the rest of network system will not be compromised;
- resource management : resource management acting at the network device can be easily implemented because network resources can be assigned to a specific set of processes;
- traffic control : control is more flexible because it can be set by network devices and obviously by network namespaces;
- consolidation : a powerful host can aggregate several servers in different network namespaces without impacting the servers' network configurations;
- mobility : it is easy to find and checkpoint network resources because they are by namespace. The virtualization allows to move IP across the network and avoid conflicts at restart;

Network interfaces made up of real hardware normally remain in the root namespace. Communication with other namespaces is made possible by way of a "virtual Ethernet" device. A virtual device can be thought of as a wire into a restricted namespace; it presents one device within that namespace and one in the parent (normally root) namespace. Packets written to one end show up at the other. With the addition of a few routing rules in the root namespace, packets meeting the right criteria can be directed to (and from) specific namespaces.

When the `CLONE_NEWNET` flag is set in the `clone` syscall, the process child is created inside a new network namespace. Setting this flag in the `unshare` syscall allows to create a new network namespace for the calling process. All the process children inherit the network namespace from their parent.

3.6.2 UTS Namespace

This description has been extracted from LWN article [22] and sourceforge documentation [39].

Instead of using the global `system_utsname` containing hostname, domainname etc, a process can request it's copy of the `uts` structure info to be cloned. The data is be copied from the original, but any further changes will not be seen by processes which are not it's children, and vice versa. The `utsname` information includes a system's hostname. Allowing this to be unshared means that processes in virtual servers can be associated with different hostnames. It also means that a job being migrated can take its hostname with it if needed.

3.6.3 PID Namespace

This description has been extracted from LWN article [18].

The PID namespace allows for creating sets of tasks, with each such set looking like a standalone machine with respect to process IDs. In other words, tasks in different namespaces can have the same IDs.

This feature is the major prerequisite for the migration of containers between hosts; having a namespace, one may move it to another host while keeping the PID values – and this is a requirement since a task is not expected to change its PID. Without this feature, the migration will very likely fail, as processes with the same IDs can exist on the destination node, which will cause conflicts when addressing tasks by their IDs.

PID namespaces are hierarchical; once a new PID namespace is created, all the tasks in the current PID namespace will see the tasks (i.e. will be able to address them with their PIDs) in this new namespace. However, tasks from the new namespace will not see the ones from the current. This means that now each task has more than one PID – one for each namespace.

To create a new namespace, one should just call the `clone(2)` system call with the `CLONE_NEWPID` flag set. After this, it is useful to change the root directory and mount a new `procfs` instance in the `/proc` to make the common utilities like `ps` work. Note that since the parent knows the PID of its child, it may `wait()` in the usual way for it to exit.

The process created by the `clone` system call with the `CLONE_NEWPID` flag set gets PID 1 inside the new namespace and behaves as the `init` process of classical Unix systems: orphan processes of the namespace are reparented on this process. If this process is killed, all processes of the namespace are killed and the namespace is freed.

3.6.4 IPC Namespace

This description has been extracted from LWN article [29].

This patch set allows to create a new copy of the current IPC namespace and to manage a private set of IPC objects (sem, shm, msg) inside this new IPC namespace. Basically, it is another building block of containers functionality.

3.6.5 User Namespace

This description has been extracted extracted from LWN article [20] and from Linux container mailing-list [23].

The user namespace allows for creating sets of tasks, with each such set looking like a standalone machine with respect to user IDs. Users in different namespaces can have the same UIDs. Independent user namespaces allow the implementation of containers (each container manages its users independently) and of checkpoint/restart/migrate (processes must be restarted with the same UIDs).

3.6.6 Mount Namespace

This section is based on the *sharedsubtree* document in LWN article [10] and on *IBM's resource for developers* article [24].

In traditional Linux systems, all processes run in the same filesystem tree. Using mount namespace, it is possible to associate a mount tree to a group of processes, a single application for instance. This possibility increases security and confidentiality (an application does not see the file system tree of other applications), allows building different filesystem trees for different execution environments (each Virtual Organisation can define its own filesystem tree across different physical organisations), and in some cases, can provide the capability to manage its own mount tree to an application (user-space mounting).

