VServer Control Daemon Reference Manual

Benedikt Böhm <hollow@gentoo.org>
Luca Longinotti <chtekk@gentoo.org>

March 3, 2007

Contents

Preface iv
Audience . . . iv
Organization of This Book . . . iv
Conventions Used in This Book . . . iv
Acknowledgements . . . v
Feedback . . . v

I Introduction 1

1 Introduction to virtualization 2
1.1. What is virtualization? . . . 2
1.2. Simulation . . . 2
1.3. Emulation . . . 3
1.4. Simulation vs. Emulation . . . 3
1.5. Native Virtualization . . . 5
1.6. Para-virtualization . . . 5
1.7. Operating System-Level Virtualization . . . 6

2 The Linux-VServer Project 8
2.1. Rationale . . . 8
2.2. The Concept . . . 8
2.3. Usage Scenarios . . . 9
2.4. Features . . . 13
2.5. History . . . 26

3 The VServer Control Daemon 27
3.1. Abstract . . . 27
3.2. Rationale . . . 27
3.3. Architecture . . . 27
3.4. Linux-VServer system call library . . . 28
3.5. VServer Control Daemon . . . 30
3.6. Command Line Wrappers . . . 36
3.7. VServer Statistics Daemon . . . 36

II Installation 37

III Configuration and Maintenance 38

IV Command Reference 39

4 VServer Control Client 40

5 VServer Control Daemon 41

6 VServer Control Daemon Administration Tool 42

7 VServer Configuration Editor 43

8 VServer Kernel Helper 44

V XML-RPC Method Reference 45

9 Introduction 46
9.1. XML-RPC Signatures 46
9.2. Performing XML-RPC Requests 47
9.3. Method Error Codes 51

10 Helper Methods 52
10.1. helper.netup 52
10.2. helper.restart 52
10.3. helper.shutdown 53
10.4. helper.startup 54

11 Daemon Administration Methods 55
11.1. vcd.login 55
11.2. vcd.status 55
11.3. vcd.user.caps.add 56
11.4. vcd.user.caps.get 56
11.5. vcd.user.caps.remove 57
11.6. vcd.user.get 58
11.7. vcd.user.remove 58
11.8. vcd.user.set 59

12 General Maintenance Methods 60
12.1. vx.create 60
12.2. vx.exec 61
12.3. vx.kill 61
12.4. vx.load 62
12.5. vx.reboot 63
12.6. vx.remove 64
12.7. vx.rename 64
12.8. vx.restart 65
12.9. vx.start 66
12.10. vx.status 66
12.11. vx.stop 67

13 Database Manipulation Methods **69**
13.1. vxdb.dx.limit.get . . . 69
13.2. vxdb.dx.limit.remove . . . 69
13.3. vxdb.dx.limit.set . . . 70
13.4. vxdb.init.get . . . 71
13.5. vxdb.init.set . . . 72
13.6. vxdb.list . . . 72
13.7. vxdb.mount.get . . . 73
13.8. vxdb.mount.remove . . . 74
13.9. vxdb.mount.set . . . 75
13.10. vxdb.name.get . . . 75
13.11. vxdb.nx.addr.get . . . 76
13.12. vxdb.nx.addr.remove . . . 77
13.13. vxdb.nx.addr.set . . . 78
13.14. vxdb.nx.broadcast.get . . . 78
13.15. vxdb.nx.broadcast.remove . . . 79
13.16. vxdb.nx.broadcast.set . . . 79
13.17. vxdb.owner.add . . . 80
13.18. vxdb.owner.get . . . 81
13.19. vxdb.owner.remove . . . 81
13.20. vxdb.vdir.get . . . 82
13.21. vxdb.vx.bcaps.add . . . 83
13.22. vxdb.vx.bcaps.get . . . 83
13.23. vxdb.vx.bcaps.remove . . . 84
13.24. vxdb.vx.ccaps.add . . . 85
13.25. vxdb.vx.ccaps.get . . . 85
13.26. vxdb.vx.ccaps.remove . . . 86
13.27. vxdb.vx.flags.add . . . 86
13.28. vxdb.vx.flags.get . . . 87
13.29. vxdb.vx.flags.remove . . . 88
13.30. vxdb.vx.limit.get . . . 88
13.31. vxdb.vx.limit.remove . . . 89
13.32. vxdb.vx.limit.set . . . 90
13.33. vxdb.vx.sched.get . . . 91
13.34. vxdb.vx.sched.remove . . . 92
13.35. vxdb.vx.sched.set . . . 92
13.36. vxdb.vx.uname.get . . . 93
13.37. vxdb.vx.uname.remove . . . 94
13.38. vxdb.vx.uname.set . . . 95
13.39. vxdb.xid.get . . . 95

Preface

Audience

This book is written for computer-literate folk who want to use the Linux-VServer technology to seperate their processes (applications) into distinct execution units for one or more of the following reasons:

1. Administrative Seperation
2. Service Seperation
3. Enhanced Security
4. Easy Maintenance
5. Fail-over Scenarios
6. Development and Testing

Most readers are probably system administrators who need to seperate their applications or programmers who want to test their programs in many different distributions or configurations.

Although this book is written to cover the whole extend of the Linux-VServer technology, it is advisable to have basic knowledge about the Linux operating system, its shell and commands as well as your distribution of choice with all its peculiarities.

Organization of This Book

Conventions Used in This Book

italic An italic font is used for filenames, URLs, emphasized text, and the first usage of technical terms.

`monospace` A monospaced font is used for error messages, commands, environment variables, names of ports, hostnames, user names, group names, device names, variables, and code fragments.

bold A bold font is used for applications, commands, and keys.

Acknowledgements

Feedback

If you found a typo in this manual, or if you have thought of a way to make this guide better, feel free to contact the authors!

If you have suggestions for improving this manual, try to be as specific as possible when formulating it. If you have found an error, please include the chapter/section/subsection name and some of the surrounding text so we can fine it easily.

Please submit a report by e-mail to one of the addresses mentioned above.

Part I.

Introduction

1. Introduction to virtualization

Today, virtualization is a broad term that refers to the abstraction of computer resources. It has been widely used since the 1960s or earlier, and has been applied to many different aspects and scopes of computing – from entire computer systems to individual capabilities or components.

1.1. What is virtualization?

> Virtualization is, at its foundation, a technique for hiding the physical characteristics of computing resources from the way in which other systems, applications, or end users interact with those resources. This includes making a single physical resource (such as a server, an operating system, an application, or storage device) appear to function as multiple logical resources; or it can include making multiple physical resources (such as storage devices or servers) appear as a single logical resource. [1]

> Virtualization is a framework or methodology of dividing the resources of a computer into multiple execution environments, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others. [2]

Due to the abstract nature of these definitions the list of virtualization types, implementations and frameworks is rather huge, and we trust that readers will appreciate that this subject cannot be covered in great detail here. Nevertheless the following list encompasses some popular virtualization techniques, to give the reader – at least – a rough understanding of key differences between current major technologies.

A rather complete list of virtualization frameworks can be found on Amit Singhs page *An Introduction to Virtualization*. [2]

1.2. Simulation

A *computer simulation* is an attempt to model a real-life situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behaviour of the system.

An interesting application of computer simulation is to simulate computers using computers. The related software is called computer architecture simulators, which can be further divided into instruction set simulators or full system simulators.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

An instruction set simulator is often provided with a debugger in order for a software engineer to debug the program prior to obtaining target hardware. GDB is one of the debuggers which have compiled-in ISS. It is sometimes integrated with simulated peripheral circuits such as timers, interrupts, serial port, general I/O port, etc to mimic the behavior of microcontrollers. [3]

1.3. Emulation

A *software emulator* allows computer programs to run on a platform (computer architecture and/or operating system) other than the one for which they were originally written. More generally, emulation refers to the ability of a program or device to imitate another program or device.

Many printers, for example, are designed to emulate Hewlett-Packard LaserJet printers because so much software is written for HP printers. By emulating an HP printer, a printer can work with any software written for a real HP printer. Emulation tricks the software into believing that a device is really some other device.

Another popular use of emulators is to mimic the experience of running arcade games or console games on personal computers. Emulating these on modern desktop computers is usually less cumbersome and more reliable than relying on the original machines, which are often old and hard to find, let alone repair, though emulation of arcade and console systems usually includes the practice of illegally downloading software from various electronic distribution sources.

In a theoretical sense, the Church-Turing thesis [4] implies that any operating environment can be emulated within any other. In practice, however, it can be quite difficult, particularly when the exact behavior of the system to be emulated is not documented and has to be deduced through reverse engineering. It also says nothing about timing constraints – if the emulator does not perform as quickly as the original hardware, the emulated software may run much more slowly than it would have on the original hardware.

1.4. Simulation vs. Emulation

There is a lot of confusion between emulators and simulators, in fact, the distinction between those two is not always easy to tell. Ed Thelen [5] has collected some opinions on the difference, some of which are quoted below.

In October 2005, Peter Hans van den Muijzenberg wrote:

```
Hi,

Regarding the difference between simulation and emulation:
Not limited to computers I use this distinction:
```

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

```
- A simulation mimics the outward appearance
- An emulation mimics the cause/process.

If you want to convince people that watching television gives you
stomach-aches, you can simulate this by holding your chest/abdomen and
moan. You can emulate it by eating a kilo of unripe apples.

BFN,
                                Peter Hans van den Muijzenberg
```

Another, quite theoretical view of the problem, from Arun Parajuli, January 2006:

```
In my view the difference between simulation and emulation is:

Emulation: A system X is said to emulate another system Y if the
behaviour of X is exactly the same as that of Y (same out put for
same input under similar conditions) but the mechanism to arrive at
the output (from the input) is different.  Emulation is generally
used when we don't exactly know the internal mechanism of the
original system but are familiar with the input/output pattern. For
example, neural networks may be used to emulate different systems.
Neural networks are trained to produce the same output for the same
input as that of the original system though the mechanism/procedure
to generate the output are quite different.

Simulation: A system X is said to simulate another system Y
when the internal mechanism/procedures of X is a mathematical
(or any other) model known to best represent the actual mechanism
of Y. Simulation is generally used when we have some mathematical
models of the original system (Y) and want to know output for a
given set of inputs. For example we may have a good mathematical
model of effect of water temperature on the hurricane formation. We
use that system to predict the nature of hurricanes for different
water temperatures. It is to be noted that the result of simulation
can sometimes be unverifiable. For example the hurricane pattern
suggested by the simulator for water temperature equal to 50 degree
Celsius will be difficult to verify because we may never encounter
such situation.

Arun Parajuli
```

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

1.5. Native Virtualization

Native virtualization, in which the virtual machine monitor simulates complete hardware to allow operation of an umodified operating system for the same type of CPU to execute within the virtual machine in complete isolation.

Figure 1.1.: Native Virtualization

Native virtualization leverages hardware-assisted capabilities available in the latest processors from Intel (Intel VT) and Advanced Micro Devices (AMD-V) to provide near-native performance. Prior to these processors, the x86 architecture did not meet some fundamental requirements for virtualization, making it difficult to implement a virtual machine monitor for this type of processor.

These requirements include: equivalence – a program running in the virtual machine should exhibit a behavior essentially identical to the original physical machine; resource control – the virtual machine monitor must be in complete control of the virtualized resources and efficiency while workload performance should not be degraded. [6]

1.6. Para-virtualization

However, native virtualization may incur a performance penalty. The virtual machine monitor must provide the guest system with an image of an entire system, including virtual BIOS, virtual memory space, and virtual devices. The virtual machine monitor also must create and maintain data structures for the virtual components, such as a shadow memory page table. These data structures must be updated for every corresponding access by the guest system.

Therefore, para-virtualization exposes a virtual architecture that is slightly different than the physical architecture. The differences in the architecture are driven by improvements

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Figure 1.2.: Para-Virtualization

in scalability or reductions in system complexity. Modifying the architecture breaks backwards compatibility with existing OS code, which is a major disadvantage. However, it enables to co-design the virtual architecture with the operating system, which gives a considerable latitude when exploring issues of scale.

Porting an operating system to run on the VMM is similar to supporting a new hardware platform, however the process is simplified because the para-virtual machine architecture is very similar to the underlying native hardware.

