Virtual Square

Users, Programmers & Developers Guide

Renzo Davoli, Michael Goldweber Editors

Contributions by:

Diego Billi, Federica Cenacchi, Renzo Davoli, Ludovico Gardenghi, Andrea Gasparini, Michael Goldweber

The Virtual Square Team

Introduction

Virtual Square is a container of projects about virtuality.

The word "virtual" has been overused and misused, everything related to computers and networks sounds virtual.

Computer Science define abstractions and interfaces. These two key concepts are strictly related. An abstraction defines the semantics of operations while an interface is the syntax required to access the operations defined by the abstraction. Programs and human users use interfaces to ask for the actions defined by an abstraction.

Virtuality means providing equivalent abstractions, providing the same interface, such that the users (programs or humans) can effectively use the virtual abstraction instead of the *real* one.

For example, a file system is an abstraction providing an application programming interface (API) composed by several calls like `open`, `read`, `close`. A virtual file system is an abstraction providing the same interface, such that the programs using the file system can use the virtual file system too. At the same time a virtual file system can apply the same abstraction to different domains not necessarily related to store and retrieve data on magnetic disks.

The main memory is an abstraction, too. The hardware, memory cells arrays and MMU, provides the programs with an interface based on two main operations `load` and `store`. A virtual memory provides the same interface while uses a mix of main and secondary memory to store data. Programs use virtual memory effectively instead of the main memory.

An entire computer hardware, a "machine" is perceived by the operating system as an abstraction. The interface is composed by the processor instruction set and by the set of bus addresses, registers and commands required to interoperate with peripheral controllers. Another abstraction, maybe a program, able to provide the same interface to the operating system is properly defined virtual machine.

The same definition applies to virtual networks, virtual devices, virtual hard disks.

We perceive the world, the reality, through our senses. Thus it is an *abstraction* for us, and the interface is made of light, colors, sounds, etc. The definition of virtual reality is consistent with our definition, in fact in what is commonly named as "virtual reality" our senses gets connected to devices that are able to provide the same interface of light, colors, sounds.

Virtuality becames in this way a powerful tool for interoperability, virtual entities can act as puzzle tiles or building blocks to provide programs with suitable interfaces and services.

This is a Virtual Square, a virtual place where different abstractions can interoperate. It is possible to read it also as a Virtual Squared, i.e. how to exploit existing virtualities to build up further virtual services (this is the meaning of Virtual Square logo, V^2)

Virtual Square is a set of different projects sharing the idea of exploit virtuality by unifying concepts and by creating tools for interoperability.

Today Virtual Square is also an international laboratory on virtuality ran by a research and development team. It started in 2004 at the University of Bologna, Italy.

The research of Virtual Square involves several aspects of virtualization. Virtual Distributed Ethernet is the V^2 Virtual Networking project. VDE is a Virtual Ethernet, whose nodes can be distributed across the real Internet. The idea of VDE sums up VPN, tunnel, Virtual Machines interconnection, overlay networking, as all these different entities can be implemented by VDE.

View-OS is the V^2 project about operating systems. The main idea is to negate the global view assumption. An operating system should provide services to a process without forcing all the processes to have its own unique *view* of the execution environment. File Systems, Networking, Device Drivers, Users, System id, can be defined or redefined at process level.

This revolutionary view on virtuality has led to a better understanding of the limits of current implementations of operating systems structure and implementation, networking stacks and interfaces, C library support. V^2 extends the Linux kernel support for virtuality and inter-process communication, implements the networking stack as a library and add the support of multiple stacks to the Berkeley Socket interface, provide self virtualization for processes and libraries by adding features to the C library. All these enhancements preserve backward compatibility with existing applications.

The description of a live research project like V^2 is like to take a snapshot of something which is rapidly evolving. Your V^2 could be different from the one here explained, maybe older because the mantainer of the tools for your Linux distribution has been late in updating the software. Typically your V^2 will have more features than the one here described, and maybe items here listed as future developments will be already included in the code at the time you read this book. The first version of this book took about three years, and several sections have been written several times for the natural evolution of the projects. We suggest to use this document to have a complete view of the project and an analysis of its ideas and tools and we ask the reader to refer to the wiki of the project `http://wiki.virtualsquare.org` for updates.

Comments, errata corrige, suggestions, bugreport and bugfixes are welcome. Researchers and developers can be reached on the IRC public forum `irc.freenode.net#virtualsquare` or using the mail addresses of the editors `renzo@cs.unibo.it` and `mikeyg@cs.xu.edu`.

This book describes the entire project including consolidated concepts like vde, or umview and young and evolving ideas like ipn or kmview. For this reason, the reader will find some tools already included in major linux distributions while others must be downloaded as source code, compiled and installed.

Renzo Davoli, Michael Goldweber

Notation

This book uses icons to describe the intended audience of each section. Icons appear as prefixes in the title and in the table of contents.

no icon Description of general ideas about the project.

★ User guide: these sections are for users of virtual square tools.

■ Programmer guide: these sections are for programmers who need to interface their programs to virtual square libraries or servers.

▲ Developer guide: these sections are for programmers aiming to develop modules or plugin for virtual square tools and libraries.

♦ Internals: these sections describe the design and implementation of virtual square libraries and tools. ♦sections are for developers aiming to contribute to V^2.

▼ Education resource: virtual square provide valuable tool for education. The sections tagged by ▼provide ideas and suggestions about using V^2 to teach computer science.

Contents

Introduction **i**

Contents **v**

List of Figures **ix**

I The Big Picture **1**

1 Virtualization and Virtual Machines **3**
1.1 Introduction to Virtual Machines 3
1.2 Virtuality, Emulation and Simulation 4
1.3 Brief history of virtuality 5
1.4 Classification 6
1.5 Emulators/Heterogeneous virtual machines 8
1.6 Homogeneous virtual machines 10
1.7 Operating System-Level Virtualization 12
1.8 Process level virtual machine. 13
1.9 Process level partial virtualization 15
1.10 Microkernel systems 20

2 V^2: The Virtual Square Framework **21**
2.1 Introduction to Virtual Square 21
2.2 V^2 goals and guidelines 22
2.3 V^2 components 23

3 What's new in Virtual Square **27**
3.1 ★VDE: a swiss-knife for virtual networking 27
3.2 ■msockets: Multi stack support for Berkeley Sockets 28
3.3 ■IPv6 hybrid stacks for IPv4 backward compatibility 30
3.4 ■What a process views 31
3.5 ★Partial Virtual Machines 33
3.6 ■Microkernels and Monolithic kernels are not mutually exclusive 34
3.7 ■Inter Process Networking: the need for multicast IPC 35

II Virtual Square Networking **37**

4 VDE: Virtual Distributed Ethernet **39**
4.1 ★VDE Main Components 40

4.2 ★VDE Connectivity Tools . . . 41
4.3 ★VDE: A Closer Look . . . 43
4.4 ★VDE Examples . . . 55
4.5 ■VDE API: The `vdeplug` Library . . . 58
4.6 ★`VDEtelweb` . . . 59
4.7 ▲Plugin Support for VDE Switches . . . 60
4.8 ♦`vde_switch` Internals . . . 65
4.9 ▼VDE in Education . . . 69

5 LWIPv6 **71**
5.1 ■LWIPv6 API . . . 72
5.2 ■An LWIPv6 tutorial . . . 72
5.3 ♦LWIPv6 Internals . . . 81
5.4 ▼LWIPv6 in education . . . 96

6 Inter Process Networking **99**
6.1 ■IPN usage . . . 100
6.2 ★Compile and install IPN . . . 104
6.3 ■IPN usage examples . . . 105
6.4 ★kvde_switch, a VDE switch based on IPN . . . 108
6.5 ▲IPN protocol submodules . . . 110
6.6 ♦IPN internals . . . 114
6.7 IPN applications . . . 116
6.8 ▼IPN in education . . . 116