A process can be forked in a new namespace (initially a clone of it parent's namespace) using the `CLONE_NEWNS` flag. A process can also request to clone its current namespace using the `unshare` system call. It is also possible to clone a mount using the `mount -bind` command. Cloned mount namespaces are not completely independent: the result on a mount request inside some namespace on other namespaces depend on the status of the mount point. The new `mount` system call manages `shared`, `slave`, `private` and `unbindable` mount points.

- `mount -make-shared mount-point`: `mount-point` is shared. A shared mount can be replicated to as many mountpoints and all the replicas continue to be exactly same: mount/umount requests from the parent are propagated to the child and vice-versa.

- `mount -make-slave mount-point`: `mount-point` is slave. A slave mount is like a shared mount except that mount and umount events only propagate towards it. All slave mounts have a master mount which is shared. `mount/umount` requests from the shared/master are propagated to slaves, but slave mounts are not propagated

- `mount -make-private mount-point`: `mount-point` is private. A private mount does not forward or receive propagation. This is the default behaviour.

- `mount -make-unbindable mount-point`: `mount-point` is unbindable.

Each of these options has a recursive version which is applied to the whole subtree.

It is possible to automatically provide a new mount namespace to users at login time using the `pam_namespace` module [16].

Another extension to the Linux mount system allows to make parts of filesystems read-only in some namespaces using read-only bind mounts (see [25]).

3.6.7 Unshare System Call

System call `unshare(2)` allows a process to disassociate parts of its execution context that are currently being shared with other processes. Part of the execution context, such as the namespace, is implicitly shared when a new process is created using `fork(2)` or `vfork(2)`, while other parts, such as virtual memory, may be shared by explicit request when creating a process using `clone(2)`.

The main use of `unshare()` is to allow a process to control its shared execution context without creating a new process.

The flags argument is a bit mask that specifies which parts of the execution context should be unshared. This argument is specified by ORing together zero or more of the following constants:

CLONE_FILES Reverse the effect of the `clone(2) CLONE_FILES` flag. Unshare the file descriptor table, so that the calling process no longer shares its file descriptors with any other process.

CLONE_FS Reverse the effect of the `clone(2) CLONE_FS` flag. Unshare file system attributes, so that the calling process no longer shares its root directory, current directory, or umask attributes with any other process.

CLONE_NEWNS This flag has the same effect as the `clone(2) CLONE_NEWNS` flag. Unshare the namespace, so that the calling process has a private copy of its namespace which is not shared with any other process. Specifying this flag automatically implies `CLONE_FS` as well.

CLONE_VM If `CLONE_VM` is set, the virtual memory of the caller is disassociated from the shared virtual memory.

Note that the current implementation of `unshare` does not support cloning the PID namespace (no `CLONE_NEWPID` flag).

3.6.8 Hijack System Call

This description has been extracted from LWN article [14].

The proposed `hijack()` system call is an extension to `clone()` that causes the new process to share resources with a specified third process rather than with the parent. Its main reason for existence is to make it easy to enter different namespaces.

3.7 `libvirt`'s LXC (LinuX Container)

This description has been extracted from web page [4].

Library `libvirt` provides a stable API for managing virtualization hosts and their guests. It started with a Xen driver, and over time has evolved to add support for QEMU, KVM, OpenVZ and recently a driver called "LXC" (short for "LinuX Containers"). The key is that no matter what hypervisor is being used, there is a consistent set of APIs, and standardized configuration format for userspace management applications in the host (and remote secure RPC to the host).

The LXC driver is the result of a combined effort from a number of people in the `libvirt` community.

Overall, libvirt wants to be the defacto standard, open source management API for all virtualization platforms and native Linux virtualization capabilities are a strong focus. The LXC driver is attempting to provide a general purpose management solution for two container virtualization use cases:

- Application workload isolation,
- Virtual private servers.