Para-virtualization has been used in previous VMMs, including VM/370 and Disco. These systems added a combination of instructions, registers, or devices to the virtual architecture to improve performance. However, because the goal of these systems was to run legacy OSs, the use of para-virtualization was minimized. [7]

The term "para-virtualization" was first used in the research literature in association with the Denali virtual machine monitor [7]. The term is also used to describe the Xen, L4, Virtual, Iron and TRANGO hypervisors. All these projects use paravirtualization techniques to support high performance virtual machines on x86 hardware.

1.7. Operating System-Level Virtualization

Operating System-Level Virtualization is a server virtualization technology which virtualizes servers on a operating system (kernel) layer. It can be thought of as partitioning a single physical server into multiple small computational partitions. Each such partition looks and feels like a real server, from the point of view of its owner. On Unix systems, this technology can be thought of as an advanced extension of the standard chroot mechanism.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

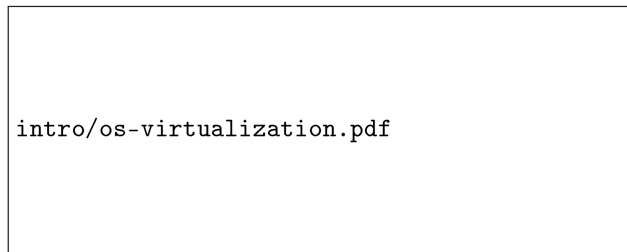

Figure 1.3.: Operating System-Level Virtualization

Many terms for the computational partitions exist and mostly depend on the implementation: virtual environments (VE), virtual private servers (VPS), jails, guests, zones, vservers, containers, etc.

The operating system level architecture has low overhead that helps to maximize efficient use of server resources. Due to a single-kernel approach, this type of virtualization introduces only a negligible overhead and allows running hundreds of virtual private servers on a single physical server. In contrast, approaches such as emulation and paravirtualization cannot achieve such level of density, due to overhead of running multiple kernels. On the other hand, operating system-level virtualization does not allow running different operating systems (i.e. different kernels), although different libraries, distributions etc. are possible. [8]

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

2. The Linux-VServer Project

intro/vserver-lo

The Linux-VServer project implements operating system-level virtualization based on *Security Contexts* which permit the creation of many independent *Virtual Private Servers* (VPS) that run simultaneously on a single physical server at full speed, efficiently sharing hardware resources.

A VPS provides an almost identical operating environment as a conventional Linux server. All services, such as ssh, mail, web and databases, can be started on such a VPS, without (or in special cases with only minimal) modification, just like on any real server.

Each virtual server has its own user account database and root password and is isolated from other virtual servers, except for the fact that they share the same hardware resources.

2.1. Rationale

Over the years, computers have become sufficiently powerful to use virtualization to create the illusion of many smaller virtual machines, each running a separate operating system instance.

Most virtual machines accomplish what they do by emulating some real or fictional hardware, which in turn consume real resources from the host system (the machine running the virtual machines). This approach, used by most system emulators, allows the emulator to run an arbitrary operating systems, even for a different hardware architecture. No modifications need to be made to the guest system (the operating system running in the virtual machine) because it isn't aware of the fact that it isn't running on real hardware.

Some system emulators require small modifications or specialized drivers to be added to host or guest to improve performance and minimize the overhead required for hardware emulation. Although this significantly improves efficiency, there are still large amounts of resources being wasted in caches and mediation between virtual machine and monitor.

But suppose you do not want to run many different operating systems simultaneously on a single box. Most applications running on a server do not require hardware access or kernel level code, and could easily share a machine with others, if they could be separated and secured...

2.2. The Concept

At a basic level, a Linux server consists of three building blocks: hardware, kernel and applications. The hardware usually depends on the provider or system maintainer, and, while it has a big influence on the overall performance, it cannot be changed that easily, and will likely differ from one setup to another.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

The main purpose of the kernel is to build an abstraction layer on top of the hardware to allow processes (applications) to work with and operate on resources without knowing the details of the underlying hardware. Ideally, those processes would be completely hardware agnostic, by being written in an interpreted language and therefore not requiring any hardware-specific knowledge.

Given that a system has enough resources to drive ten times the number of applications a single Linux server would usually require, why not put ten servers on that box, which will then share the available resources in an efficient manner?

Most server applications (e.g. httpd) will assume that it is the only application providing a particular service, and usually will also assume a certain filesystem layout and environment. This dictates that similar or identical services running on the same physical server, but for example, only differing in their addresses, have to be coordinated. This typically requires a great deal of administrative work which can lead to reduced system stability and security.

The basic concept of the Linux-VServer solution is to separate the user-space environment into distinct units (Virtual Private Servers) in such a way that each VPS looks and feels like a real server to the processes contained within.

Although different Linux distributions use (sometimes heavily) patched kernels to provide special support for unusual hardware or extra functionality, most Linux distributions are not tied to a special kernel.

Linux-VServer uses this fact to allow several distributions, to be run simultaneously on a single, shared kernel, without direct access to the hardware, and share the resources in a very efficient way.

2.3. Usage Scenarios

2.3.1. Administrative Separation

As the hardware evolves, it is tempting to put more and more tasks on a server. Though Linux could reliably handle it, at some point, the system administrator will likely end up with too much stuff and users on one system and worrying about system updates. Additionally, separating different or similar services which otherwise would interfere with each other, either because they are poorly designed or because they are simply incapable of peaceful coexistence for whatever reason, may often be complex or even impossible.

The Linux-VServer project addresses this issue. The same box is able to run multiple virtual servers and each one does the job it is supposed to do. If the administrator needs to upgrade to PHP 5 for a given project, he can update just one virtual server and does not affect the others.

Also, the root password of a virtual servers can be given to foreign administrators, thus allows him to perform updates, restart services or update system configuration without

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

having to know or worry about other virtual servers hosted on the same machine. This allows a clever provider to sell Virtual Private Servers, which uses less resources than other virtualization techniques, which in turn allows to put more units on a single machine.

The list of providers doing so is relatively long, and so this is rightfully considered the main area of application. See the Linux-VServer project page[1] for a (possibly incomplete) list of companies providing Virtual Private Servers based on the Linux-VServer technology.

2.3.2. Enhancing Security

While it can be interesting to run several virtual servers in one box, there is one concept potentially more generally useful. Imagine a physical server running a single virtual server. The goal is to isolate the main environment from any service, any network. You boot in the main environment, start very few services and then continue in the virtual server.

The service in the main environment would be:

- Unreachable from the network.
- Able to log messages from the virtual server in a secure way. The virtual server would be unable to change/erase the logs. Even a cracked virtual server would not be able the edit the log.
- Able to run intrusion detection facilities, potentially spying the state of the virtual server without being accessible or noticed. For example, tripwire could run there and it would be impossible to circumvent its operation or trick it.

Another option is to put the firewall in a virtual server, and pull in the DMZ, containing each service in a separate VPS. On proper configuration, this setup can reduce the number of required machines drastically, without impacting performance.

2.3.3. Resource Indepedence

Since virtual servers are only guests on the hardware they are using, they are not aware of the specifics: they do not contain disk configurations, kernels or network configurations.

One key feature of a virtual server is the independence from the actual hardware. Most hardware issues are irrelevant for a virtual server installation.

The main server acts as a host and takes care of all the details. The virtual server is just a client and ignores all the details. As such, the client can be moved to another physical server with very few manipulations.

[1] `http://linux-vserver.org/VServer_Hosting`

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

For example, to move the virtual server from one physical computer to another, it sufficient to do the following:

- shutdown the running virtual server
- copy it over to the other machine
- copy the configuration
- start the virtual server on the new machine

No adjustments to user setup, password database or hardware configuration are required, as long as both machine architectures are binary compatible.

Thus, once a virtual server is using more resource than expected, the administrator can easily move it to another machine without the need to worry about configuration files for disk layout, network interfaces etc. A virtual server is just a directory on the filesystem of host system.

2.3.4. Distribution Independence

People are often talking about their preferred distribution. Should one use Fedora, Debian or something else? Should one give a spin to the latest and greatest distribution just for the sake of it?

With virtual servers, the choice of a distribution is less important. When you select a distribution, you expect it will do the following:

- Good hardware support/detection
- Good package technology/updates
- Good package selection
- Reliable packages

The choice is important because every service running on a box will be using the same distribution. Most distributions out there are good and reliable. Still each one has its peculiarities and probably flaws. For example, one distribution is doing a great job on security but is not delivering the latest and greatest PHP. Now because you have decided to use this distribution for some projects, using virtual servers does not prevent you from using another distribution for other projects or even a second virtual server for existing projects.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

2.3.5. Fail-over Scenarios

Pushing the limit a little further, replication technology could be used to keep an up-to-the-minute copy of the filesystem of a running virtual server. This would permit a very fast fail-over if the running server goes offline for whatever reason.

All the known methods to accomplish this, starting with network replication via rsync, or drbd, via network devices, or shared disk arrays, to distributed filesystems, can be utilized to reduce the down-time and improve overall efficiency.

2.3.6. Experimenting and Upgrading

If the system administrator intends to upgrade a system to get new features or security updates, he probably first wants to test new packages on a development machine, before the production server can be updated. Under normal circumstances, i.e. all test have been passed, the upgrade procedure will look something like this:

- Doing a backup of the server
- Perform all the upgrades and install the new applications

Two hours later something does not work as expected and – to make it even worse – it works fine on the development machine. Every system administrator has experienced this scenario at least once.

Another solution to this problem would be to install the new production server on new hardware, but this is not always possible, due to lack of hardware or the effort needed to clone an existing machine.

Using virtual servers, all this becomes very easy:

- Stop the virtual server in production
- Make a copy of the virtual server
- Perform the upgrades in the new virtual server

To get back to the example from above, two hours later something does not work as expected and there is no immediate fix for the problem.

Again, using virtual servers, the (temporary) solution to this problem is very easy:

- Stop the new virtual server and assign it a new IP address
- Start both the old and new virtual server

Now the old one is still online and the issue can be tracked down on your new virtual server using a different IP address. After the problem has been fixed the new virtual server will be reassign the old IP address and serve for production.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

2.3.7. Development and Testing

Consider a software tool or package which should be built for several versions of a specific distribution (Mandrake 8.2, 9.0, 9.1, 9.2, 10.0) or even for different distributions.

This is easily solved with Linux-VServer. Given plenty of disk space, the different distributions can be installed and running side by side, simplifying the task of switching from one to another.

Of course this can be accomplished by chroot() alone, but with Linux-VServer it's a much more realistic simulation.

2.4. Features

Recent Linux Kernels already provide many security features that are utilized by Linux-VServer to do its work. Especially features such as the Linux Capability System, Resource Limits, File Attributes and the Change Root Environment. The following sections will give a short overview about each of these and the additional features implemented by the Linux-VServer project.

2.4.1. Context Seperation

The separation mentioned in section 2.2 requires some modifications to the kernel to allow for the notion of contexts. The purpose of this "context" is to hide all processes outside of its scope, and prohibit any unwanted interaction between a process inside the context and a process belonging to another context.

This separation requires the extension of some existing data structures in order for them to become aware of contexts and to differentiate between identical uids used in different virtual servers.

It also requires the definition of a default context that is used when the host system is booted, and to work around the issues resulting from some false assumptions made by some user-space tools (like pstree) that the init process has to exist and to be running under id '1'.

To simplify administration, the host context is not treated differently than any other context as far as process isolation is concerned. To allow for a process overview, a special spectator context has been defined to peek at all processes at once.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

2.4.2. Network Seperation

While context separation is sufficient to isolate groups of processes, a different kind of separation, or rather a limitation, is required to confine processes to a subset of available network addresses.

Several issues have to be considered when doing so – for example, the fact that bindings to special addresses like `INADDR_ANY` or the local host address have to be handled in a very special way.

Currently, the Linux-VServer kernel does not make use of virtual network devices (and maybe never will) to minimize the resulting overhead. Therefore socket binding and packet transmission have been adjusted.

2.4.3. Capabilities and Flags

A capability is a token used by a process to prove that it is allowed to perform an operation on an object. The Linux Capability System is based on "POSIX Capabilities", a somewhat different concept, designed to split up the all powerful root privilege into a set of distinct privileges.