III View-OS **117**

7 View-OS **119**
7.1 The global view assumption and its drawbacks . . . 120
7.2 The ViewOS idea . . . 121
7.3 Goals and applications . . . 122
7.4 ViewOS implementation . . . 123

8 Pure_libc **125**
8.1 ■Pure_Libc API . . . 125
8.2 ■Pure_Libc Tutorial . . . 127
8.3 ♦Pure_Libc design choices . . . 130
8.4 ♦Pure_Libc internals . . . 130
8.5 ▼Pure_Libc in education . . . 132

9 *MView **133**
9.1 *MView file system management . . . 134
9.2 *MView modularity . . . 136
9.3 ★*MView basic usage . . . 136
9.4 ▲*MView modules . . . 140
9.5 ♦UMview internals . . . 147
9.6 ♦UMview patches for the Linux Kernel . . . 162
9.7 ♦KMview internals . . . 168
9.8 ▼*MView in education . . . 177

10 *MView modules **179**
10.1 ★umnet: virtual multi stack support . . . 179
10.2 ▲How to write umnet submodules . . . 185
10.3 ♦umnet internals . . . 185
10.4 ★umfuse: virtual file systems support . . . 186
10.5 ★Some third parties umfuse submodules . . . 189
10.6 ▲How to write umfuse submodules . . . 190
10.7 ♦umfuse internals . . . 192
10.8 ♦Some notes on umfuse modules internals . . . 193
10.9 ★umdev: virtual devices support . . . 193
10.10 ▲How to write umdev submodules . . . 196
10.11 ♦umdev internals . . . 196
10.12 ★umbinfmt: interpreters and foreign executables support . . . 197
10.13 ★ummisc: virtualization of time, system name, etc. . . . 198
10.14 ▲How to write ummisc submodules . . . 200
10.15 ♦ummisc internals . . . 203
10.16 ★ViewFS . . . 204
10.17 ♦ViewFS internals . . . 209
10.18 ★Umview/kmview as login shells . . . 209
10.19 ▼*MVIEW modules in education . . . 211

IV and they lived happily ever after... **213**

11 Conclusions and future developments **215**
11.1 Conclusions . . . 215
11.2 Acknowledges . . . 217

The V^2 Team **218**

A GNU Free Documentation License **221**

Bibliography **229**

List of Figures

1.1 A pictorial view of Virtuality 4
1.2 Process Virtual Machine vs. System Virtual Machine 7

2.1 Virtual Square tools and libraries 24

3.1 Kernel's and Process' views on the same world 32

4.1 Comparison between a real Ethernet and a VDE 40
4.2 VDE plug (point-to-point) 41
4.3 VDE command line interface options. Note: The actual command line interface set may differ depending upon the version of the tool and options enabled during the compilation. 45
4.4 `vdeplug` Library: libvdeplug.h 59
4.5 Vdeplug stream encoding functions: libvdeplug.h 59
4.6 `VDEtelweb`: A sample telnet session 60
4.7 `VDEtelweb`: A sample web session 61
4.8 A minimal VDE plugin 62
4.9 VDE built-in events 64
4.10 VDE dump.c plugin 66
4.11 Conversions between untagged to untagged packets and viceversa . 68

5.1 LWIPv6 API: Interface definition 73
5.2 LWIPv6 API: socket library and I/O 74
5.3 The source code of `lwipnc.c` 79
5.4 The source code of `tinyrouter.c` 80
5.5 LWIP architecture 82
5.6 LWIP and the other components of an operating system 82
5.7 Pbuf chain 87
5.8 Pbuf header segments 88
5.9 Structure of a netif driver 89
5.10 LWIP IP layers 91
5.11 Headers for packet fragmentation 91
5.12 Reassembly of a packet 92
5.13 Function calls in IP packet reassembly 92
5.14 Function involved in UDP packets management 94
5.15 Sequence of calls (limited to the management of the IP protocol) . 97
5.16 Interactions between an application and the LWIPv6 library 98

6.1 minihub.c: A minimal VDE hub based on IPN. 109

6.2 MPEG TS policy submodule: mpegts_main.c 113
6.3 IPN: Data stuctures used for message queuing 114

8.1 PURE_LIBC API (pure_libc.h) 126
8.2 A first PureLibc example. 127
8.3 A PureLibc example using dynamic loading 128
8.4 A simple virtualization based on PureLibC 129

9.1 The mount metaphore 134
9.2 Composition of Partial Virtual Machines 136
9.3 A simple "hello world" *MView module 141
9.4 Source code of abitmore.c: a module that changes the nodename . 146
9.5 Example of ctl event notification function. 148
9.6 Design of UMview Implementation 149
9.7 UMview nested invocation and (simplified) treepoch timestamping, symbols in the first column are reference to Figure 9.8 156
9.8 Treepoch timestamping: tree structure 157
9.9 System Call management: syscall argument substitution 157
9.10 *mview specific system calls 160
9.11 Structure of pcb.h file 163
9.12 The code to probe PTRACE features supported by the current kernel 167
9.13 Some executions of test_ptracemulti 168
9.14 A minimal tracer based on the KMview kernel module 173

10.1 umnetnull: The null network submodule for umnet 186
10.2 Use of the mount option *except* of umfuse 189
10.3 A common use of the option *except* 189
10.4 user_data argument use for fuse/umfuse compatibility 191
10.5 Umfuse: Program to dynamic library run-time conversion for fuse modules 192
10.6 umdevnull.c: the simplest virtual device 197
10.7 miscdata.c: a misc submodule with a virtual file system for parameters 201
10.8 fakeroot.c: a simple misc submodule 203

Part I

The Big Picture

CHAPTER 1

Virtualization and Virtual Machines

1.1 Introduction to Virtual Machines

What does *"virtual"* mean? The word is used in a great variety of meanings, but the most general one refers to something that doesn't physically exist, but *appears real*.

In the Computer Science world, defining *what* is virtual is not so straightforward, being the software itself a non-physical entity. An abstraction for computer science is a software tool that is useful for some specific task. For example libraries provide abstractions. Each abstraction has an interface. In computer science when there is a well known abstraction A accessible by an interface $I(A)$, a new abstraction A' providing the same interface, i.e. $I(A') = I(A)$, that can be effectively used instead of A can be named A *virtual*.

In figure 1.1 an abstraction A is depicted as a hammer. This well known tool has an interface to the lower layer (the flat surface to hit the nail) and an interface to the upper layer (a handle). If a new tool A' maybe having a completely different shape, has the same interface and can be effectively used instead of A, A' can be named a virtual A. Sometimes, when there is nothing better to use, a shoe can be a virtual hammer. However, Hammers are hardware tools.

When dealing with software tools it is quite common that A' uses A for its implementation.

Virtual Machines are well known examples of virtual software entities. Following the general introduction, a fairly correct definition of *Virtual Machine* may be "a software layer that provides an environment, between a user and a computer, able to run other software".

Nowadays Virtual Machines are widely adopted for a vast range of applications: programming languages (e.g. Java), web hosting services, multi-operating system servers. Aside Virtual Machines, there are other tools that exploit the concept of *virtuality*, like the Virtual Memory in the majority of

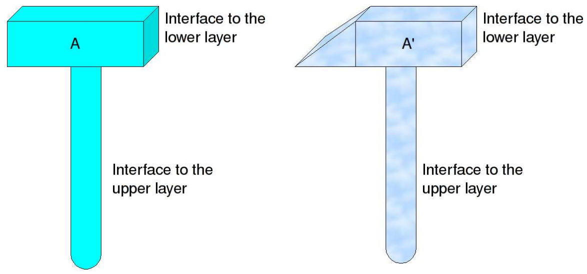

Figure 1.1: A pictorial view of Virtuality

modern operating systems: it provides an abstraction of a large memory area, reserving a portion of the disk to be used in addition to the available main memory. The processes use the Virtual Memory provided by the kernel as it were *real* main memory.