The first use case provides the ability to run an application in primary host OS with partial restrictions on its resource / service access. It still runs with the same root directory as the host OS, but its filesystem namespace may have some additional private mount points present. It may have a private network namespace to restrict its connectivity, and it ultimately has restrictions on its resource usage (eg memory, CPU time, CPU affinity, I/O bandwidth).

The second use case provides a completely virtualized operating system in the container (running the host kernel of course), akin to the capabilities of OpenVZ / Linux-VServer. The container has a totally private root filesystem, private networking namespace, whatever other namespace isolation the kernel provides, and again resource restrictions. Some people like to think of this as 'a better chroot than chroot'.

In terms of technical implementation, at its core is direct usage of the new `clone()` flags. By default all containers get created with `CLONE_NEWPID`, `CLONE_NEWNS`, `CLONE_NEWUTS`, `CLONE_NEWUSER`, and `CLONE_NEWIPC`. If private network config was requested they also get `CLONE_NEWNET`.

The workload isolation case, after creating the container, just adds a number of filesystem mounts in the containers private FS namespace. The VPS case, do a `pivot_root()` onto the new root directory, and then add any extra filesystem mounts the container config requested.

The `stdin/out/err` of the process leader in the container is bound to the slave end of a Pseudo TTY, `libvirt` owning the master end so it can provide a virtual text console into the guest container. Once the basic container setup is complete, `libvirt` exec the so called '`init`' process. Things are thus setup such that when the '`init`' process exits, the container is terminated / cleaned up.

On the host side, the `libvirt` LXC driver creates what is called a '`controller`' process for each container. This is done with a small binary `/usr/libexec/libvirt_lxc`. This is the process which owns the master end of the Pseudo-TTY, along with a second Pseudo-TTY pair. When the host admin wants to interact with the container, they use the command '`virsh console CONTAINER-NAME`'. The LXC controller process takes care of forwarding I/O between the two slave PTYs, one slave opened by `virsh` console, the other being the containers' `stdin/out/err`. If the controller is killed, then the container also dies.

Basically one can think of the `libvirt_lxc` controller as serving the equivalent purpose to the '`qemu`' command for full machine virtualization - it provides the interface between host and guest, in this case just the container setup, and access to text console - perhaps more in the future.

For networking, libvirt provides two core concepts:

- Shared physical device. A bridge containing one of the physical network interfaces on the host, along with one or more of the guest `vnet` interfaces. So the container appears as if its directly on the LAN;
- Virtual network. A bridge containing only guest `vnet` interfaces, and NO physical device from the host. IPtables and forwarding provide routed (+ optionally NATed) connectivity to the LAN for guests.

The latter use case is particularly useful for machines without a permanent wired ethernet - eg laptops, using wifi, as it lets guests talk to each other even when there's no active host network. Both of these network setups are fully supported in the LXC driver in presence of a suitably new host kernel.

3.8 Smack

This section has been extracted from LWN articles [17] and [35].

Smack implements mandatory access control (MAC) using labels attached to tasks and data containers, including files, SVIPC, and other tasks. Smack is a kernel based scheme that requires an absolute minimum of application support and a very small amount of configuration data.

Smack attaches labels to tasks and system objects. The default rule is that a task can access an object only if they have the same smack label. This default rule provides a simple compartmentalization of processes and objects.

Chapter 4

Isolation Techniques for XtreemOS

4.1 Performance Isolation

In order to guarantee acceptable performance for an application, each subsystem manager (memory, network, disk, ...) of the resource must provide some means to guarantee that the application gets a minimal share of the subsystem. Performance isolation seems straightforward using virtual machines as the major components of the physical resource have been partitionned between virtual machines: disk space, memory, physical devices. However, some resources such as bandwidth (I/O, network, CPU) are shared between the virtual machines and must be controlled.