POSIX Capabilities

A process has three sets of bitmaps called the inheritable(I), permitted(P), and effective(E) capabilities. Each capability is implemented as a bit in each of these bitmaps that is either set or unset.

When a process tries to do a privileged operation, the operating system will check the appropriate bit in the effective set of the process (instead of checking whether the effective uid of the process is 0 as is normally done).

For example, when a process tries to set the clock, the Linux kernel will check that the process has the `CAP_SYS_TIME` bit (which is currently bit 25) set in its effective set.

The permitted set of the process indicates the capabilities the process can use. The process can have capabilities set in the permitted set that are not in the effective set.

This indicates that the process has temporarily disabled this capability. A process is allowed to set a bit in its effective set only if it is available in the permitted set. The distinction between effective and permitted exists so that processes can "bracket" operations that need privilege.

The inheritable capabilities are the capabilities of the current process that should be inherited by a program executed by the current process. The permitted set of a process is masked against the inheritable set during `exec()`. Nothing special happens during `fork()` or `clone()`. Child processes and threads are given an exact copy of the capabilities of the parent process.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

The implementation in Linux stopped at this point, whereas POSIX Capabilities require the addition of capability sets to files too, to replace the SUID flag (at least for executables).

Upper Bound for Capabilities

Because the current Linux Capability system does not implement the filesystem related portions of POSIX Capabilities which would make setuid and setgid executables secure, and because it is much safer to have a secure upper bound for all processes within a context, an additional per-context capability mask has been added to limit all processes belonging to that context to this mask. The meaning of the individual caps (bits) of the capability bound mask is exactly the same as with the permitted capability set.

Context Capabilities

As the Linux capabilities have almost reached the maximum number that is possible without heavy modifications to the kernel, it was a natural step to add a context-specific capability system.

The Linux-VServer context capability set acts as a mechanism to fine tune existing Linux capabilities. It is not visible to the processes within a context, as they would not know how to modify or verify it.

In general there are two ways to use those capabilities:

- Require one or a number of context capabilities to be set in addition to a given Linux capability, each one controlling a distinct part of the functionality. For example the `CAP_NET_ADMIN` could be split into RAW and PACKET sockets, so you could take away each of them separately by not providing the required context capability.
- Consider the context capability sufficient for a specified functionality, even if the Linux Capability says something different. For example the `mount` system call requires `CAP_SYS_ADMIN` which adds a dozen other things we do not want, so we define `VXC_SECURE_MOUNT` to allow mounts for certain contexts.

The difference between the context flags and the context capabilities is more an abstract logical separation than a functional one, because they are handled very similar.

2.4.4. Resource Isolation

Most resources are somewhat shared among the different contexts. Some require more additional isolation than others, either to avoid security issues or to allow for improved accounting.

Those resources are:

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- shared memory
- inter-process communication
- user and process IDs
- filesystem tagging
- pseudo terminals
- netowork sockets

2.4.5. Resource Accounting

Some runtime properties of a context are useful to the administrator, either for keeping an overview of used resources, to get a feeling for the capacity of the host, or for billing them in some way to a customer.

There are two different kinds of accountable properties, those having a current value which represents the state of the system (for example the speed of a vehicle), and those which monotonically increase over time (like the mileage).

Most of the state type of properties also qualify for applying some limits, so they are handled specially and are described in more detail in the following section.

Popular candidates for context accounting include:

- amount of CPU time spent
- number of forks done
- socket messages by type
- network packets transmitted and received

2.4.6. Resource Limits

Most properties related to system resources, might it be the memory consumption, the number of processes or file-handles, qualify for imposing limits on them.

The Linux kernel provides the `getrlimit` and `setrlimit` system calls to get and set resource limits per process. Each resource has an associated soft and hard limit. The soft limit is the value that the kernel enforces for the corresponding resource. The hard limit acts as a ceiling for the soft limit: an unprivileged process may only set its soft limit to a value in the range from 0 up to the hard limit, and (irreversibly) lower its hard limit. A privileged process (one with the `CAP_SYS_RESOURCE` capability) may make arbitrary changes to either limit value.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

The Linux-VServer kernel extends this system to provide resource limits for whole contexts, not just single processes. Additionally a few new limit types missing in the vanilla kernel were introduced.

Additionally the context limit system keeps track of observed maxima and resource limit hits, to provide some feedback for the administrator.

2.4.7. CPU Scheduler

It is important to have a decent understanding of both processes and threads before learning about schedulers, though explaining processes and threads in depth is outside of the scope of this document, thus only a summary of the things that one must know about them is provided here. Readers are encouraged to gain an in-depth understanding of processes and threads first. Excellent sources are listed in the bibliography [**?**, **?**, **?**].

Programs and Processes

A program is a combination of instructions and data put together to perform a task when executed. A process is an instance of a program (what one might call a "running" program). An analogy is that programs are like classes in languages like C++ and Java, and processes are like objects (instantiated instances of classes). Processes are an abstraction created to embody the state of a program during its execution. This means keeping track of the data that is associated with a thread or threads of execution, which includes variables, hardware state (e.g. registers and the program counter, etc...), and the contents of an address space.

Threads

A process can have multiple threads of execution that work together to accomplish its goals. These threads of execution are aptly named threads. A kernel must keep track of each thread's stack and hardware state, or whatever is necessary to track a single flow of execution within a process. Usually threads share address spaces, but they do not have to (often they merely overlap). It is important to remember that only one thread may be executing on a CPU at any given time, which is basically the reason kernels have CPU schedulers. An example of multiple threads within a process can be found in most web browsers. Usually at least one thread exists to handle user interface events (like stopping a page load), one thread exists to handle network transactions, and one thread exists to render web pages.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Scheduling in Linux

Multitasking kernels (like Linux) allow more than one process to exist at any given time, and furthermore each process is allowed to run as if it were the only process on the system. Processes do not need to be aware of any other processes unless they are explicitly designed to be. This makes programs easier to develop, maintain, and port.

Though each CPU in a system can execute only one thread within a process at a time, many threads from many processes appear to be executing at the same time. This is because threads are scheduled to run for very short periods of time and then other threads are given a chance to run. A kernel's scheduler enforces a thread scheduling policy, including when, for how long, and in some cases where (on Symmetric Multiprocessing (SMP) systems) threads can execute.

Normally the scheduler runs in its own thread, which is woken up by a timer interrupt. Otherwise it is invoked via a system call or another kernel thread that wishes to yield the CPU. A thread will be allowed to execute for a certain amount of time, then a context switch to the scheduler thread will occur, followed by another context switch to a thread of the scheduler's choice. This cycle continues, and in this way a certain policy for CPU usage is carried out.

Token Bucket Extension

While the basic idea of Linux-VServer is a peaceful coexistence of all contexts, sharing the common resources in a respectful way, it is sometimes useful to control the resource distribution for resource hungry processes.

The basic principle of a *token bucket* is not very new. It is given here as an example for the CPU scheduler. The same principle also applies to other scheduler priorities, network bandwidth limitation and resource control in general.

The Linux-VServer scheduler uses this mechanism in the following way: consider a bucket of a certain size S which is filled with a specified amount of tokens R every interval T, until the bucket is "full" – excess tokens are spilled. At each timer tick, a running process (here running means actually needing the CPU as opposed to "running" as in "existing") consumes exactly one token from the bucket, unless the bucket is empty, in which case the process is put on a hold queue until the bucket has been refilled with a minimum M of tokens. The process is then rescheduled.

A major advantage of a token bucket is that a certain amount of tokens can be accumulated in times of quiescence, which later can be used to burst when resources are required.

Where a per-process token bucket would allow for a CPU resource limitation of a single process, a per-context token bucket allows to control the CPU usage of all confined processes.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Another approach – which is also implemented – is to use the current fill level of the bucket to adjust the process priority, thus reducing the priority of processes belonging to excessive contexts.

Hard CPU Limit

The simplest scheduler configuration is to give every context an upper bound for CPU allocation, i.e. a context cannot utilize more timeslices than its limit. Consider a context which should get at most L percent of CPU time. Given that a running process/context consumes one token at each timer tick, then the upper bound for allocation can be determined by the following formula:

$$L = \frac{R}{T}$$

It is advantageous to smooth operation of the algorithm to make the interval as small as possible (or much smaller than the bucket size). You can in most cases simplify the fraction, such as changing $30/100$ to $3/10$.

Burst time

While a general upper bound for CPU allocation prevents processes from eating all CPU time, it may be advantageous to allow CPU-bound processes a certain amount of *burst time*, i.e. the hard CPU limit operates not until burst time is over.

Consider a context with a hard limit of $1/2$ of CPU time and a bucket size of 15000 tokens. Given that the kernel timer runs at 1000Hz, CPU-bound processes may use 100% of CPU-time for 30 seconds until the hard limit kicks in. The following formula can be used to calculate the maximum size S of a bucket, using B as the desired burst time in seconds:

$$S = B \cdot \mathrm{Hz} \cdot (1 - L)$$

Hold time

During burst time a process/context uses 100% of CPU time before getting penalized by the hard limit. However, there is no extra penalty for these processes yet, though in some configurations it may be desirable to hold processes for a certain amount of time after they have used up all their burst time.

Again, consider a context with a hard limit of $1/2$ of CPU time. To penalize processes after they have used up all their burst time, the minimum size M of the bucket can be calculated with the following formula, using H as the desired hold time in seconds:

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

$$M = H \cdot \mathrm{Hz} \cdot L$$

Therefore, a hold time of 5 seconds would result in a minimum bucket size of 2500 tokens given that the kernel timer runs at 1000Hz.

Guarantees

A guarantee is nearly the same as a pure hard limit, except that you *must not* allocate more than 100% of CPU time to all (running) contexts. In other words, if there are N running contexts and each one should have a guarantee of more then $1/N$ of CPU time, it would result in less CPU time for each context than it was originally guaranteed. The important factor here is the sum of hard limits for all running contexts:

$$\sum_{i=1}^{N} L_i \leq 1$$

Fair Share

While hard limits and guarantees allow an upper bound for CPU allocation, it still has a noticable flaw. Consider a configuration with 5 contexts each limited to $1/5$ of CPU time, where two of these contexts are running CPU intensive processes and the rest is idle. Given that each context may only allocate $1/5$ of CPU time, $3/5$ of CPU time are wasted since 3 contexts are idle.

To distribute the wasted CPU time *fair* among contexts that could need it, the Linux-VServer kernel allows to artificially advance time during idle times, so buckets get refilled in an instant according to a second limit/ratio, namely R^2 and T^2.

Again, consider 5 configured contexts, each limited to $1/5$ of CPU time. Whenever at least one context is idle the remaining CPU time should be distributed equally among all other running contexts. Therefore if two of five contexts are running, the remaining $3/5$ of CPU time should be divided into $3/10$ for each of them. The following formula can be used to calculate the percentage of CPU time D allocated to context k of N *running* contexts, using C as the idle CPU time:

$$D_k = C \cdot \frac{1}{N} + L_k$$

However, this formula will only procude correct results if L^2 is the same for all (running) contexts, since it is merely a special case of the following formula which can be used to calculate CPU distribution if L^2 differs from context to context:

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

$$D_k = \frac{C \cdot L_k^2}{\sum_{i=1}^{N} L_i^2} + L_k$$

Therefore L^2 may not be considered the percentage of idle time a context will get assigned, but rather a proportion of all L_k^2 values among running contexts.

2.4.8. Virtual Host Information

One major difference between the Linux-VServer approach and virtual machines is that there is no virtualization part as a side-effect, so some aspects that would be covered by virtual machines have to be controlled manually where approriate.

For example, a virtual machine does not need to think about system uptime, because naturally the operating system running inside the virtual machine was started somewhere in the past and will not have any problem to tell the time it thinks it began running.

A context can also store the time when it was created, but that will be different from the systems uptime, so in addition, there has to be some function, which adjusts the values passed from kernel to user-space depending on the context the process belongs to.

This is what the Linux-VServer kernel actually refers to as as virtualization, although it is merely faking some values passed to and from the kernel to make the processes think that they are on a different machine.

Currently the following information is modified/faked for the purpose of virtualization:

- system time
- system uptime
- host and domain name
- machine type and kernel version
- context memory availability
- context disk space
- context load average

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

2.4.9. Filesystem Namespace

Namespaces are already include in the Linux kernel and allows processes to have a different view of the filesystem. By default, there is only a single filesystem tree shared by all processes, however processes can request a new namespaces by means of the `clone` system call.