Basically, virtual machines implement the idea of *abstraction* [27]: virtualization tools are usually a set of abstraction layers, one above the other. Each level hides the underlying level's details, and provides to the upper level a higher and more abstract vision of the system. Applying this concept to virtual machines can lead to a turnover of the concept itself: in many cases the goal of a virtual machine is to emulate low abstraction layers for use in higher layers (e.g. hardware abstraction).

A virtualization tool can either add an abstraction layer to the existent ones, or "redefine" a layer modifying its semantics. In the majority of cases, each level has to interact only with the immediately underlying layer's interface; however, in some cases it's possible to bypass one or more layers in order to access directly low layers.

In the light of these considerations, prior to introducing the Virtual Square Framework, it's very important to understand the meaning of "virtualization", to see the different types of virtual machines, and to find a classification criterion for them.

1.2 Virtuality, Emulation and Simulation

The basic idea about virtualization is that for any X the virtual X can be effectively used instead of X.

A simulator is a device that mimics another environment. A flight simulator is a program designed to behave like a real aircraft, accepting commands and generating telemetry, without regard to the behavior of real flight hardware. Simulator are generally used to observe or study the behavior of some phenomena or abstraction. Simulators often use mathematical models of the phenomena. Network simulators do not trasfer actual data like flight simulators do not carry passengers from one place to another. Simulation cannot be used to implement virtuality.

For our purposes we define an emulator as a device built to work like another. A software emulator is a software artifact whose interface/API is identical to that of some computer program or physical device. From this point of view the definition of emulator seems to be very similar to virtuality. Emulators in fact can be used to implement virtuality, provided they are effective, i.e. they can really be used instead of the emulated entities. For example, emulators running too slow can be used for debugging or for evaluation but are not examples of virtualization.

On the other hand, there can be virtual entities not based on emulation. For instance a program providing the virtual appearance of several running copies of itself is not a case of emulation. In fact the virtual copies of the program behave in the same way and can be used one in place of the other, but there is not a separate software tool that provides the same interface.

Sometimes it is not clear if a virtual entity is based on emulation or not. For example sometimes the interface implemented by a virtual entity is just an abstract specification. It is the case of the Java Virtual Machine: a JVM is not an emulator just because the physical Java Machine does not exist.

Emulation or virtualization can be even limited to a subset of the interface. The virtualization extensions provided in modern processors dispatch software and hardware interrupts to different kernels, implement shadow page tables etc. In this way one processor provide virtual instances of itself to the virtual machines running on it. The main part of the binary interface, e.g. the instruction set, is not virtual but a small subset of specific instructions permits the efficient implementation of virtual machines. Virtual machine monitors are "software artifacts" providing the same interface/API of real machines, by giving the right response to software and hardware interrupts, page faults etc, when forwarded by the processor. Should these virtual machines be categorized as emulators?

Actually trying to answer to this question is not relevant to us. The point is that Virtual machines implemented in that way match our definition of virtuality, thus it is a proper definition.

1.3 Brief history of virtuality

The first Virtual Machine has been produced by IBM in the 60's, with the technology called VM/370 [2, 6], that made possible the use of big and expensive machines for many users. This system was composed of several components: each user had a single-user operating system called CMS (Conversational Monitor System), that didn't run on the hardware but on a virtualization layer called Control Program, that communicated with the real machine. The goal of this technology was to provide the same interface of the real hardware to all the users in a safe way: a single user couldn't bypass his dedicated space, causing damage to others.

In this early technology, virtualization meant a mechanism to split the real machine into many copies, ensuring software compatibility, isolation and performance comparable to the original. Years later, the introduction of multiuser operating systems and the wide proliferation of personal computers pushed this kind of virtual machine in the background.

Only in the 90's the idea of virtualization rose up again [20], thanks to the Java technology, that accomplished the demand of a mechanism able to make a computer program portable to different architectures. The solution was (and still is) to compile the source code for an emulated machine (Java Virtual Machine): then the Virtual Machine translates the requests for the host architecture. This way, a compiled Java program works in as many architectures as the ones the JVM has been ported to.

In recent years, a renewed interest for hardware virtualization got a foothold [26]: at the same time, powerful computing resources started to be available at low cost for both the server and desktop markets. This increased the spread of virtual machines for several applications: retro-compatibility of software for different and obsolete architectures, development and testing of new programs, prototyping of complex architectures, isolation of users and data.

As we will see, the panorama of virtualization tools is quite heterogeneous: it is possible to virtualize the hardware as well as the operating system, or to make partial virtualizations for single processes.

1.4 Classification

In order to understand better which virtual machines exist and how the Virtual Square Framework is related to them, some kind of virtual machine taxonomy is needed. Two classifications will be considered for the purpose of this work: the first one is drawn on an article from James E. Smith and Ravi Nair [7], the second one from the Virtual Square project [27]. These two visions are different but complementary, and both are useful for the aims of this work.

The first classification outlines different kinds of virtualizations from the point of view of the user, whereas the second one takes into account the technical features of virtual machines, regardless of their use cases.

Smith & Nair taxonomy

We will first introduce the concept of *machine*, and therefore *virtual machine*, from the points of view of the process and the system.

The *process* considers the *machine as the logic space of memory* associated to it, together with the set of instructions and registers of the CPU and the interface between the operating system and the hardware (ABI - Application Binary Interface[1]). The I/O is filtered by the system calls made available from the underlying software layers.

The *operating system* and the applications on the whole consider *the hardware as the machine*: the interface to the lower lever is the instruction set of the processor (ISA - Instruction Set Architecture [2]).

This difference matches spontaneously with a distinction between two kinds of virtual machines, *depending on which machine* definition is used.

As shown in Fig. 1.2, in the first case (*process virtual machine*), there is only the environment suitable for the execution of a single program by the user; in the second case (*system virtual machine*) the whole environment needed for an operating system and all its applications is reconstructed.

[1]An ABI describes the low-level interface between an application program and the operating system, between an application and its libraries, or between component parts of the application. It allows compiled object code to function without changes on any system using a compatible ABI.

[2]An ISA is a specitication of the binary opcodes that identify the native commands a particular CPU can execute. An ISA describes the aspects of an architecture that are visible to the programmers: data types, registers, instruction, memory, interrupts, I/O.

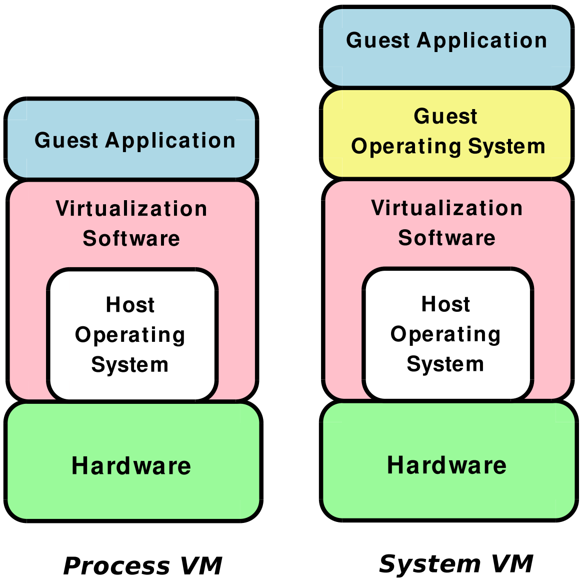

Figure 1.2: Process Virtual Machine vs. System Virtual Machine

Smith and Nair consider then another partitioning of the definition: virtualization (either process' or system's) can modify the behaviour of the layer it lies on, or use directly the real machine functions.