On the other hand, control groups allow a partition of the process space and the management of subsystem resource usage by each partition. However, in general, the control groups currently implemented in Linux do not provide any means to guarantee that a group receives a minimal part of the resource. The memory control group, for instance, tracks the memory usage and manages an upper bound of memory usage. But there is no lower bound management. Low watermark management (stopping page reclaim from the group when memory usage is lower than this watermark) seems to be planned in the future. It is still possible to protect the memory allocated to some application through a global management the memory control groups. Another possibility is to combine the memory control groups with CPUSet: CPUSet allows to assign memory nodes to control groups.

4.2 Resource Usage Management

Effective resource usage accounting is a major requirement in XtreemOS. Control groups allow to track resource usage for each subsystem: CPU cycles, memory, I/O bandwidth, etc. In order to track resource usage during an XtreemOS session

(corresponding to one job), all processes attached to the session must be attached to the session control groups.

Tracking resource usage with virtual machines is a bit different. We need to consider two different cases: the case where the user can install his own operating system in the virtual machine and the case where an existing XtreemOS virtual machine runs a user application. In the first case, it is not possible to rely on the virtual machine for resource usage management: resource usage must be evaluated by the hypervisor (memory, disk space allocated to the virtual machine; CPU and I/O bandwidth consumed by the virtual machine). In the second case, it is possible to combine hypervisor resource evaluation and virtual operating system computation.

4.3 Confidentiality/Sandboxing

Running applications inside different PID namespaces ensures full isolation between application processes: processes of one application have no knowledge of other application processes. Running applications inside different mount namespaces provide isolation of application data. Using mandatory access control (SELinux or Smack), it is possible to limit the actions of user applications upon their execution context and to provide some level of sandboxing. It is also possible to run applications inside private network namespaces. Each network namespace has its own routing and filtering rules and can limit the internet visibility of applications (application firewalling).

Running applications inside different virtual machines ensures confidentiality and enables sandboxing.

4.4 Accounting

Linux natively provides support for accounting resource usage (CPU, swap, memory, I/O) at task level using the `taskstat` system. Using `cgroupstats` control group, accounting can be extended to groups of processes. Resource usage can be monitored in real time.

Virtual machines partition some physical resources (memory, disk partition) and time-share other resources such as network bandwidth and CPU cycles. In order to provide accurate accounting, it is necessary to combine effective resource usage accounted by the virtual operating system and resource usage measured by the hypervisor.

4.5 Execution Environment

The most natural way to provide uniform execution environments is through virtualization: a VO can configure a virtual machine model and deploy it on each administration domain. If a virtual machine runs a single application, it is even possible to always provide the same UID/GIDs to a user. It might even be possible to always provide the same host name and IP address as long as no external service is exported by the application.

A similar environment can also be provided through namespace management: each VO provides its own environment (file tree) and the applications are rooted on the VO's tree. It should be possible to run each application inside its own namespace (mount, PID, UID, IPC and NET) in order to provide a uniform environment.

4.6 Application Boundary

XtreemOS must track application boundaries (containing all system objects created on behalf of the application) in order to implement accounting, checkpointing and cleanup. Virtual machines provide a simple way to locate application objects. It is even possible to run accounting on the whole virtual machine, to checkpoint it and to shutdown the virtual machine when the application terminates. A similar result can be obtained if each application runs inside its own namespaces (container): all objects of the application namespaces belong to the application. For instance it is possible to fork each application on a resource node inside a new PID namespace. The PID namespace isolates the application from other processes running on the same node. The initial process become the parent of all processes created by the application and ensures that no more process is running when the application is terminated. The termination of this initial process corresponds to the termination of the application. Killing the initial process kills the whole application.

Both solutions lock the application objects inside their respective namespaces: these objects have no way to escape from their namespaces. On the other hand, there is no simple means to inject (create) new objects from outside. For instance giving users the possibility to *log in* their application context using xos-ssh can be tricky in both cases. If a virtual machine has a routable IP address, it can be configured to run an `xos-sshd` daemon. In the other case, it is necessary to install some kind of proxy/tunnel on the node hypervisor. When the application is *jailed* inside private namespaces, running an `xos-sshd` daemon inside the network namespace is also possible as long as it is possible to associate a routable IP address to each network namespace. If no routable IP address can be used, it

should be possible to run a shared `xos-sshd` on the host and to use the experimental `hijack()` (see section 3.6.8) system call in order to fork a user process inside the application namespace.