For example, if there exist two processes A and B in different namespaces, a `mount` system call issued by process A will not affect any mounts in the namespace of process B and vice-versa.

A namespace is automatically destroyed once all processes using it have died. All mounts are automatically unmounted in that namespace, so there is no need to unmount them manually. In fact, there would be no way to unmount a device in a namespace that does not exist anymore, therefore it cannot happen that some dead mounts hang around in unreachable namespaces preventing, for example, removing a media from cdrom drive.

There are mainly two reasons for the usage of filesystem namespaces. On the one hand the hosts namespace will not be poluted with mounts for context, i.e. `/proc/mounts` is virtualized. On the other hand there is an extra layer of security added by recursively bind mounting the virtual servers root directory to / in namespaces belonging to contexts, therefore even if a chroot exploit is discovered, an attacker would still see the virtual servers root filesystem.

2.4.10. Filesystem Attributes

Originally, filesystem attributes were only available with ext2, but now all major filesystems implement a basic set of File Attributes that permit certain properties to be changed. Here is an excerpt from the `chattr(1)` man page to give an overview of some attributes, and what they mean. For a detailed reference, please refer to the `chattr(1)` man page.

- When a file with the **A** attribute set is accessed, its atime record is not modified. This avoids a certain amount of disk I/O for laptop systems.
- A file with the **a** attribute set can only be open in append mode for writing. Only the superuser or a process possessing the `CAP_LINUX_IMMUTABLE` capability can set or clear this attribute.
- A file with the **c** attribute set is automatically compressed on the disk by the kernel.
- When a directory with the **D** attribute set is modified, the changes are written synchronously on the disk.
- A file with the **d** attribute set is not candidate for backup when the dump(8) program is run.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- A file with the **i** attribute cannot be modified: it cannot be deleted or renamed, no link can be created to this file and no data can be written to the file. Only the superuser or a process possessing the `CAP_LINUX_IMMUTABLE` capability can set or clear this attribute.
- When a file with the **s** attribute set is deleted, its blocks are zeroed and written back to the disk.
- When a file with the **S** attribute set is modified, the changes are written synchronously on the disk.
- A file with the **t** attribute will not have a partial block fragment at the end of the file merged with other files.
- When a file with the **u** attribute set is deleted, its contents are saved. This allows the user to ask for its undeletion.

Information regarding file attributes can be found in the kernel source code. Every file system uses a subset of all known attributes, though it depends on the file system which attributes are defined or even used.

One thing can be said for sure – the file attributes listed in the kernel source code are defined – those that are not listed are not defined and in turn can not be used for a particular file system. However, many of those file attributes defined and understood by the kernel have no effect. Most file systems define those flags in a specific header file found within the kernel source tree. They also define a so called user-modifiable mask (those are the flags the user can change with the ioctl system call).

Those flags have also partially different meaning depending on the node type (i.e. dir, inode, fifo, pipe, device) and it is not trivial to say if a filesystem makes use of any user modifiable flag – things like immutable are easy to verify (from user space) but how to verify e.g. tail-merging from user space? Usually only source code review will show if it is implemented and used.

For example, as of writing, the ext2/3 filesystem do not support compression, although it is defined and well understood by ext2/3 but there is no implementation, i.e. nothing is compressed.

Therefore, the Linux-VServer kernel implements an own set of filesystem attributes for most vanilla file systems, including ext2/3/4, reiser, xfs, jfs, ocfs2 and others. The next sections discuss file system attributes and tagging in greater detail, since these attributes are used for different aspects of isolation/security.

2.4.11. Filesystem Tagging

Although it can be disabled completely, this modification is required for more robust filesystem level security and context isolation. It is also mandatory for context disk limits and per-context quota support on a shared partition.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

The concept of adding a context id to each file to make the context ownership persistent sounds simple, but the actual implementation is non-trivial – mainly because adding this information either requires a change to the on disk representation of the filesystem or the application of some tricks.

One non-intrusive approach to avoid modification of the underlying filesystem is to use the upper bits of existing fields, like those for UID and GID to store the additional context id.

Once context information is available for each inode, it is a logical step to extend the access controls to check against context too. Currently all inode access restrictions have been extended to check for the context id, with special exceptions for the host and spectator contexts.

Untagged files belong to the host context and are silently treated as if they belong to the current context, which is required for unification. If such a file is modified from inside a context, it silently migrates to the new one, changing its xid.

The following tagging methods are implemented:

UID32/GID32 This format uses currently unused space within the disk inode to store the context information. As of now, this is only defined for ext2/ext3 but will be also defined for xfs, reiserfs, and jfs as soon as possible. Advantage: Full 32bit uid/gid values.

UID32/GID16 This format uses the upper half of the group id to store the context information. This is done transparently, except if the format is changed without prior file conversion. Advantage: works on all 32bit U/GID FSs. Drawback: GID is reduced to 16 bits.

UID24/GID24 This format uses the upper quarter of user and group id to store the context information, again transparently. This allows for about 16 million user and group ids, which should suffice for the majority of all applications. Advantage: works on all 32bit U/GID FSs. Drawback: UID and GID are reduced to 24 bits.

2.4.12. Unification

Because one of the central objectives for Linux-VServer is to reduce the overall resource usage wherever possible, a truly great idea was born to share files between different contexts without interfering with the usual administrative tasks or reducing the level of security created by the isolation.

Files common to more than one context, which are not very likely going to change, like libraries or binaries, can be hard linked on a shared filesystem, thus reducing the amount of disk space, inode caches, and even memory mappings for shared libraries.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

The only drawback is that without additional measures, a malicious context would be able to deliberately or accidentally destroy or modify such shared files, which in turn would harm the other contexts.

One step is to make the shared files immutable by using the `IMMUTABLE` file attribute (and removing the Linux capability required to modify this attribute). However an additional attribute (`IUNLINK`) is required to allow removal of such immutable shared files, to allow for updates of libraries or executables from inside a context.

Such hard linked, immutable but unlink-able files belonging to more than one context are called *unified* and the process of finding common files and preparing them in this way is called *unification*.

The reason for doing this is reduced resource consumption, not simplified administration. While a typical Linux server install will consume about 500MB of disk space, 10 unified servers will only need about 700MB and as a bonus use less memory for caching.

2.4.13. ProcFS Security

ProcFS security provides a mechanism to protect dynamic entries in the proc filesystem from being seen in every context. The system consists of three flags for each ProcFS entry: `ADMIN`, `WATCH` and `HIDE`.

The `HIDE` flag enables or disables the entire feature, so any combination with the `HIDE` flag cleared will mean total visibility. The `ADMIN` and `WATCH` flags determine where the hidden entry remains visible – for example if `ADMIN` and `HIDDEN` are set, the host context will be the only one able to see this specific entry.

2.4.14. Chroot Barrier

The `chroot` system call changes the root directory of the current process. This directory will be used for pathnames beginning with `/`. The root directory is inherited by all children of the current process.

However several problems are known while using the chroot system call:

- This call changes an ingredient in the pathname resolution process and does nothing else.
- This call does not change the current working directory
- This call does not close open file descriptors

These facts disclose several ways to break out of a chroot environment, back to the original root. A more detailed description of know exploits and their explanation can be found on the Linux-VServer project website [**?**] and will not be discussed in this book.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

While early Linux-VServer versions tried to fix this by "funny" methods, recent versions use a special marking, known as the *chroot barrier*, on the parent directory of each VPS to prevent unauthorized modification and escape from confinement. This barrier is implemented as a Filesystem Attribute and prevents a path walk into a directory with enabled barrier.

2.4.15. Disk Limits & Quota

This Feature requires the use of tagged files, and allows for independent disk limits for different contexts on a shared partition. The number of inodes and blocks for each filesystem is accounted, if a hash was added for the context-filesystem tuple.

Those values, including current usage, maximum and reserved space, will be shown for filesystem queries, creating the illusion that the shared filesystem has a different usage and size, for each context.

Similar to disk limits, per-context quota uses separate quota hashes for different contexts on a shared filesystem. This is not required to allow for Linux-VServer quota on separate partitions.

2.5. History

Jacques Gélinas created the VServer project a number of years back. He still does vserver development and the community can be glad to have him. He's a genius, without him, Linux-VServer would not exist. Three cheers for Jack.

But sometime during 2003 it became apparent that Jack didn't have the time to keep vserver development up to pace. So in November, **Herbert Pötzl** officially took charge of development. He now releases the vserver kernel patches, announcing them on the vserver mailing list and making them available for the public.

Additionally, **Enrico Scholz** decided to reimplement Jack's vserver tools in **C**. These are now distributed as util-vserver. They are backward compatible to Jack's tools as far as possible, but follow the kernel patch development more closely.

In 2005, **Benedikt Böhm** started another reimplementation of the userspace utilities, although with a completely different architecture in mind. This new implementation is known as VServer Control Daemon as you already may know when you read this manual `;-)`

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

3. The VServer Control Daemon

3.1. Abstract

In order to ease management of Virtual Private Servers a central instance is needed to provide an open and well-known *Application Programming Interface (API)* not only for local, but especially for remote usage and execution. The *VServer Control Daemon (VCD)* is a daemon running in the host context and provides the aforementioned API using *XML-RPC*, a simple protocol for *Remote Procedure Calls (RPC)* using the XML Markup Language.

3.2. Rationale

The current user-space implementation of the Linux-VServer kernel API suffers a mechanism to call any of the management commands regardless of the language or location of the caller. Such callers include non-C lanuages like Python, PHP or Ruby, remote GUIs for KDE, Gnome or even Windows as well as web control panels for service providers.

Therefore the VServer Control Daemon defines an API accessible by any caller capable of both the HTTP and the XMLRPC protocol – two open standards implemented in most common languages.

3.3. Architecture

The complete VServer Control Daemon package consists of four independent modules:

libvserver The libvserver module contains a library that implements the Linux-VServer system call in the C language. It provides a convenient way to execute Linux-VServer commands in other programs or libraries. Additionally basic low-level command-line tools are provided to work with these commands.

vcd The VServer Control Daemon module uses – among other libraries – libvserver to implement the core of the package. This module includes the configuration storage backend, the XML-RPC server, command-line clients and the kernel user-space helper.

vwrappers The vwrappers module contains a lot of wrappers for system packages usually found on Linux systems. This provides a convenient way to execute commands inside virtual private servers. Additionally, some utilities for the extended filesystem attributes are provided enhancing the versions in libvserver with directory tree recursion and various other options.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

vstatd The VServer Statistics Daemon module contains a *very* small daemon that keeps track of resource utilization of virtual private servers. For statistical analysis all collected data is stored in a round-robin database (RRD), the industry standard data logging and graphing application.

A detailed descriptions of these modules can be found in the next sections. However, the reader may skip to part II on page 38 for the installation guide and jump back here occasionally once a more detailed module explanation is needed.

3.4. Linux-VServer system call library

The Linux-VServer project provides an interface to the various virtualization and configuration commands that have been added to the kernel. This interface between the operating system and the user programs or libraries is known as *system call*.

For a long time the Linux-VServer project used a few different system calls to accomplish different aspects of virtualization and configuration. However, the number of commands grew rapidly and a few system calls were simply not enough to handle every aspect of virtualization. Therefore, the Linux-VServer system call now uses a system call multiplexer based on a command identifier, which has similar semantics like the `ioctl` system call.

The Linux kernel maintainers have reserved a system call number for the Linux-VServer project on most architectures now and while the opinion on system call multiplexing might differ from developer to developer, it was generally considered a good decision not to have more than one system call...

The advantage of different system calls would be simpler handling on different architectures. However, this has not been a problem so far, as the data passed to and from the kernel has strong typed fields conforming to the C99 types.

The Linux-VServer system call requires three arguments regardless of what the actual command is:

```
sys_vserver(uint32_t cmd, uint32_t id, void __user *data);
```

cmd a command number

id an identifier

data a user-space data-structure of yet unknown size

To allow for some structure for debugging purposes and some kind of command versioning, the command number is split into three parts: the lower 12 bits contain a version number, then 4 bits are reserved, the upper 16 bits are divided into 8 bits for the command number and 6 bits for category number, again reserving 2 bits for the future.