In the first case, the process or system *inside* the virtualization will see a machine with a structure coherent with the real one: in this case, the objective is to provide additional properties such as isolation or security. In the second case, the applications run are often coded for other architectures: therefore, the objective is to make the execution on different hardware possible (for compatibility purposes), or to create a common framework for many architectures (e.g. Java or .NET).

Virtual Square taxonomy

The Virtual Square projects aim to promote and collect virtualization technologies for computer systems, operating systems and networks. The objective is to allow a dynamic mapping of the network and its resources in a flexible way, in order to enable the mobility of applications, resources and users [9].

Following this broaden view on virtuality, we try to provide more general categories which apply not only to virtual machines.

1. *Host system intrusiveness.* The virtual monitor, i.e. the software which implements the virtualization, can be run on the host system at different layers, with different layers of privileges. It is possible to classify this case with the following:

   User level access. The virtualization can be run by any user program, without the possibility to operate directly on the real hardware or use any privileged access to it. This way a high degree of environment isolation is obtained, but, on the other side, the user suffer strict limits on the possible operations. In this case the overall realiability of the host system is not affected by the virtualization, but there may be performance issues.

Superuser access. The monitor needs superuser privileges or resources. It is a good compromise between intrusiveness and performance; security problems due to some monitor's code imperfections are possible, and it is inadvisable to allow non-trusted users to run this kind of virtualization.

Kernel patch or module. In this case the monitor needs a modification of the operating system kenrel in order to run. Kernel patches or modules usually add dramatic improvements to the system efficiency, in spite of a higher intrusiveness, not to mention the need of administration privileges on the host machine. The security and reliabilityof the whole system can be affected by errors generated by the code running in the kernel.

Native mode. The virtualization monitor *is* the operating system kernel of the system or the driver for the virtualized entity.

2. *Consistency to the lower layer interface*. The virtualization monitor can provide the same interface of the environment where it runs. In this case, i.e. *Homogeneous virtualization*, the monitor can provide services like access control or create virtual multiple instances of the virtualized entities. A *Homogeneous virtualization* monitor supports the same set of programs supported by the environment where the monitor itself runs. The opposite case is named *Heterogeneous virtualization*. *Homogeneous virtualization* can be *partial*: the virtualization can be limited to a partial subset of functions, leaving some resources intact. *Homogeneous virtualization* monitors can be modular: different virtualizations can be combined together as the interface does not change.

3. *Paravirtualization*. This is an optimization of the interface provided by a virtual monitor. The prefix *para* means similar. The point is that the implementation of a fully consistent virtualization sometimes is not efficient. Features required for the full compatibility sometimes are meaningless for the virtualization. It is the case of the management/minimization of disk seeks for virtual disks. Paravirtualization provide a different interface optimized for virtualization thus requires custom client software.

1.5 Emulators/Heterogeneous virtual machines

QEMU. The QEMU[3] Virtual Machine can emulate different architectures, such as Intel x86/x86-64, PowerPC and Motorola M68K. Consequently, it supports a wide variety of operating systems and virtual environments, and allows the execution of multiple OSes and architectures at the same time.

[3] `http://www.qemu.org`

Following our taxonomy, QEMU has the lower level of intrusiveness, and user-privileged execution is possible, and provide heterogeneous virtualization.

Following the Smith & Nair taxonomy, QEMU is a virtual machine that provides a complete environment, and can make conversions between different ISAs, so it can run code from other architectures.

QEMU dynamically translates the instruction of the virtualized processes [3]: the first time QEMU encounters an instruction, converts it in the matching host ISA, and stores it. The next time it encounters the same instruction, a further conversion won't be needed (avoiding waste of resources), because it uses the stored instruction.

Aside the complete virtualization, QEMU supports an emulation tool that allows the execution under Linux (only) of programs compiled for another architecture.

QEMU is an open source project, therefore the support is good and the bugfixes are quick; furthermore, it supports several architectures, becoming one of the most flexible and popular virtualization tools.

QEMU offered also a Linux kernel module named kqemu that exploits the hardware resources, when possible, thus enhancing the VM performance. This kernel module is no longer maintained.

PearPC. This is a program that emulates the PowerPC[4] architecture, and should be "architecture-independent". Currently, it works on little-endian machines with POSIX-11 compliant operating systems (Linux and others) or Windows. It provides a complete hardware emulation, but it has a high degree of performance degradation (between 1/15 and 1/500 of the native performance).

PearPC can be used by non-privileged user, but it's not possible to exploit the real hardware features, because they are use different ISA.

Mac-on-Linux. This VM[5] allows the use of two operating systems (Linux/ppc and MacOS) at the same time, on the same architecture. It's a case of system virtualization, but there is no translation of the ISA, being the host system a PPC anyway.

Mac-on-Linux requires a kernel module, therefore this degree of intrusiveness requires administration rights, but the performance is very good.

GXemul. This VM[6] (formerly known as mips64emul) is a computer architecture emulator being developed by Anders Gavare, that can run various unmodified guest operating systems as if they were running on real hardware. Currently emulated processor architectures include ARM, MIPS, PowerPC, and SuperH. Guest operating systems that have been verified to work inside the emulator are NetBSD, OpenBSD, Linux, HelenOS, Ultrix, and Sprite.

[4] `http://pearpc.sourceforge.net`
[5] `http://www.maconlinux.org`
[6] `http://gavare.se/gxemul/`

GXemul's processor emulation uses dynamic translation into an intermediate representation (IR). The translation step which would translate this IR into native code on the host has not been implemented. That step is not necessary, because the IR is already in a format which can be executed. In other words, it should be possible to port the emulator to new host architectures only with a mere recompilation; there is no need to implement a native code generation backend for each host architecture to get it running.

The source code of GXemul has been released as Free Software.

1.6 Homogeneous virtual machines

Kvm. KVM is the acronym for Kernel-based Virtual Machine. It implements hardware-assisted virtualization of the x86 architecure, as enabled by Intel VT-x or AMD-V features available on modern processors.

KVM support is included in the mainstream Linux kernel. KVM itself does not provide any emulation, a program at user level provides the emulation for virtual devices (it uses the device `/dev/kvm` to communicate with the kvm kernel support). KVM user level monitor is based on Qemu.

KVM may be configured to provide/support paravirtualization for networking and block devices.

KVM is free software.

VirtualBox. VirtualBox was initially developed by the German company Innotek. It is currently maintained by Oracle as Innotek was purchased by Sun Microsystems (and Sun became part of Oracle).

VirtualBox provides the virtualization of the x86 architecture on the same environment. It runs under several Operating Systems: GNU-Linux, Windows, MacOS, Solaris. It uses hardware-assisted virtualization. VirtualBox does not use paravirtualization: this is introduced as a feature to provide full compatibility with existing operating systems.

VirtualBox is available as free software, there is a proprietary extensions pack including the remote desktop and USB forwarding features.

Xen. It is an IBM supported project[7], and is a very efficient and Virtual Machine. It uses paravirtualization. Xen is based on a microkernel structure: it has the least essential features in order to manage guest virtual machines. Particularly, Xen doesn't support directly the whole hardware of the real machine, but relies on the device drivers the first activated virtual machine (domain 0).

With regard to the Smith & Nair taxonomy, Xen is a complete virtual machine, while with regard to the Virtual Square taxonomy, it is a native mode virtualization, i.e. it has the highest intrusiveness level (requires a kernel patch, and the recompilation of all the guest operating systems used).

---

[7] `http://xensource.org`

Denali. It is defined as an *isolation kernel*, that is, a kernel that isolates the virtual environments in it. Denali's architecture[8] is made of three main components: an *instruction set*, a *memory unit*, and an *I/O architecture*.