4.7 Virtual Machines

XtreemOS can be run inside virtual machines. The main restriction is that, in order to export services (core and resource nodes), these virtual machines must have routable IP addresses.

An XtreemOS node should have the capability to run virtual machines using standard Linux virtualization support (VMWare, KVM, ...). Predefined virtual machines can be used to run application submitted by users in a well defined environment (see above). It should also be possible to allow users to submit their own virtual machine for execution on the grid.

Chapter 5

Isolation in LinuxSSI and Mobile Device Flavours of XtreemOS

5.1 LinuxSSI

In this section, we present a systematic analysis of the combination of SSI and virtualization technologies [19].

5.1.1 Containers upon Single System Image

With this architecture, the SSI abstracts the distributed aspect of the resources. Based on this "simplified" and "unified" view of the distributed system, global resources can be dynamically and transparently assigned to containers in order to fit applications needs at best. In other terms, containers could dispose of more resources than is available on one node.

5.1.2 Single System Image upon Containers

This architecture is not realistic because no individual kernel can run in a container, only user-level applications can be hosted.

5.1.3 Type-I Virtualization upon Single System Image

This approach enables the implementation of a "global Type-I hypervisor', including SSI features into the hypervisor. Such a global hypervisor can transparently and globally manage resources (creation of an SMP illusion) and typically the resources allocated to VMs are not restricted to the local resources.

5.1.4 Single System Image upon Type-I Virtualization

In this case, a hypervisor is deployed on all cluster nodes and the SSI is executed in different VMs; each VM being potentially hosted by different hypervisors.

5.1.5 Type-II Virtualization upon Single System Image

The SSI globally manages all the distributed resources; the Type-II virtualization hypervisor can therefore allocate distributed resources to VMs on demand in a transparent manner.

5.1.6 Single System Image upon Type-II Virtualization

As for Type-I, each node runs VMs, and the SSI is deployed upon them.

5.1.7 Perspectives

Currently, namespaces, containers, cgroups, ... are not managed by LinuxSSI (Kerrighed). The problem is not simple to solve. The Kerlabs company considers to work on it during the next year (2009). They will study the possibility to (i) integrate all namespaces into Kerrighed and/or to (ii) put Kerrighed on a container. The case (i) was studied in the Section 5.1.1 and the conclusion is Kerrighed could make containers taking advantage of cluster resources. The case (ii) was studied in the Section 5.1.2 and the conclusion is it is not realistic. However, like the case studied in the Section 5.1.4 and the Section 5.1.6, this kind of solution could allow a user to share parts of its resources (via Kerrighed) with others. In this manner, it is possible for a user to specify resource limitation for him and for others.

All these perspectives need to be studied. This is why it is not currently possible to define a clear roadmap.

5.2 Mobile Devices

As long as they are not shared between users and they are used as pure client nodes in XtreemOS, mobile devices do not need to provide any form of isolation: only one user is logged in at any time. The mobile device has not been allocated to some Grid user by XtreemOS, so we find no need for accounting in this scenario.

Needs for some form of isolation appear when the mobile device offers services to grid users. Resource usage enforcement (and accounting) is necessary for predefined services. User isolation as in the standard XtreemOS flavour becomes necessary if the device runs user-defined codes (application).

Mobile devices have limited capabilities. A Linux kernel compiled with support for control groups and various namespaces might need more CPU cycles and memory. The processors used in some mobile devices provide minimal protection of the hardware (registers, memory), which may prevent isolation of user processes. Running virtual machines is even more problematic as these processors do not have any support for hardware virtualization.