In theory, there are 64 categories with up to 256 commands in each category, allowing for 4096 revisions of each command, which is far more than will ever be required.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

The following categories and commands are defined in the Linux-VServer 2.2.0 release:

```
       |VERSION|CREATE |MODIFY |MIGRATE|CONTROL|EXPERIM| |SPECIAL|SPECIAL|
       |STATS  |DESTROY|ALTER  |CHANGE |LIMIT  |TEST   | |       |       |
       |INFO   |SETUP  |       |MOVE   |       |       | |       |       |
-------+-------+-------+-------+-------+-------+-------+ +-------+-------+
SYSTEM |VERSION|VSETUP |VHOST  |       |       |       | |DEVICES|       |
HOST   |     00|     01|     02|     03|     04|     05| |     06|     07|
-------+-------+-------+-------+-------+-------+-------+ +-------+-------+
CPU    |       |VPROC  |PROCALT|PROCMIG|PROCTRL|       | |SCHED. |       |
PROCESS|     08|     09|     10|     11|     12|     13| |     14|     15|
-------+-------+-------+-------+-------+-------+-------+ +-------+-------+
MEMORY |       |       |       |       |       |       | |SWAP   |       |
       |     16|     17|     18|     19|     20|     21| |     22|     23|
-------+-------+-------+-------+-------+-------+-------+ +-------+-------+
NETWORK|       |VNET   |NETALT |NETMIG |NETCTL |       | |SERIAL |       |
       |     24|     25|     26|     27|     28|     29| |     30|     31|
-------+-------+-------+-------+-------+-------+-------+ +-------+-------+
DISK   |       |       |       |       |DLIMIT |       | |INODE  |       |
VFS    |     32|     33|     34|     35|     36|     37| |     38|     39|
-------+-------+-------+-------+-------+-------+-------+ +-------+-------+
OTHER  |VSTAT  |       |       |       |       |       | |VINFO  |       |
       |     40|     41|     42|     43|     44|     45| |     46|     47|
=======+=======+=======+=======+=======+=======+=======+ +=======+=======+
SPECIAL|EVENT  |       |       |       |FLAGS  |       | |       |       |
       |     48|     49|     50|     51|     52|     53| |     54|     55|
-------+-------+-------+-------+-------+-------+-------+ +-------+-------+
SPECIAL|DEBUG  |       |       |       |RLIMIT |SYSCALL| |       |COMPAT |
       |     56|     57|     58|     59|     60|TEST 61| |     62|     63|
-------+-------+-------+-------+-------+-------+-------+ +-------+-------+
```

While a single system call is advantageous from the kernel point of view, user programs and libraries tend to have a single function per command, although this is not always the case for multiplexed system calls, as can be seen with `ioctl`. Nevertheless, the libvserver module implements one function for each command the Linux-VServer multiplexer knows about.

For this purpose, the library uses a custom syscall implementation written by Herbert Pötzl rather than using the macros provided by `libc` to make the library as self-contained as possible.

The complete interface documentation for libvserver can be found on the projects' website [9] and will not be discussed in further detail here.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

3.5. VServer Control Daemon

The VServer Control Daemon is the core of the the package and implements all aspects of virtual server management, using `libvserver` to issue Linux-VServer system calls, `xmlrpc-c` to implement the XML-RPC protocol and `SQLite3` for data storage. Figure 3.1 illustrates the basic relationship between libraries, XML-RPC methods and clients.

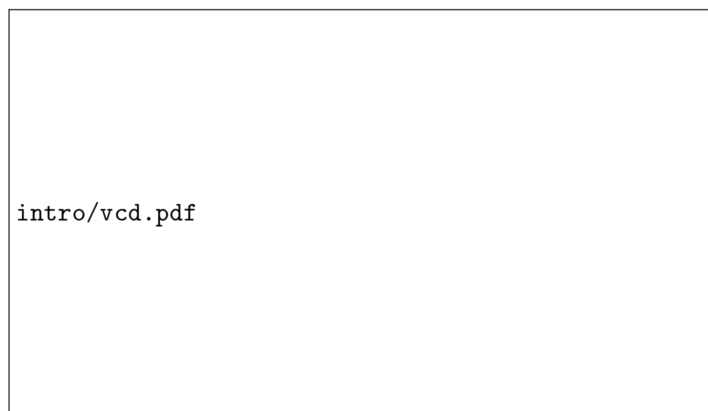

Figure 3.1.: VServer Control Daemon Architecture Overview

3.5.1. Configuration Database - VXDB

The configuration database (*VXDB*) stores all virtual private server related configuration data like disk limits, CPU scheduler buckets, or network adresses. Furthermore the daemon stores information about its users and access control as well as owner information in the database. For convenience and size reasons the database is implemented using *SQLite3*.

SQLite3 is a small C library that implements a self-contained, embeddable, zero-configuration SQL database engine. The decision for using SQLite as database backend is based on the following key-features of SQLite:

- Transactions are atomic, consistent, isolated, and durable even after system crashes and power failures
- Zero-configuration - no setup or administration needed
- A complete database is stored in a single disk file
- Database files can be freely shared between machines with different byte orders
- Small code footprint: less than 250KB
- Faster than popular client/server database engines for most common operations

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- Simple, easy to use API
- Self-contained: no external dependencies

3.5.2. The XML-RPC Server

The *XML-RPC Server* is the core of the VServer Control Daemon and implements the XML-RPC standard for Remote Procedure Calls (RPC). XMLRPC is a specification and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Wide Area Network (Internet) or Local Area Network (Intranet).

The XML-RPC Protocol

XML-RPC is a wire protocol that describes an XML serialization format that clients and servers use to pass remote procedure calls to each other. There are two features that make this protocol worth knowing. The first is that the details of parsing the XML are hidden from the user. The second is that clients and servers don't need to be written in the same language.

XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned. Figure 3.2 illustrates the serialzation/deserialization in an XML-RPC session.

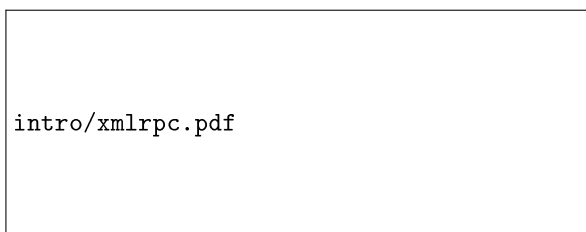

Figure 3.2.: XMLRPC data flow

Here are some examples of remote procedure call (RPC) style communications:

- There is a server that can measure atmospheric temperature. A client anywhere in the world can ask the server at any time what the temperature is. The "what temperature is it?" request and the "the temperature is..." response constitute an RPC transaction.
- There is a server that can turn a light on or off. A client can tell the server to turn the light on. A request to turn the light on and the acknowledgement that the light has been turned on constitute an RPC transaction.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

- There is a server that knows the phone numbers of a million people. A client can supply a name and get back the phone number of the named person.

Here are some kinds of communication that are not RPC:

- A long-lived connection such as an SSH login session.
- A high volume transfer such as an FTP download.
- A one-way transmission such as a UDP packet.
- A dialogue such as an SMTP (mail) transaction.

Based on XML nearly any application can be enabled to call methods defined by the XML-RPC Server. Additionally, the fact that XML is written in plain-text and also easily parsable by humans allows easy tracing and debugging of XML-RPC sessions in case of failure or other strange behaviour.

On startup, the server defines a global registry of methods accessible by its clients. In order to allow structure for the amount of methods currently defined, all methods are devided in logical units – also known as namespaces – and prefixed with the namespace and a dot, like `vx.start` or `helper.shutdown` for the start and shutdown methods of the vx and helper namespace, respectively.

Refer to part V on page 46 for a detailed description of the XML-RPC protocol and the request and response format used for defined methods.

Authentication

Authentication in the VServer Control Daemon is based on the cryptographic hash function *WHIRLPOOL*. WHIRLPOOL is a cryptographic hash function designed by Vincent Rijmen and Paulo S. L. M. Barreto. The hash has been recommended by the NESSIE project. It has also been adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) as part of the joint ISO/IEC 10118-3 international standard.

WHIRLPOOL is a hash designed after the Square block cipher. WHIRLPOOL is a Miyaguchi-Preneel construction based on a substantially modified Advanced Encryption Standard (AES). Given a message less than 2256 bits in length, it returns a 512-bit message digest.

For security reasons the clear-text password is never stored in VXDB. The client will send the password as plain-text – the server then creates a WHIRLPOOL hash using the submitted password and compares its result with the hash stored in VXDB. It is also possible to create the hash digest on the client side, prefix the result with the string `WHIRLPOOLENC//` and send the result to the server for authentication. The server then compares the hash after stripping the the prefix again.

Note that an attacker can still authenticate against the server with the hash only, he just cannot retreive the original password again. Therefore, the server should *never* listen on

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

an untrusted network if no other security measures have been setup, like an SSL or VPN tunnel.

Access Restrictions

For a fine-grained access control the server implements its own set of capabilities. A capability is a lot like a drivers license. As an example, consider your car license. It allows you to drive a certain set of vehicles (it designates a particular object set, in our case XML-PRC methods), and anyone holding a car license is allowed to drive the same set of objects (cars).

Refer to part V on page 46 for a detailed description of available access restrictions and their usage in defined methods.

Owner Checks

To ensure the distinction between your car and your neighbours Lamborghini another access control system has to be implemented. Therefore the server also implements owner checks for most of its methods. This results in an extension to the capability model explained above. The server implements another object set based on the ownership information stored in the database. Instead of allowing everyone holding a certain capability to operate on all virtual servers, only the intersection of both object sets is permitted for operation.

Still, this model has a noticable flaw: Imagine a company has two hundred cars and the top management should have access to all cars. Adding all members of the management to the owner list of every single car can become a pain in the ass very quickly. Therefore the user database in VXDB implements an additional configuration option – the administrator flag. Using this flag all owner checks are passed without even consulting the owner lists in the database.

3.5.3. XML-RPC Clients

The *XML-RPC Clients* provided in the VServer Control Daemon module provide facilities to connect and execute methods on remote servers. Several clients exist for different purpose, in most cases they are aligned with the method namespaces mentioned above. Available commands are discussed in detail in part IV on page 40.

It is important to know that the connection between server and client is not persistent, i.e. you send one request, get one answer, and the connection will be closed afterwards. This also implies the necesity of passing authentication information with every method call. After the request has been received and processed the method returns a fault notification in case of any error or a method specific return value.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Refer to part V on page 46 for a detailed description of the XML-RPC protocol and the request and response format used for defined methods as well as error codes and their meaning.

3.5.4. Kernel Helper

For some purposes, it makes sense to have a user-space tool to act on behalf of the kernel, when a process inside a context requests something usually available on a real server, but naturally not available inside a context.

The best example for this is the `reboot` system call, when invoked from inside a virtual server, the kernel helper on the host system will be called to perform the shutdown (and probably startup) procedure for this particular virtual server and not for the physical machine.

Previously, the kernel helpers sole purpose was to handle reboot requests. The helper implemented for VCD, however, uses the full aspect of the helper interface now. This includes creation and disposal of virtual servers and keeping track of reboot requests. Figures 3.3 and 3.4 illustrate the rather complex assembly of the startup and shutdown procedure, respectively.

Figure 3.3.: Interaction between vshelper and vcd on startup

During startup of virtual servers, i.e. a client has called the `vx.start` method, the daemon issues a system call to create the necessary network and process context. The kernel then calls the kernel helper using the `startup` action, which in turn calls the `helper.startup` method.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

The daemon then queries the database for initial configuration, like capabilities, network addresses or scheduler values, issues the approriate system calls to set these configuration values on the network and process context and returns the absolute path to the init binary relative to the virtual servers root filesystem to the helper.

Having successfully retreived the path to the init binary the kernel helper is now able to fork a new process, migrate it to the network and process context created before, and execute the init binary. The parent process then blocks until at least one process appeared inside the context. However, this is the case as soon as the child process has issued the system call for migration, therefore the helper is not able to check if the init command actually succeeded.