The virtual instruction set has been designed to provide performance and simplicity: it is a subset of the x86 ISA, therefore, in the majority of cases it's possible to run the instructions directly on the processor, gaining in terms of performance. In order to avoid some security issues, certain instructions are catched and managed by the Denali's monitor that rewrites binaries and uses some memory protection techniques.

The memory unit takes care of the isolation of each virtual machine's memory, and the I/O system has an interface similar to the x86, but simplified. The interrupt management is different in order to enhance support and performance of virtual machines: the interrupts are delayed and enqueued, and delivered to a VM only when it's actually running.

Denali is able to manage a very high number of virtual machines: having enough CPU resources, the VMs can be thousands.

The critical point of this approach is the forced recompilation of every application in the virtual machine. This implies an intrusiveness even higher of Xen, therefore the market refused this technology. Nonetheless, this project is very important, being the first one that implemented a paravirtualization technique.

VMware. This is a proprietary project[9] that implements a rather complicated architecture; it allows the virtualization of complete machines on x86 architectures, keeping the same interface between host and guest systems, either Linux or Windows systems.

There are several versions of the VM, but the *GSX server* and *ESX server* are particularly interesting. They differ in the way they realize the virtual machine's abstraction: in the first case, a host operating system that runs *GSX server* as an application is necessary, while in the second case the VM works directly with the hardware, and doesn't need an operating system.

VMware is distributed together with some tools for virtual systems administration, that make it suitable for commercial use or for non-expert users. On the other side, being a commercial, closed-source product, it can't be studied in depth.

VMware allows direct access to hardware resources of the real machine, and tries to make a compromise between a "classic" virtualization and a paravirtualization of some devices, making the overall VM rather efficient.

Other VMware features are: the possibility to customize the VM hardware, deciding the number of CPUs or some kind of peripherals, and the unification of more host machines seen as a unique machine, thanks to a virtual infrastructure layer. Finally, the mobility of VMs is possible without stopping the service.

[8] `http://denali.cs.washington.edu`
[9] http://www.vmware.com

Microsoft Virtual PC and Virtual Server. These are two versions of the Microsoft virtualization system: one is for personal use[10], the other for server use[11]. In both cases, the product supports hardware emulation (not ISA translation, host and guest both have a x86 architecture) and allows the creation of virtual machines that can communicate thanks to a virtual networking environment.

The difference between Virtual PC and Virtual Server concern scalability and the hardware emulation for server use. These are a proprietary closed-source products, that don't allow a further study of their structure.

1.7 Operating System-Level Virtualization

A kernel providing Operating System-Level Virtualization creates the abstraction of virtual machines by slicing its resources into partitions (named containers or zones or environments) which appear to users as independent computers.

Linux Containers (LXC). Recent Linux kernels include the support for process control groups (cgroups) and namespaces for resources.

Linux Containers use these feature to provide the abstraction of independent virtual execution environments. From the user point of view containers may appear as *full* virtual machine. Containers are in some sense similar to chroot: chroot limits the process *view* on the file system to a specific subtree, containers use namespaces to limit the *view* also in pid, networking, inter-process communication etc.

Namespaces and control group require superuser access, thus LXC is a feature for system administration.

Virtuozzo and OpenVZ. Virtuozzo[12] aims at making available several Linux "virtual private servers" on one or more hosts. Each VPS is a collection of applications that share the same virtual environment. Different virtual environments can differ in the vision of the file system, of the network, or for accessible memory areas.

Unlike UML, Virtuozzo doesn't provide multiple kernels execution: the only active copy is in the host system, and all the VPS use it.

Virtuozzo's core is a patch that modifies the operating system, adding the possibility to create different environments, where the standard Linux process tree is reconstructed (from *init* to the user processes). The performance is very good (the system calls have to cross only one kernel and not two, and there's no translation), but the flexibility is less than in UML (which, for example, can run kernels with different versions).

Virtuozzo can be classified as a *process VM*, with the same ISA. The intrusiveness in the host system is rather high, because a veritable "guest system" doesn't exist, but there are only some tools to create and manage the VPS.

[10] `http://www.microsoft.com/Windows/virtualpc`
[11] `http://www.microsoft.com/virtualserver`
[12] `http://www.swsoft.com/en/products/virtuozzo`

Virtuozzo is born as a proprietary, closed-source system, but nevertheless an open version has been recently released: OpenVZ, with some reduced functionalities. There is also a Windows version of Virtuozzo.

Linux Vserver. This is an open source system[13] used to create virtual servers. Its implementation is described as "a combination of security contexts, segmented routing, chroot, quota management extensions, and other standard tools". Basically, it associates a *context* to each process, and isolates the contexts one from the other.

The isolation takes place in 5 areas: file system (exactly as in chroot), processes (only the processes of the same context are visible among themselves), network (each server has a static IPv4 address), superuser capability (a virtual server administrator can't do everything it usually does on a real machine), interprocess communication (possible only among processes within the same context). This is realized by some kernel patches and external management tools.

No other layers are added in the 5 areas, so the performance is the same as the real system's, but the cons are less flexibility, and the unability of a virtual server administrator to have full privileges (e.g. it can't modify the network configuration).

We talk about Operating System Virtualization in the case of Virtual Machines that apply at process level, showing the inferface of an Operating System. These systems are less portable than complete systems, because they depend strictly from the interface they need to make available, and therefore from the virtualized operating system.

At this level, the term "virtualization" has more meanings than at the hardware level: in fact, some of the following examples clearly differ one another, but they all belong to the same category.

Other implementations. The same concept of Linux Containers was implemented in several operating systems under different names like Solaris Zones and IBM AIX System Workload Partitions (WPARs).

1.8 Process level virtual machine.

A virtualization is at process level when the virtual machine monitor is directly interfaced to the processes. This means that the service requests are directly managed by the monitor which implements a high level interface. (In the system level virtualization the monitor bevahes as bare hardware so that it is possible to boot an operating system kernel on it).

User-Mode Linux. This project[14] shares the same basic principles and techniques of Virtual Square's UMView (that will be described later in 1.9). User-Mode Linux (UML) monitor is a Linux Kernel, in fact the source code for UML is included in the standard distribution of the kernel: the command `make ARCH=um` generates of a UML monitor/kernel. Each process running in a UML machine is also a process (or more precisely a

[13] `http://www.openvz.org`
[14] `http://user-mode-linux.sourceforge.net`

thread) of the host machine, all the requests to the system (system calls) are processed by the monitor instead of the hosting kernel.

The objective of UML is to make possible the execution of a Linux kernel in user mode [12, 11]. This system is often used for kernel debugging, process isolation, usage of different environments without real machines, or for sandbox experiments (isolated environments where the user can test potentially dangerous operations without affecting the real system). The difference between UML and UMview is that while UML boots-up an entire Linux kernel, UMview is a modular partial virtual machine.

UML achieves this modifying the kernel, in order to make possible its execution as a regular user process; the UML interface is made by the system calls, that are redirected to the UML kernel (and not at the hosting kernel).

This procedure is possible thanks to the `ptrace()` system call that makes UML able to check its internal processes by system call tracing. A process that executes a system call is intercepted by UML, which can modify its behaviour. The *MView project, later documented in this book, applies exactly the same principle.

The elements virtualized by UML, and the techniques used, are:

System call. They are catched by `ptrace()` so that the calls towards the operating system are nullified (the process actually executes a `getpid()`); then UML runs its own system call management routine, finally the process' state is modified in response to the *virtual* system call.

Signals and traps. These are managed with the same technique used for the system calls.

Device and timer. The devices outside the virtual machine are implemented using `SIGIO`: UML receives a `SIGIO` when there is an input from a device, then determines which real descriptor is waiting, and associates it with the corresponding virtual IRQ, calling the appropriate routine. A similar technique is used with `SIGALRM` for the clock interrupts.