Chapter 6

Conclusion

Recent developments in the Linux kernel have introduced various mechanisms to support secure and controlled sharing of computation platforms in the context of grid and cloud computing. Control groups allow to account, limit and protect various subsystem (CPU, memory, disk, network) usage by group of tasks. It is possible, in XtreemOS, to run each user application inside its own control group. This solution allows to limit the impact of the whole application on its environment and to bill for resource usage. The support for namespace management recently introduced in Linux allows to manage the view of user applications on their environment. Running user applications inside their own private namespace provides confidentiality and security (user application don't see other application processes or files) and eases the implementation of checkpointing and application migration for load balancing. The use of namespaces also helps in the implementation of stable development and execution environments. For instance, a resource can support a filesystem tree per VO and run each application inside the filesystem namespace provided by the VO in charge of its execution. Finally, the requirements concerning resource management, security, and execution environments can be fulfilled using virtual machines. It is possible to run XtreemOS on the raw hardware and to run user applications inside virtual machines provided by virtual organisations. This solution provides full isolation of the applications. The virtual machines do not need to be XtreemOS-based but can still provide access to XtreemOS features such as XtreemFS through virtual devices. On the order hand a virtualization platform (not based on XtreemOS) can run XtreemOS inside virtual machines. This solution also fulfills the initial requirements.

These solution based mainly on Linux kernel mechanisms may necessitate some hardware support, mainly for virtualization, and may require more processor and memory resources. This may limit their use on mobile device flavours of XtreemOS. But, as long as mobile devices are not in charge of executing user-defined applications, the need for isolation is not critical. These new mechanisms

recently introduced in the Linux kernel have not yet been ported to the LinuxSSI flavour of XtreemOS: no system-wide management of control groups and namespaces. Basing XtreemOS isolation mechanisms on control groups and private namespaces implies some development of LinuxSSI.

Bibliography

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles, SOSP'03*, October 2003. Available at `http://citeseer.ist.psu.edu/dragovic03xen.html`.

[2] Fabrice Bellard. *QEMU, a Fast and Portable Dynamic Translator*. Technical report, USENIX Association, 2005.

[3] Fabrice Bellard. *QEMU: Open Source Processor Emulator*, 2008. Available at `http://http://bellard.org/qemu/`.

[4] Daniel P. Berrange. *An introduction to libvirt's LXC (LinuX Container) support*, 2008. Available at `https://lists.linux-foundation.org/pipermail/containers/2008-September/013237.html`.

[5] Eric W. Biederman. *An introduction and A path for merging network namespace work*, 2007. Available at `http://lwn.net/Articles/219597/`.

[6] *The CFS scheduler*, 2008. Available at `http://www.mjmwired.net/kernel/Documentation/scheduler/sched-design-CFS.txt`.

[7] *Control Groupstats*, 2008. Available at `http://www.mjmwired.net/kernel/Documentation/accounting/cgroupstats.txt`.

[8] Ken Chen. *Cpuacct Cgroup*, 2008. Available at `http://lkml.org/lkml/2008/12/3/635`.

[9] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX*

Symposium on Networked Systems Design and Implementation, 2005. Available at http://www.cl.cam.ac.uk/netos/papers/2005-migration-nsdi-pre.pdf.

[10] Jonathan Corbet. *Documentation/sharedsubtree.txt*, 2005. Available at http://lwn.net/Articles/159092/.

[11] Jonathan Corbet. *Controlling memory use in containers*, 2007. Available at http://lwn.net/Articles/243795/.

[12] Jonathan Corbet. *Network namespaces*, 2007. Available at http://lwn.net/Articles/219794/.

[13] Jonathan Corbet. *Process containers*, 2007. Available at http://lwn.net/Articles/236038/.

[14] Jonathan Corbet. *System call updates: indirect(), timerfd(), and hijack()*, 2007. Available at http://lwn.net/Articles/260172/.

[15] *CPUSETS*, 2008. Available at http://www.mjmwired.net/kernel/Documentation/cpusets.txt.

[16] Janak Desai, Chad Sellers, and Steve Grubb. *pam_namespace - setup a private namespace*. Available at http://www.kernel.org/pub/linux/libs/pam/Linux-PAM-html/sag-pam_namespace.html.

[17] Jake Edge. *Smack for simplified access control*, 2007. Available at http://lwn.net/Articles/244531/.