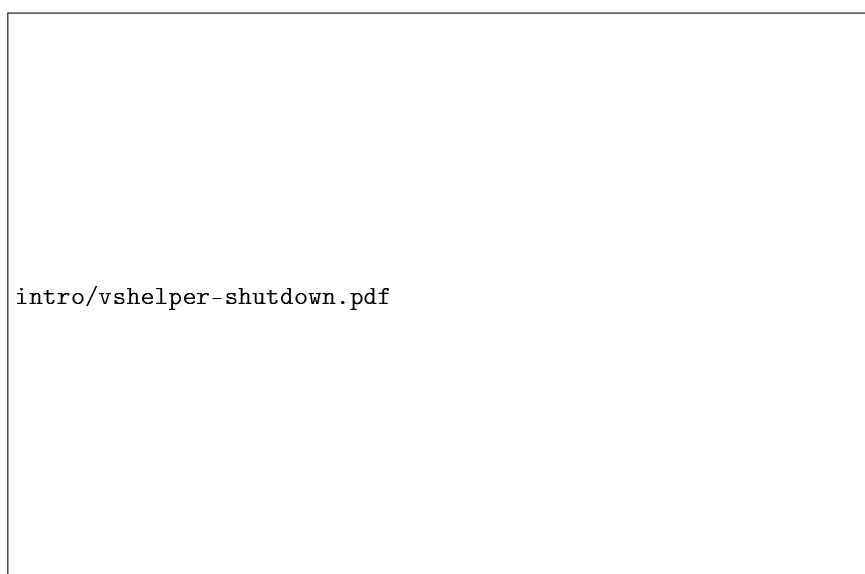

Figure 3.4.: Interaction between vshelper and vcd on shutdown

Disposal and potential restart is even more complicated by the fact that the helper is called twice during shutdown. Once a client has called the `vx.stop` method or the root user inside the virtual server has called the `halt` command, the virtual servers init system shuts down all running services and calls `sys_reboot` afterwards, with a special magic to distinct halt and reboot.

On a reboot request the kernel then calls the helper using the `reboot` action, which in turn calls the `helper.reboot` method to record the reboot request in the database as it is neede later. As soon as the helper returns the kernel kills the init process – and possibly other remaining processes – and calls the helper a second time using the `shutdown` action.

The shutdown helper in turn waits for the network and process context to disappear and calls the `helper.shutdown` method afterwards. As long as no reboot request was recorded the daemon does nothing and returns immediately. Otherwise, it will call itself using the `vx.start` method to initiate the startup sequence again.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

3.5.5. The Template Management

The *Template Management* consists of various scripts and XMLRPC methods used to build and create new virtual private servers. The *Template Build* process assembles a complete root filesystem usable in virtual private servers, and stores its content in a single tarball, the *Template Cache.*

3.6. Command Line Wrappers

3.7. VServer Statistics Daemon

The *Statistics Collector* is a very light-weight daemon used to collect time-series data of running virtual private servers. This data includes memory usage, number of processes or cpu usage. The collected data is stored in *Round Robin Databases (RRD)* - the industry standard data for logging and graphing applications - for later use in reports or graphing processes.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Part II.

Installation

Part III.

Configuration and Maintenance

Part IV.

Command Reference

4. VServer Control Client

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

5. VServer Control Daemon

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

6. VServer Control Daemon Administration Tool

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

7. VServer Configuration Editor

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

8. VServer Kernel Helper

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Part V.

XML-RPC Method Reference

9. Introduction

9.1. XML-RPC Signatures

The XML-RPC signatures used in this part of the manual define a translation between the XML-RPC value and C-friendly data types that represent the same information. For example, it might say that a floating point number translates to or from a C double value, or that an array of 4 integers translates to or from 4 C int values.

A format string usually describes the type of one XML-RPC value. But some types (array and structure) are compound types – they are composed recursively of other XML-RPC values. So a single format string might involve multiple XML-RPC values.

But first, lets look at the simple (not compound) format types. These are easy. The format string consists of one of the strings in the list below. The string is called a format specifier, and in particular a simple format specifier.

The XML-RPC server uses only three native XML-RPC datatypes:

int signed 32-bit integer (-2,147,483,648 to +2,147,483,647)

bool boolean value (true/false)

string character string of arbitrary length

Unfortunately the XML-RPC protocol does not define 64-bit integer values, therefore the XML-RPC server uses the native string datatype and converts all leading digit characters to 64-bit integer representation.

This chapter uses the following convention to denote these integer datatypes:

int32 signed 32-bit integer ($-2^{31} \cdots 2^{31} - 1$)

uint32 unsigned 32-bit integer ($0 \cdots 2^{32} - 1$)

int64 signed 64-bit integer ($-2^{63} \cdots 2^{63} - 1$)

uint64 unsigned 64-bit integer ($0 \cdots 2^{64} - 1$)

The simple datatypes mentioned above are not sufficent to represent data in an efficient way. Therefore compound datatypes are defined in XML-RPC. There are two compound datatypes:

array An array holds a series of data elements. Individual elements are accessed by their position in the array.

struct A structure is a collection of simple datatypes under a single compound. This collection can be of different types, and each has a name which is used to select it from the structure. A structure is a convenient way of grouping several pieces of related information together.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

The signature denotes arrays using normal braces, structures use curly braces. Here are some examples of XML-RPC signatures:

- A single `string` datatype named *path*:
  `string path`
- An `array` of three int datatypes:
  `(int int, int)`
- An `array` with one `bool`, one `string` and one `int` datatype:
  `(bool, string, int)`
- A `struct` with one `string` and one `int` datatype:
  `{string name, int id}`

Please note that the next chapters use the following convention to declare methods and their parameters, although they do not involve arrays.

```
vx.create(string name, string template, int rebuild)
```

Instead this convention matches the function declaration in the C language. Translated to a XML-RPC signature format string the above would become a struct:

```
{string name, string template, int rebuild}
```

Read on to the next section to learn how method requests are performed and how this struct of parameters is placed in the request.

9.2. Performing XML-RPC Requests

An XML-RPC method call is an HTTP-POST request. The body of the request is in XML. The specified method executes on the server and the value it returns is also formatted in XML.

Here is an example of an XML-RPC request:

9.2.1. Header Requirements

The following requirements must be met when sending requests to the XML-RPC server:

- The `/RPC2` location handler to explicitly denote XML-RPC requests
- A User-Agent and Host must be specified
- The Content-Type is `text/xml`
- The Content-Length must be specified and must be correct

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

9.2.2. Payload Format

The payload is in XML, a single <methodCall> structure.

The <methodCall> must contain a <methodName> sub-item, a string, containing the name of the method to be called. The string may only contain identifier characters, upper and lower-case A-Z, the numeric characters, 0-9, underscore, dot, colon and slash.

If the procedure call has parameters, the <methodCall> must contain a <params> sub-item. The <params> sub-item can contain any number of <param>s, each of which has a <value>.

Simple Datatypes

The XML-RPC server only uses three native datatypes as mentioned above:

int signed 32-bit integer

boolean 0 (false) or 1 (true)

string character string of arbitrary length

If no type is indicated, the type is string.

Structures

A value can also be of type <struct>. A structure contains <member>s and each <member> contains a <name> and a <value>. Here is an example of a two-element <struct>:

Structures can be recursive, any <value> may contain a <struct> or any other datatype, including an <array>, described below.

Arrays

A value can also be of type <array>. An array contains a single <data> element, which can contain any number of <value>s. Here's an example of a four-element array:

Unlike structures array elements do not have names. In contrary to C arrays you can mix datatypes as the example above illustrates.

Arrays can be recursive, any value may contain an <array> or any other type, including a <struct>, described above.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

9.2.3. Method Initialization

The XML-RPC server implemented in VCD extends the request format to allow user authentication. This extension is fully compatible with the standard XML-RPC protocol, since it is implemented using the `<params>` section of the XML-RPC request. A client submits username and password as first `<param>` in the `<params>` section using the following struct signature:

```
{string username, string password}
```

username unique username with a maximum of 64 characters

password password for the specified user

The second `<param>` in the `<params>` section is a variable-length struct of parameters specific to the requested method. Refer to the next chapters for a description of all methods and their parameters.

Here is an example of a valid XML-RPC request to VCD:

Once the XML-RPC request has been received by the server it translates the XML payload to internal data representation, performs user authentication and calls the specified method using the struct in the second `<param>` value as method parameters.

9.2.4. Authentication and Access Restrictions

As mentioned in section 3.5 authentication is based on the cryptographic hash function WHIRLPOOL. The server generates a WHIRLPOOL hash using the specified password in the authentication struct in the first `<param>` value. Afterwards it looks up the specified user in the database and compares the stored password hash with the previously generated hash. If the hashes do not match the server returns an error.

Additionally the XML-RPC server implements a set of capabilities to enable certain functionality for specific users only. If the requested method requires capabilities not configured for the authenticated user the server returns an error. The server implements the following capabilities:

AUTH User may modify the internal user database

DLIM User may modify disk limits

INIT User may start/stop virtual servers

MOUNT User may modify mount points

NET User may modify network interfaces

BCAP User may modify system capabilities

CCAP User may modify context capabilities

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

CFLAG User may modify context flags

RLIM User may modify resource limits

SCHED User may modify context schedulers

UNAME User may modify utsname/virtual host information

CREATE User may create and destruct virtual servers

EXEC User may execute commands in virtual server

INFO User may retrieve internal server information

HELPER User may call helper methods

Please refer to the next chapters to learn which method requires which capability.

If the user has passed authentication most of the implemented methods in VCD do owner checks. In general all methods expecting a string name as first parameter do owner checks. A user can pass owner checks in two ways:

- He is listed as owner in the owner list of the virtual server specified in name
- He has the admin flag enabled in VXDB

Once the user has passed all the authentication steps mentioned above, the specified method executes and returns a fault notification or a method specific return value.

9.2.5. Response Format

An XML-RPC response is a normal HTTP response. The body of the response is in XML. The specified method has executed on the server and the value it now returns is also formatted in XML.

- Unless there is a low-level error, the HTTP server always returns 200 OK.
- The Content-Type is `text/xml`.
- Content-Length must be present and correct.

The body of the response is a single XML structure, a `<methodResponse>`, which can contain a single `<params>` which contains a single `<param>` which contains a single `<value>`.

The `<methodResponse>` could also contain a `<fault>` which contains a `<value>` which is a `<struct>` containing two elements, one named `<faultCode>`, an `<int>` and one named `<faultString>`, a `<string>`.

A `<methodResponse>` can not contain both a `<fault>` and a `<params>`. Here is an example of a response to an XML-RPC request:

A fault might look like the following example:

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Please note that even in the case of a fault notification the HTTP status code is still 200 OK. Refer to the next section to learn about possible error codes and their meaning.

9.3. Method Error Codes

This part of the manual devides error codes in two categories: generic method errors and specific method errors. Although they do not differ in their value or implementation it is advisable to know that the same error code may have a slightly different meaning for a specific method. The following table explains all error codes and their generic meaning.

Constant	Value	Description
MEAUTH	100	Unauthorized
MEPERM	101	Operation not permitted
MENOUSER	102	User does not exist
MEINVAL	200	Invalid argument
MEEXIST	201	An object already exists
MENOVPS	202	Virtual server does not exist
MENOVG	203	Virtual server group does not exist
MESTOPPED	300	Virtual server not running
MERUNNING	301	Virtual server still/already running
MEBUSY	302	Operation still in progress
MEVXDB	400	Low-level database error
MECONF	401	Invalid configuration file
MESYS	500	System call or library call failed
MEEXEC	1000+	Command execution failed

Please refer to the next chapters to learn which methods apply different meaning to these error codes.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

10. Helper Methods

10.1. helper.netup

This method is used to setup necessary network configuration during startup of virtual servers. It is called by `vshelper` once the kernel has created the network context. This method is not supposed to be called by a user or administrator.

Synopsis

```
helper.netup(int xid)
```

xid Unique context ID

Access Restrictions

This method requires the `HELPER` capability and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

10.2. helper.restart

This method is used to remember reboot requests from vitrtual servers. It is called by `vshelper` once virtual servers have issued a reboot system call. After the request has been stored in VXDB the kernel terminates all processes belonging to the virtual server, and calls `helper.shutdown` afterwards. This method is not supposed to be called by a user or administrator.

Synopsis

```
helper.restart(int xid)
```

xid Unique context ID

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Access Restrictions

This method requires the HELPER capability and does not need to pass owner checks.