Memory fault. When a UML process accesses a non-valid memory area, the host system makes a `SIGSEGV`. UML catches it and checks if the error is really due to an illegal access (in this case the signal is forwarded to the process), or if it's about a page not yet mapped (in this case the mapping is taken care by the standard code for page faults).

Virtual memory. The physical memory is simulated by a temporary file, big as the RAM assigned to the virtual machine, in which the virtual memory pages of the different processes are mapped. UML uses the Linux kernel code for this, making it very similar to a real system.

Host file system. UML allows access to the real file system through a virtual file system called *hostfs*, that translates the VFS calls into calls to the real system.

Following the Smith & Nair taxonomy, UML belongs to the *process VM* category, with the same ISA: the translation doesn't change the type of executable files. According the Virtual Square taxonomy, UML runs at user level as a standard user process. It provides the same interface of the host operating system: as such it is a case of homogeneous virtualization.

Wine Wine allows to run Windows software on other operating systems (GNU-Linux, Solaris, MacOSX). "Wine Is Not an Emulator." this is the *motto* of this project. The authors mean that Wine does not emulate a processor but the compatibility is provided by *virtualizing* the Windows API.

Other Virtual Machines Windows Environmental Subsystems virtualize the API of other operating systems (e.g. posix or os/2) or obsolete versions of the same OS (e.g. Win16).

Another process virtualization support is Virtualsquare's purelibc. Purelibc provides the way for a process to track its own system requests. We name this idea "self-virtualization". A process can change in this way its interactions with the system. It is possible to add/change the features of a library by virtualizing its requests without the need to change the library code. For example by libvirt a library can believe to open a specific file while it is opening another file or using a chunk of memory instead.

1.9 Process level partial virtualization

Aside the VM just described, there is a category of applications that aren't definable as "virtual machines", but nonetheless they introduce some kind of virtualization. These system are not virtualization environments and don't place themselves completely between application and system, so they don't have the *isolation* property.

The aim of these tools is often to introduce new elements, generally not achievable in a real system, for testing, debugging, mobility or other purposes. We will concentrate on tools aimed at file system management and virtual network management, that are topics related to Virtual Square's ViewOS and *MView.

Virtual Networks

Other well known kinds *virtuality* have been developed in the field of networking. A virtual network is a network which appears to operating systems and applications different from how it really is.

Virtual Private Networks. A VPN is a protected network used to provide confidential communication on public networks. A remote computer can perceive other computers connected through the VPN as they were on a Local Area Network (LAN). When a computer is connected to a VPN, all the network traffic is routed through the VPN itself, in order to avoid accidental disclosures of confidential data and external attacks.

VPN are generally implemented by encrypted tunnels. This kind of network is commonly used by traveling employees to access the company internal network.

Overlay Networks. An Overlay Network uses the same protocols of real networks to communicate on virtual interfaces, using virtual addresses and virtual connections. The main goal of an ON is the independence from changes in the real underlying topology.

Several Peer-to-Peer (P2P) applications provide some kind of ON for a quick reconfiguration and for hiding the real data exchange. The replicated service Atamai [1] uses ON methods to increase the level of reliability for streaming services.

Networks for Virtual Machines. Another kind of virtual networks are those provided by VM monitors in order to give virtual networking services to their client VM. All the virtual network interfaces should be virtually interconnected to other VM and to the real networks.

Several VM monitors provide their own networking support: XEN provides a virtual network between the different domains, Domain 0 can run bridging or routing software to make a gateway to the real networks, Qemu and GXemul provide virtual networks which emulate a Masqueraded (NAT) network.

The VM monitor creates all the network connections for the hosted virtual machines. This method does not require any configuration, but it's not possible to forward an incoming connection from a real network to a Virtual Machine.

MacOnLinux, and several other VM monitors running in dual mode, add kernel code to provide their own view of the network like virtual bridges or interface sharing.

Virtual Networks are used for many purposes, from new network models testing, to local virtual networks used in production environments, to isolation of different network services running on the same machine.

Virtual Distributed Ethernet (VDE) is the VirtualSquare contribution for virtual networking. It is often referred as the "swiss knife of virtual networking" as it can used for many applications and in substitution of many existing tools.

VDE. This is a tool[15] to create and use virtual networks [8]; it allows to link computers placed in distant locations on the Internet, designing arbitrary topologies, creating virtual LANs to connect them.

The interface of VDE towards the real network is realised using the TUN/TAP support of the Linux or MacOS X kernels; it supports the virtual machines QEMU, KVM, UML, VirtualBox, Bochs and MPS.

Basically, the idea is to connect different machines independently from the restriction imposed to the users. The virtual network topology can be managed in every way the real system administrator is allowed to. VDE creates an interface identical to a real network interface, either in the real side and in the virtual side.

The tunnel used to transport data-link-layer data can rely on any streaming protocol (for example, a ssh connection is enough to create a ciphered VPN with VDE).

[15] `http://vde.sourceforge.net`

VDE tools are intuitive as they provide virtualization of real physical network entities: switches, hubs, cables.

File system

Like networks, virtual file systems are aimed at particular needs: for example to round on operating systems limits, or to create better abstractions for data management. Sometimes, labeling this services as "virtual" may be inaccurate, for they often realize a unique vision for the whole system, without creating neither a virtual machine nor an isolated environment.

However, file systems are one of the most interesting aspects of virtualization techniques, since the subject is very closely related to Virtual Square's ViewOS. Several types of file systems have been analyzed, even if not strictly "virtual", trying to understand their architecture and implementation, in order to achieve code reusability. View-OS uses the central role played by file systems in UNIX environment to extend the idea of file system virtualization to other virtualization like support of virtual devices or virtual networking stacks.

chroot. It realizes something very similar to a virtualization: it shows a file system different from the real one to a certain process tree, but in reality it's a mere subtree of the original file system.

The implementation is achieved through a single system call: the kernel itself offers this feature. In order to run *chroot()*, administrator privileges are required, and the kernel just changes the root of the filesystem to the one specified, only for the process that runs the function and for its children (the so-called *chroot cage*).

Chroot is often used to isolate some programs, but it has some notable limits:

- It shows only the origin of the file system: this implies the use of other tools for other purposes, like network virtualization, and it isn't a very flexible tool.
- There are several issues in relation to security: it's not possible to give root privileges to a user within a "cage" without the danger of the user easily "going out" of it and access the rest of the file system.

schroot. `schroot` makes chroot available to unprivileged users under predefined restrictions defined in a configuration file.

It is a root set-userid executable. This tool prove that chroot is useful for users. On the other hand this tool can be very dangerous as a misconfiguration can create root escalation vulnerabilities.

fakeroot/fakeroot ng. Fakeroot is an application which gives to a normal user the perception to access files as the administrator of the operating system. Using fakeroot it is possible to create and manage archives containing files owned by root. It is just a virtual, emulated view of ownership rights and permissions of files and directories.

As the project's home page defines [16]: "Fakeroot-ng is a clean re-implementation of fakeroot. The core idea is to run a program, but wrap all system calls that program performs so that it thinks it is running as root, while it is, in practice, running as an unprivileged user"

FUSE. This project[17] aims to allow users to load drivers for file systems, either real or virtual. This is not a "proper" virtual machine, because it doesn't distinguish between virtualized and non-virtualized processes, but applies to the whole system the effect if its execution: any file system mounted through FUSE is seen from all the processes in the same way. Fuse is composed by three levels:

- a kernel module that interfaces to VFS, the abstraction layer of the Linux filesystem, and creates a special device `/dev/fuse` used to interact with the module. The last versions of the linux kernel integrate this component;
- a library that allows the communication between the `/dev/fuse` device and the file system modules, modifying the requests of the device into a set of system calls (`open(), read(), symlink()`);
- a file system implementation, made using the interface exported from the library. This module is an executable file, invoked by the `mount` command. Once the program is run, it registers in the kernel through the library, and waits on a file descriptor assigned through `/dev/fuse`. This channel will be used by the kernel to run the code for the system calls.