[18] Pavel Emelyanov and Kir Kolyshkin. *PID namespaces in the 2.6.24 kernel*, 2007. Available at http://lwn.net/Articles/259217/.

[19] Jérôme Gallard, Geoffroy Vallée, Adrien Lebre, Christine Morin, Pascal Gallard, and Stephen L. Scott. Complementarity between virtualization and single system image technologies. In *VHPC 2008, 3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing*, Las Palmas de Gran Canaria, Canary Island, Spain, 2008. Held in conjunction with Europar 2008, Springer LNCS.

[20] Cedric Le Goater. *User namespaces: add unshare*, 2008. Available at http://lwn.net/Articles/237662/.

[21] Robert P. Goldberg. Architecture of Virtual Machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, Cambridge, MA, USA, March 26-27, 1973.

[22] Serge E. Hallyn. *uts namespaces: Introduction*, 2006. Available at `http://lwn.net/Articles/179345/`.

[23] Serge E. Hallyn. *User namespaces: introduction*, 2008. Available at `https://lists.linux-foundation.org/pipermail/containers/2008-August/012675.html`.

[24] Serge E. Hallyn and Ram Pai. *Applying mount namespaces*, 2007. Available at `http://www.ibm.com/developerworks/linux/library/l-mount-namespaces.html`.

[25] Dave Hansen. *Read-only bind mounts*, 2007. Available at `http://lwn.net/Articles/250206/`.

[26] Matt Helsley. *Container Freezer v6: Reuse Suspend Freezer*, 2008. Available at `https://lists.linux-foundation.org/pipermail/containers/2008-August/012376.html`, justifications at `http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-08/msg06889.html`.

[27] Matt Helsley. *Container Freezer v6: Reuse Suspend Freezer*, 2008. Available at `http://lwn.net/Articles/293642/`.

[28] *Linux 2.6.25*, 2008. Available at `http://kernelnewbies.org/Linux_2_6_25`.

[29] Kirill Korotaev. *IPC namespace*, 2006. Available at `http://lwn.net/Articles/187274/`.

[30] Paul Menage. *Linux documentation: Control Groups*, 2008. Available at `http://www.mjmwired.net/kernel/Documentation/cgroups.txt`.

[31] *Linux Containers - Network Namespace*. Available at `http://lxc.sourceforge.net/network.php`.

[32] Daisuke Nishimura. *cgroup swap subsystem*, 2008. Available at `https://lists.linux-foundation.org/pipermail/containers/2008-March/010216.html`.

[33] OpenVZ. *OpenVZ welcome page*, 2007. Available at http://wiki.openvz.org/Main_Page.

[34] Andrea Righi. *io-throttle controller documentation*, 2008. Available at `http://www.uwsg.indiana.edu/hypermail/linux/kernel/0805.3/0091.html`.

[35] Casey Schaufler. *Smack: Simplified Mandatory Access Control Kernel*, 2007. Available at `http://lwn.net/Articles/252378/`.

[36] Ryo Tsuruta. *bio-cgroup: Introduction*, 2008. Available at `http://lwn.net/Articles/299731/`.

[37] Ryo Tsuruta. *I/O bandwidth controller and BIO tracking*, 2008. Available at `http://lwn.net/Articles/300191/`.

[38] Satoshi Uchida. *Yet another I/O bandwidth controlling subsystem for CGroups based on CFQ*, 2008. Available at `http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-04/msg00146.html`.

[39] *Linux Containers - Utsname Namespace*. Available at `http://lxc.sourceforge.net/utsname.php`.

[40] VMware. *VMware Welcome Page*, 2007. Available at http://www.vmware.com.

[41] Rafael J. Wysocki. *swsusp: freeze user space processes first*, 2006. Available at `http://lwn.net/Articles/170717/`.

[42] XtreemOS Consortium. *Application References, Requirements, Use Cases and Experiments*. Deliverable D.4.2.3, July 2007.

[43] XtreemOS Consortium. *Reproducible Evaluation of Distributed Servers*. Deliverable D.3.2.6, November 2008.