Return Value

On success NIL is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

10.3. helper.shutdown

This method is used to cleanup state data after virtual server shutdown. If a reboot request has been recorded in VXDB by `helper.restart` the virtual server is started again. This method is not supposed to be called by a user or administrator.

Synopsis

```
helper.shutdown(int xid)
```

xid Unique context ID

Access Restrictions

This method requires the HELPER capability and does not need to pass owner checks.

Return Value

On success NIL is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

10.4. helper.startup

This method is used to setup necessary configuration during startup. This includes all configuration except network configuration, which is handled in `helper.netup`.

Synopsis

```
helper.startup(int xid)
```

xid Unique context ID

Access Restrictions

This method requires the `HELPER` capability and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

11. Daemon Administration Methods

11.1. vcd.login

This method is used to authenticate against the user database. Internally it is a no-op method, since every method call must be authenticated. This is only usefull for GUIs or web panels, to let users login without actually doing something.

Synopsis

```
vcd.login()
```

Access Restrictions

This method requires no capabilities and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

11.2. vcd.status

This method is used to retreive internal daemon statistics. The daemon stores a lot of runtime information in VXDB that can be used by system administrators or user to check the daemon or account healthiness respectively.

Synopsis

```
vcd.status()
```

Access Restrictions

This method requires the `INFO` capability and does not need to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

11.3. vcd.user.caps.add

This method is used to add capabilities to the internal user database.

Synopsis

```
vcd.user.caps.add(string username, string cap)
```

username Unique username

cap Capability to add

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

11.4. vcd.user.caps.get

This method is used to get information about configured capabilities in the internal user database.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vcd.user.caps.get(string username)
```

username Unique username

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success an `array` of `strings` - on for each capability - is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

11.5. vcd.user.caps.remove

This method is used to remove information about configured capabilities from the internal user database.

Synopsis

```
vcd.user.caps.remove(string username, string cap)
```

username Unique username

cap Capability to remove

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Errors

This method does not return any errors beside the generic method errors.

11.6. vcd.user.get

This method is used to get information about configured users in the internal user database.

Synopsis

```
vcd.user.get(string username)
```

username Unique username

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success a `struct` with the following signature is returned, on error a fault notification is returned.

```
{string password, bool admin}
```

password Password hash for the specified user

admin Set the administrator flag for the specified user

11.7. vcd.user.remove

This method is used to remove information about configured users from the internal user database.

Synopsis

```
vcd.user.remove(string username)
```

username Unique username

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

11.8. vcd.user.set

This method is used to set (add & change) information about configured users in the internal user database.

Synopsis

```
vcd.user.set(string username, string password, bool admin)
```

username Unique username

password Password for the specified user (if user already exists this value may be empty to not change the password)

admin Set the administrator flag for the specified user

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

12. General Maintenance Methods

The vx family of functions provide general maintenance facilities for virtual servers.

12.1. vx.create

This method is used to create a new virtual server using a template cache.

Synopsis

```
vx.create(string name, string template, bool force, bool copy, string
vdir)
```

name Unique name to use for the new virtual server

template Name of the template cache to use for the new virtual server

rebuild If the virtual server already exists and this is set to true a recursive unlink is performed before the template extraction takes place

Access Restrictions

This method requires the `CREATE` capability and needs to pass owner checks. The administrator passes owner check even if the specified virtual server does not exist, in order to be able to create new ones. This differs slightly from the normal owner check behaviour.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MEINVAL An empty template cache name was supplied, or the supplied value contained invalid characters

MEEXIST The virtual server already exists and the rebuild parameter was not set to true

MERUNNING The virtual server already exists, the rebuild parameter was set to true, but the existing virtual server is still running

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

12.2. vx.exec

This method is used to execute a simple command inside a virtual server. This method does not support redirection, pipes and other shell stuff, the command line is executed with a plain `execvp`.

Synopsis

```
vx.exec(string name, string command)
```

name Unique virtual server name

command Simple command line to execute

Access Restrictions

This method requires the `EXEC` capability and needs to pass owner checks.

Return Value

On success a combined string of `STDOUT` and `STDERR` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MESTOPPED The specified virtual server is currently not running, therefore the command cannot be executed

12.3. vx.kill

This method is used to send signals to processes inside a virtual server.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vx.kill(string name, int pid, int sig)
```

name Unique virtual server name

pid Process ID to send signal to, there are two special cases:
0 = send signal to all processes
1 = send signal to the real pid of a faked init

sig Signal Number, see the `signal(7)` man page for more information

Access Restrictions

This method requires the `INIT` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MESTOPPED The specified virtual server is currently not running, therefore no signals can be sent

12.4. vx.load

This method is used to receive the current load average, i.e. the number of jobs in the run queue (state R) or waiting for disk I/O (state D) averaged over 1, 5, and 15 minutes.

Synopsis

```
vx.load(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `INFO` capability and needs to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success a `struct` with the following signature is returned, on error a fault notification is returned.

```
{string 1m, string 5m, string 15m}
```

1m Load average in the last minute

5m Load average in the last 5 minutes

15m Load average in the last 15 minutes

Errors

This method does not return any errors beside the generic method errors.

12.5. vx.reboot

This method is used as a wrapper to vx.exec using the configured `reboot` command.

Synopsis

```
vx.reboot(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `INIT` capability and needs to pass owner checks.

Return Value

On success the return value of `vx.exec` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MESTOPPED The specified virtual server is currently not running, therefore it cannot be stopped

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

12.6. vx.remove

This method is used to remove a virtual server from the configuration database and filesystem.

Synopsis

```
vx.remove(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `CREATE` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MERUNNING The specified virtual server is currently running, therefore it cannot be removed

MEEXIST The root filesystem fo the virtual server still exists after removal and no mount point exists for it. Manual check is needed.

12.7. vx.rename

This method is used to rename a virtual server in the configuration database. This does not involve filesystem changes, in particular, the virtual servers root filesystem is not moved or renamed.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vx.rename(string name, string newname)
```

name Unique virtual server name

newname New unique virtual server name

Access Restrictions

This method requires the `CREATE` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MERUNNING The specified virtual server is currently running, therefore it cannot be renamed

MEEXIST The specified new virtual server name currently exists and cannot be used

12.8. vx.restart

This method is used to restart virtual servers. It has the same effect as `vx.reboot` except that it waits for the virtual server to stop and startup again, which cannot be acomplished by the former. Technically, this is a combination of `vx.stop` and `vx.start` only.

Synopsis

```
vx.restart(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `INIT` capability and needs to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MESTOPPED The specified virtual server is currently not running, therefore it cannot be stopped

12.9. vx.start

This method is used to start a virtual server.

Synopsis

```
vx.start(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `INIT` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MERUNNING The specified virtual server is currently running, therefore it cannot be started again

12.10. vx.status

This method is used to obtain status information about a virtual server.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vx.status(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `INIT` capability and needs to pass owner checks.

Return Value

On success `struct` with the following signature is returned, on error a fault notification is returned.

```
{bool running, int nproc, int uptime}
```

running This value is set to true if the specified virtual server is currently running, false otherwise

nproc This value is set to the number of processes inside the specified virtual server, if it is running, 0 otherwise

uptime This value is set to the uptime in seconds of the specified virtual server, if it is running, 0 otherwise

Errors

This method does not return any errors beside the generic method errors.

12.11. vx.stop

This method is used as a wrapper to vx.exec using the configured `halt` command.

Synopsis

```
vx.stop(string name)
```

name Unique virtual server name

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Access Restrictions

This method requires the `INIT` capability and needs to pass owner checks.

Return Value

On success the return value of `vx.exec` is returned, on error a fault notification is returned.

Errors

Beside the generic method errors, this method may return the following specific errors:

MESTOPPED The specified virtual server is currently not running, therefore it cannot be stopped

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

13. Database Manipulation Methods

The `vxdb` family of functions provide manipulation facilities for the configuration database.

13.1. vxdb.dx.limit.get

This method is used to get information about configured disk limits.

Synopsis

```
vxdb.dx.limit.get(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `DLIM` capability and needs to pass owner checks.

Return Value

On success a `struct` with the following signature is returned, on error a fault notification is returned.

```
{uint32 space, uint32 inodes, int reserved}
```

space Maximum amount of disk space for this virtual server in KB

inodes Maximum number of inodes for this virtual server

reserved Disk space reserved for the super-user (root) *inside* the virtual server in percent

Errors

This method does not return any errors beside the generic method errors.

13.2. vxdb.dx.limit.remove

This method is used to remove information about configured disk limits.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vxdb.dx.limit.remove(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `DLIM` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.3. vxdb.dx.limit.set

This method is used to set (add & change) information about configured disk limits.

Synopsis

```
vxdb.dx.limit.set(string name, uint32 space, uint32 inodes, int reserved)
```

name Unique virtual server name

space Maximum amount of disk space for this virtual server in KB

inodes Maximum number of inodes for this virtual server

reserved Disk space reserved for the super-user (root) *inside* the virtual server in percent

Access Restrictions

This method requires the `DLIM` capability and needs to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.4. vxdb.init.get

This method is used to get information about configured `init`, `halt` and `reboot` commands.

Synopsis

```
vxdb.init.get(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `INIT` capability and needs to pass owner checks.

Return Value

On success a `struct` with the following signature is returned, on error a fault notification is returned.

```
{string init, string halt, string reboot}
```

init Absolute path of the init command *inside* the virtual server (defaults to `/sbin/init` if empty)

halt Absolute path of the halt command *inside* the virtual server (defaults to `/sbin/halt` if empty)

reboot Absolute path of the reboot command *inside* the virtual server (defaults to `/sbin/reboot` if empty)

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Errors

This method does not return any errors beside the generic method errors.

13.5. vxdb.init.set

This method is used to set (change) information about configured `init`, `halt` and `reboot` commands.

Synopsis

```
vxdb.init.set(string name, string init, string halt, string reboot)
```

name Unique virtual server name

init Absolute path of the init command *inside* the virtual server (defaults to `/sbin/init` if empty)

halt Absolute path of the halt command *inside* the virtual server (defaults to `/sbin/halt` if empty)

reboot Absolute path of the reboot command *inside* the virtual server (defaults to `/sbin/reboot` if empty)

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.6. vxdb.list

This method is used to get a list of all currently configured and owned virtual servers.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vxdb.list(string username)
```

username Unique username (this value may be empty to return a list containing all virtual servers)

Access Restrictions

This method requires no capabilities and needs to pass owner checks. The owner checks do not check the usual `name` parameter here, instead it returns all virtual servers that would pass owner checks.

Return Value

On success an `array` of `strings` – one for each virtual server – is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.7. vxdb.mount.get

This method is used to get information about configured mount points.

Synopsis

```
vxdb.mount.get(string name, string dst)
```

name Unique virtual server name

dst Absolute destination path for the mount point inside the virtual server (this value may be empty to return a list of all configured mount points)

Access Restrictions

This method requires the `MOUNT` capability and needs to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success an `array` of `structs` – one for each mount point – with the following signature is returned, on error a fault notification is returned.

```
{string src, string dst, string type, string opts}
```

src Absolute source path *inside* the virtual server or arbitrary identifier (defaults to `none` if empty)

dst Absolute destination path for the mount point inside the virtual server

type Filesystem type for the specified source path (defaults to `auto` if empty)

opts Mount options for the specified filesystem (defaults to `defaults` if empty)

Errors

This method does not return any errors beside the generic method errors.

13.8. vxdb.mount.remove

This method is used to remove information about configured mount points.

Synopsis

```
vxdb.mount.remove(string name, string dst)
```

name Unique virtual server name

dst Absolute destination path for the mount point inside the virtual server (this value may be empty to remove all configured mount points)

Access Restrictions

This method requires the `MOUNT` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Errors

This method does not return any errors beside the generic method errors.

13.9. vxdb.mount.set

This method is used to set (add & change) information about configured mount points.

Synopsis

```
vxdb.mount.set(string name, string src, string dst, string type, string
opts)
```

name Unique virtual server name

src Absolute source path *inside* the virtual server or arbitrary identifier (defaults to `none` if empty)

dst Absolute destination path for the mount point inside the virtual server

type Filesystem type for the specified source path (defaults to `auto` if empty)

opts Mount options for the specified filesystem (defaults to `defaults` if empty)

Access Restrictions

This method requires the `MOUNT` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.10. vxdb.name.get

This method is used to lookup the name of a virtual server by its corresponding context ID.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vxdb.name.get(int xid)
```

xid Unique context ID

Access Restrictions

This method requires the INFO capability and does not need to pass owner checks.