The project offers several interesting topics:

- FUSE provides a simple interface, so it's very easy to develop new modules;
- the modules are rather similar to the ones used in Virtual Square's *MView, therefore the compatibility of the two projects can be easily achieved;
- there are many file systems used and tested in FUSE: this, united with the previous point, leads to a good reuse of the code.

Unionfs. This project[18] takes into account the different needs of users and administrators of a system: sometimes, an administrator wants data, software, and configuration kept in separate places, or wants to maintain interrelated data in different disks or partitions, while the user will find more intuitive to find everything in the same position.

The solution proposed by Unionfs is the *unification*: more sources (branches) of data are fused and shown as one to the processes.

Unionfs implements this by virtualization, with a layer that keeps the Unix semantics and merges the visions of different filesystems, that can be either on physical partitions, on logical images, or on removable disks.

[16] http://fakeroot-ng.lingnu.com/index.php/Home_Page
[17] `http://fuse.sourceforge.net`
[18] `http://www.filesystems.org/project-unionfs.html`

Unionfs uses a series of layers that correspond to the data sources, each of which is associated to a priority: the vision is given merging these layers, and acting on the top priority file system in case of conflicts. The applications of this tool are:

- a user sees his data in a unique place, while the administrator keeps them in different locations;
- adding and using software from an image or a CD-ROM is possible merely by merging the filesystem with the system's one;
- it is possible to make a snapshot of the system, creating a read-only archive with a backup, meanwhile, new data is saved in a temporary archive that will make a new system backup, and so on;
- sandboxing: sometimes, a user needs to recover the system to its original state, for example in case of sporadic usage by non-trusted users. This can be realized with two levels: the first with read-only permissions, and the second will be deleted every time the user wants to clean the sandbox.

Unionfs is implemented by a kernel module, that once loaded and used, creates a new layer, shown as a new type of filesystem by the kernel, and as VSF by the different file system managers.

PlasticFS. Unlike Unionfs, this one[19] is a dynamic library loaded by `LD_PRELOAD`, and not a kernel module. The objective is to modify the vision of some portion of the file system from the point of view of a process. Some of the possible transformations allow to achieve the behavior of *chroot()*, to change the capital letters of a file name, to unite parts of a filesystem like Unionfs. The PlasticFS solution has two problems: from a technical point of view, using the `LD_PRELOAD` variable forces the recompilation of the system library, or its rewriting; from a security point of view, any software can overcome the library calling directly the kernel services, therefore PlasticFS can't be used in security-critical contexts.

ViewOS: umview/kmview.

ViewOS is a general purpose, modular solution for process level virtualization available for unprivileged users.

umview is the implementation of View-OS based on standard system call tracing facilities already available in the Linux kernel, kmview requires a specific module in the kernel[20].

ViewOS modules implement specific virtualizations: file system, device, networking, etc.

Users by ViewOS can have the same services provided by chroot/schroot, fakeroot(-ng), fuse, lxc. In some sense ViewOS can be seen as a virtual implementation of namespaces for users. ViewOS is based on the implementation of a partial modular virtual machine. Each system call generated by the virtualized processes get processed by the kernel or by the virtual machine monitor

[19] `http://plasticfs.sourceforge.net`

[20] and a kernel supporting the utrace extension (see: http://people.redhat.com/roland/utrace/).

depending upon specific conditions. For example it possible for users to (virtually) mount file systems: all the requests for files in the mounted file systems get processed by the file system module of ViewOS while for all the other files the requests are processed by the real kernel.

ViewOS provide a unified approach to a number of virtualizations: it is easier for users than approaching different syntaxes, and it reduces the incompatibilities. The approach is also safer than using the other tools: ViewOS does not require root access, so bugs and misconfiguration cannot damage other processes or create security threats.

1.10 Microkernel systems

These systems don't belong strictly to the virtual machines category, but they have some features that are interesting for the Virtual Square's ViewOS project, and more generally, for the virtualization concept itself.

A microkernel system, in fact, tries to maintain the less components possible at a privileged status, leaving all the system policies management (scheduling, file system, security, network and so on) to user-level servers.

The features taken into consideration for this work are:

modularity: on microkernels, adding services and features to the system it's almost always possible by adding modules. This achieves some of the effects of several virtual machines: usage of functionalities normally not in the system, or association of different managers for different kinds of requests;

user level usage: the services are run with reduced privileges, ensuring good levels of security and stability. This way, testing and debug are easily achievable without need of virtual machines like UML;

dynamicity: since the services are provided by non-kernel servers, there aren't limit to their complexity: the system is then really flexible and is able to provide different interfaces based on processes, users, or other criteria. This feature is essential for ViewOS.

GNU/Hurd[21] is a microkernel POSIX compliant system developed by GNU. Following the Unix view, Hurd provides an interface to everything through the filesystem: every device (even network interfaces) is manageable by file operations, and the kernel only takes care of the message passing between the application and the processes. Hurd provides also a series of libraries for a faster development of servers for managing single files, subtrees of the filesystem, or network interfaces.

Moreover, Hurd has the possibility to assign a service to *every* file, making the whole filesystem potentially "virtual".

[21] `http://www.gnu.org/software/hurd`

CHAPTER 2

V^2: The Virtual Square Framework

2.1 Introduction to Virtual Square

Virtual systems offer interesting perspectives both in research and in applications. In the previous chapter we have seen several research projects and a great number of commercial products that exploits the "virtuality" concept.

The majority of those tools generally provide isolated environments of virtual machines and networks. It is possible to interoperate between similar virtual services and to connect virtual machines and networks with real systems, but it is not generally possible to interoperate directly between hetherogenous systems (within the virtual environment).

Every component of a computer system (memory, processors, networks, devices, filesystems) can be virtualized and they can interoperate with each other. The goal of the Virtual Square Framework (V^2) [10] is to build and use entirely virtual systems together with mixed environments comprised of both real and virtual entities. In other words, the objective is to create a "Square" where existing tools can communicate and interoperate.

Started as a research project, now V^2 is a framework that contains several tools, with the aim of investigating what is "virtuality" today, trying to give a unified approach at different forms of virtuality. The research focuses on looking for common principles between virtual services and tools, *and to make the granularity of virtualization as fine as possible*. It can be compared to a game based on building bricks (say "Lego"): only incompatible bricks are available, and the goal of the game is to create new interconnection bricks. From this perspective V^2 can be seen as a complex middleware able to create interfaces among virtual and real systems and networks.

The word "Square" in this context can be explained in two ways:

- V^2 has the meaning of Virtual "Squared", because the objective was to create new virtual machines and networks in the form of *nested* virtual services: virtual components based on other virtual components.

- The authors also conceive the project as a central "square": a place where different entities can be connected and interact, where new ideas are exploited, shared and communicated, as it happpens in our cities' public squares.

2.2 V^2 goals and guidelines

Virtual Square goals can be summarized in the following three items:

Communication. A unified virtual communication infrastructure is needed in order to achieve interoperability between different kind of virtual machines.

Unification. Different kinds of virtuality, now realised as different services and tools and based on different models, have to be unified and considered as specific cases of a broader definition of "virtual machine".

Extension. A uniform view of virtuality will lead to the emerging of new application domains, that will extend virtual execution environments.

Moreover, the design of V^2 follows some basic principles:

Re-use of existing tools. Virtual Square tools enhance the features of existing programs and run on common and widely-used operating systems (mostly on GNU/Linux, but some tools have been ported to other platforms). This way, V^2 is not only an academic experiment, but it's really usable on real contexts.

Modularity. Each tool has been designed with a composable, modular structure. Users can design the virtual architecture needed for a specific task, that can be put in place running and connecting the required modules. V^2 aims to provide modules for every useful combination of tools.