Return Value

On success a string containing the virtual server name is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.11. vxdb.nx.addr.get

This method is used to get information about configured network addresses.

Synopsis

```
vxdb.nx.addr.get(string name, string addr)
```

name Unique virtual server name

addr Network address in dot-decimal form (this value may be empty to return a list of all configured network adresses)

Access Restrictions

This method requires the NET capability and needs to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success an `array` of `structs` – one for each network address – with the following signature is returned, on error a fault notification is returned.

```
{string addr, string netmask}
```

addr Network address in dot-decimal form

netmask Network mask in dot-decimal form (defaults to `255.255.255.0` if empty)

Errors

This method does not return any errors beside the generic method errors.

13.12. vxdb.nx.addr.remove

This method is used to remove information about configured network addresses.

Synopsis

```
vxdb.nx.addr.remove(string name, string addr)
```

name Unique virtual server name

addr Network address in dot-decimal form (this value may be empty to remove all configured network adresses)

Access Restrictions

This method requires the `NET` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

13.13. vxdb.nx.addr.set

This method is used to set (add & change) information about configured network addresses.

Synopsis

```
vxdb.nx.addr.set(string name, string addr, string netmask)
```

name Unique virtual server name

addr Network address in dot-decimal form

netmask Network mask in dot-decimal form (defaults to `255.255.255.0` if empty)

Access Restrictions

This method requires the `NET` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.14. vxdb.nx.broadcast.get

This method is used to get information about the configured broadcast address.

Synopsis

```
vxdb.nx.broadcast.get(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `NET` capability and needs to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success a `string` containing the broadcast address is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.15. vxdb.nx.broadcast.remove

This method is used to remove information about the configured broadcast address.

Synopsis

```
vxdb.nx.broadcast.remove(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `NET` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.16. vxdb.nx.broadcast.set

This method is used to set (change) information about the configured broadcast address.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vxdb.nx.broadcast.set(string name, string broadcast)
```

name Unique virtual server name

broadcast Broadcast address in dot-decimal form

Access Restrictions

This method requires the `NET` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.17. vxdb.owner.add

This method is used to add users to the list of configured owners in the internal user database.

Synopsis

```
vxdb.owner.add(string name, string username)
```

name Unique virtual server name

username Unique username

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Errors

This method does not return any errors beside the generic method errors.

13.18. vxdb.owner.get

This method is used to get information about configured owners in the internal user database.

Synopsis

```
vxdb.owner.get(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success an `array` of `strings` – one for each owner – is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.19. vxdb.owner.remove

This method is used to remove information about configured owners from the internal user database.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vxdb.owner.remove(string name, string username)
```

name Unique virtual server name

username Unique username (this value may be empty to remove all configured owners)

Access Restrictions

This method requires the `AUTH` capability and does not need to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.20. vxdb.vdir.get

This method is used to get the absolute root filesystem path of a virtual server.

Synopsis

```
vxdb.vdir.get(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `INFO` capability and does not need to pass owner checks.

Return Value

On success a `string` containing the absolute root filesystem path is returned, on error a fault notification is returned.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Errors

This method does not return any errors beside the generic method errors.

13.21. vxdb.vx.bcaps.add

This method is used to add information about configured system capabilities.

Synopsis

```
vxdb.vx.bcaps.add(string name, string bcap)
```

name Unique virtual server name

bcap System capability to add

Access Restrictions

This method requires the `BCAP` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.22. vxdb.vx.bcaps.get

This method is used to get information about configured system capabilities.

Synopsis

```
vxdb.vx.bcaps.get(string name)
```

name Unique virtual server name

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Access Restrictions

This method requires the `BCAP` capability and needs to pass owner checks.

Return Value

On success an `array` of `strings` – one for each system capability – is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.23. vxdb.vx.bcaps.remove

This method is used to remove information about configured system capabilities.

Synopsis

```
vxdb.vx.bcaps.remove(string name, string bcap)
```

name Unique virtual server name

bcap System capability to remove

Access Restrictions

This method requires the `BCAP` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

13.24. vxdb.vx.ccaps.add

This method is used to add information about configured context capabilities.

Synopsis

```
vxdb.vx.ccaps.add(string name, string ccap)
```

name Unique virtual server name

ccap Context capability to add

Access Restrictions

This method requires the `CCAP` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.25. vxdb.vx.ccaps.get

This method is used to get information about configured context capabilities.

Synopsis

```
vxdb.vx.ccaps.get(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `CCAP` capability and needs to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success an `array` of `strings` – one for each context capability – is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.26. vxdb.vx.ccaps.remove

This method is used to remove information about configured context capabilities.

Synopsis

```
vxdb.vx.ccaps.remove(string name, string ccap)
```

name Unique virtual server name

ccap Context capability to remove

Access Restrictions

This method requires the `CCAP` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.27. vxdb.vx.flags.add

This method is used to add information about configured context flags.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vxdb.vx.flags.add(string name, string flag)
```

name Unique virtual server name

flag Context flag to add

Access Restrictions

This method requires the CFLAG capability and needs to pass owner checks.

Return Value

On success NIL is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.28. vxdb.vx.flags.get

This method is used to get information about configured context flags.

Synopsis

```
vxdb.vx.flags.get(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the CFLAG capability and needs to pass owner checks.

Return Value

On success an array of strings – one for each context flag – is returned, on error a fault notification is returned.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Errors

This method does not return any errors beside the generic method errors.

13.29. vxdb.vx.flags.remove

This method is used to remove information about configured context flags.

Synopsis

```
vxdb.vx.flags.remove(string name, string flag)
```

name Unique virtual server name

flag Context flag to remove

Access Restrictions

This method requires the `CFLAG` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.30. vxdb.vx.limit.get

This method is used to get information about configured resource limits.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Synopsis

```
vxdb.vx.limit.get(string name, string limit)
```

name Unique virtual server name

limit Limit type to get (this value may be empty to retrieve all configured resource limits)

Access Restrictions

This method requires the `RLIM` capability and needs to pass owner checks.

Return Value

On success an `array` of `structs` – one for each resource limit – with the following signature is returned, on error a fault notification is returned.

```
{string limit, uint64 soft, uint64 max}
```

limit Limit type to add or change

soft Softl imit for the specified type

max Hard limit for the specified type

Errors

This method does not return any errors beside the generic method errors.

13.31. vxdb.vx.limit.remove

This method is used to remove information about configured resource limits.

Synopsis

```
vxdb.vx.limit.remove(string name, string limit)
```

name Unique virtual server name

limit Limit type to remove (this value may be empty to remove all configured resource limits)

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Access Restrictions

This method requires the RLIM capability and needs to pass owner checks.

Return Value

On success NIL is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.32. vxdb.vx.limit.set

This method is used to set (add & change) information about configured resource limits.

Synopsis

```
vxdb.vx.limit.set(string name, string limit, uint64 soft, uint64 max)
```

name Unique virtual server name

limit Limit type to add or change

soft Softl imit for the specified type

max Hard limit for the specified type

Access Restrictions

This method requires the RLIM capability and needs to pass owner checks.

Return Value

On success NIL is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

13.33. vxdb.vx.sched.get

This method is used to get information about configured CPU scheduler buckets.

Synopsis

```
vxdb.vx.sched.get(string name, int cpuid)
```

name Unique virtual server name

cpuid CPU ID as listed in /proc/cpuinfo

Access Restrictions

This method requires the SCHED capability and needs to pass owner checks.

Return Value

On success an array of structs – one for each CPU ID – with the following signature is returned, on error a fault notification is returned.

```
{int cpuid, int interval, int fillrate, int interval2, int fillrate2,
int tokensmin, int tokensmax}
```

cpuid CPU ID as listed in /proc/cpuinfo

interval Interval between fills in jiffies

fillrate Tokens to fill each interval

interval2 Interval between fills in jiffies (IDLE time setting)

fillrate2 Tokens to fill each interval (IDLE time setting)

tokensmin Minimum number of tokens to schedule processes

tokensmax Maximum number of tokens in the bucket

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

13.34. vxdb.vx.sched.remove

This method is used to remove information about configured CPU scheduler buckets.

Synopsis

```
vxdb.vx.sched.remove(string name, int cpuid)
```

name Unique virtual server name

cpuid CPU ID as listed in `/proc/cpuinfo`

Access Restrictions

This method requires the `SCHED` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.35. vxdb.vx.sched.set

This method is used to set (add & change) information about configured CPU scheduler buckets.

Synopsis

```
vxdb.vx.sched.set(string name, int cpuid, int interval, int fillrate,
int interval2, int fillrate2, int tokensmin, int tokensmax)
```

name Unique virtual server name

cpuid CPU ID as listed in `/proc/cpuinfo`

interval Interval between fills in jiffies

fillrate Tokens to fill each interval

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

interval2 Interval between fills in jiffies (IDLE time setting)

fillrate2 Tokens to fill each interval (IDLE time setting)

tokensmin Minimum number of tokens to schedule processes

tokensmax Maximum number of tokens in the bucket

Access Restrictions

This method requires the `SCHED` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.36. vxdb.vx.uname.get

This method is used to get information about configured virtual system information.

Synopsis

```
vxdb.vx.uname.get(string name, string uname)
```

name Unique virtual server name

uname System information type (this value may be empty to retrieve information about all configured system information types)

Access Restrictions

This method requires the `UNAME` capability and needs to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success an array of structs – one for each system information type – with the following signature is returned, on error a fault notification is returned.

```
{string uname, string value}
```

uname System information type

value System information value

Errors

This method does not return any errors beside the generic method errors.

13.37. vxdb.vx.uname.remove

This method is used to remove information about configured virtual system information.

Synopsis

```
vxdb.vx.uname.remove(string name, string uname)
```

name Unique virtual server name

uname System information type

Access Restrictions

This method requires the UNAME capability and needs to pass owner checks.

Return Value

On success NIL is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

13.38. vxdb.vx.uname.set

This method is used to set (add & change) information about configured virtual system information.

Synopsis

```
vxdb.vx.uname.set(string name, string uname, string value)
```

name Unique virtual server name

uname System information type

value System information value to set

Access Restrictions

This method requires the `UNAME` capability and needs to pass owner checks.

Return Value

On success `NIL` is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

13.39. vxdb.xid.get

This method is used to lookup the Context ID of a virtual server by its name.

Synopsis

```
vxdb.xid.get(string name)
```

name Unique virtual server name

Access Restrictions

This method requires the `INFO` capability and does not need to pass owner checks.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Return Value

On success an `int` containing the Context ID of the specified virtual server is returned, on error a fault notification is returned.

Errors

This method does not return any errors beside the generic method errors.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License

Bibliography

[1] Andi Mann. Virtualization 101: Technologies, benefits, and challanges. Technical report, Enterprise Management Associates, 2006. http://www.emausa.com/web/EMA_Virtualization_WP-0806.pdf.

[2] Amit Singh. An introduction to virtualization, 1994-2006. http://www.kernelthread.com/publications/virtualization/.

[3] Wikipedia. Instruction set simulator – wikipedia, the free encyclopedia, Feb 2007. http://en.wikipedia.org/wiki/Instruction_Set_Simulator.

[4] B. Jack Copeland. The church-turing thesis. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2002.

[5] Ed Thelen. Emulation/simulation, 2007. http://ed-thelen.org/comp-hist/emulation.html.

[6] Wikipedia. Instruction set simulator – wikipedia, the free encyclopedia, Feb 2007. http://en.wikipedia.org/wiki/Native_virtualization.

[7] Andrew Whitaker/Marianne Shaw/Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, University of Washington, 2002. http://denali.cs.washington.edu/pubs/distpubs/papers/denali_usenix2002.pdf.

[8] Wikipedia. Instruction set simulator – wikipedia, the free encyclopedia, 2007. http://en.wikipedia.org/wiki/Operating_system-level_virtualization.

[9] Benedikt Böhm. libvserver api documentation, 2007. http://people.linux-vserver.org/~hollow/libvserver/doc.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License