No architectural constraints. Virtual Square tools are highly configurable and easy-fitting for different operational environments and application needs. It's up to the user to decide which tool is best suitable for his/her hardware or software architecture, via a wide range of compatibility. Architectural constraints imposed by modern operating systems can be viewed just as a lack of design, and not as a mere pursuit of security enforcement. Virtual Square aims to overcome this restrictions in an effective way. The TIMTOWTDI (there is more than one way to do it) philosophy defined by the Perl programming language designers should apply to operating systems, networking, etc. particularly when dealing with virtuality.

As for architectural contraints, the lack of design of the modern operating systems prevents the use of several functionalities. For example a user cannot mount his/her own disk image in an easy way without acquiring administration privileges. Actually, this operation is possible through a virtual machine: the user can run a User-Mode Linux and mount the image on that virtual computer because *within it* he has administration privileges. Anyway, this is a very complicated and resource consuming task, for an entire linux kernel have to be loaded as a process.

The disk image is a personal file of the user, so the mount operation is nothing more than a “sophisticated” access to this file. Current operating systems forbid this access because the POSIX model[1] supports only *global* mount made by *system administrators*: this way the file system structure changes for all the users and processes.

Virtual Square aims to avoid this kind of architectural constrains, by giving effective abstractions to overcome this kind of problems without administration privileges.

Openness. Virtual Square is an open laboratory for Virtuality. All the code is available on public repositories under free software licenses, most of which under GPLv2 on Sourceforge.net and Savannah.nongnu.org. All the programs and tools, finished or under development, must be used as prototypes to test the Virtual Square concepts. Everyone can download, study, and use V^2 tools, testing the effectiveness of the proposed ideas. The authors underline their wish to fulfil the principles of the Scientific Method as defined by Galileo Galilei: all scientific publications should be subject to this method, including computer science publications [17].

2.3 V^2 components

The Virtual Square Framework currently contains a set of software tools and libraries, that extends over 100.000 lines of C code. The structure of the whole project is shown in Figure 2.1. The relation “A $\rightarrow$ B” means that “component A *depends on* component B”, so B must be installed, otherwise A won’t work.

Here is a brief description of the V^2 main components, which will be described further on the next chapters:

VDE. Virtual Distributed Ethernet is a Virtual Ethernet local area network. It is able to interconnect virtual machines (Qemu, μ/MPS, UserMode Linux), partial Virtual Machines (*MView), as well as virtual interfaces of real systems (TUN/TAP)[2]. All the connected machines see VDE interfaces as they were interconnected by a standard Ethernet LAN, although they can run on different real computers, maybe geographically distributed. It is like a multipoint VPN that interconnects virtual machines and real hosts.

LWIPv6. This is a complete TCP-IP (multi) stack implemented as a library, that works with both IPv4 and IPv6 protocols. Technically, LWIPV6 is

[1] Portable Operating System Interface is the collective name of a family of related standards specified by the IEEE to define the application programming interface (API) for software compatible with variants of the Unix operating system. Linux is mostly a POSIX compliant Operating System.

[2] TUN and TAP are virtual network kernel drivers. They implement network devices that are supported entirely in software, which is different from ordinary network devices that are backed up by hardware network adapters. TAP simulates an Ethernet device and it operates with Layer 2 packets such as Ethernet frames. TUN simulates a network layer device and it operates with Layer 3 packets such as IP packets. TAP is used to create a Network bridge, while TUN is used with routing.

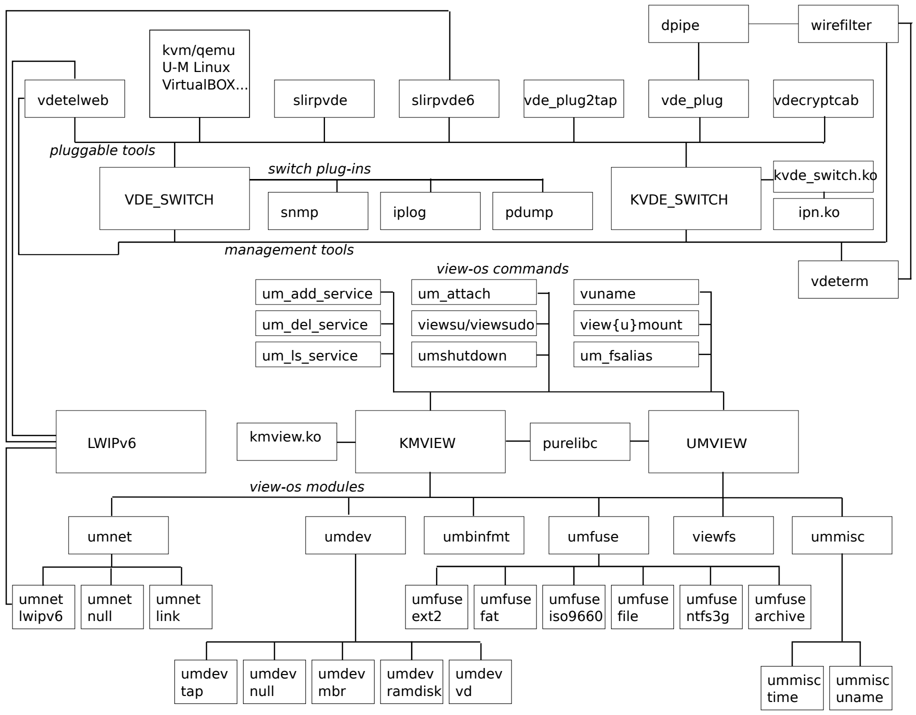

Figure 2.1: Virtual Square tools and libraries

not a dual stack but a hybrid stack: one single stack able to deal with both protocols. This library is used by VDE and *MView (and possibly by any application) to interface directly with virtual networks, without administration privileges. LWIPv6 implements the msocket interface to run several networking stacks at the same time.

VDETelWeb. This application is a Web server and a Telnet server that uses LWIPV6 in order to remotely configure the VDE switches.

IPN. This is a new communication service that exploits the networking methodologies in the context of Inter Process Networking. With IPN processes communicate as if they were in a networking environment. IPN is implemented as a kernel module and supports many inter process communication switching policies. The kvde_switch policy implements VDE switches in the kernel.

purelibc. This is an overlay library that converts the GLIBC, the GNU standard C library, into a pure library: a library that can upcall instead of running the system calls. With PURELIBC a process can trace (and virtualize) its own calls. It is used by *MView modules to support the recursion of virtual environments.

***MView.** User/Kernel Mode ViewOS is the implementation of one of the basic principles of V^2: ViewOS. ViewOS achieves the concept of freedom of processes: each process can have its own perception of the running environment. This way, the *granularity of virtualization* is extended to processes. View-OS flips over the default behavior of the operating systems, where all the processes access the resource in a uniform way: the same pathname means the same file, and the same IP address means the same host, reachable using one shared IP stack. ViewOS overcomes this model (the *global view assumption*) and offers an environment where each process has the freedom to define its own view of the system.

The leading star of *MView means UMView or KMView, that are two different and interchangeable implementations of ViewOS. They offer identical functionalities to the user, but in two different ways: UMView is implemented entirely in user space, and requires no modification to the kernel, nor administration privileges to be installed, while KMView depends on a kernel module (thus requiring administrator access at least to compile and load the module) and on the utrace tracing mechanism, providing faster performance than UMView.

*MView is a Partial Virtual Machine which implements the View-OS concepts in a GNU-Linux environment. It is just the skeleton, for it doesn't implement any virtualization by itself, but under this skeleton it's possible to dynamically load modules which can redefine the system calls of the processes running inside the virtual machine. These modules can specify under which conditions the virtual system call must be used instead of the existing real calls. This way it's possible to write modules able to virtualize entire sections of file systems, networks, devices, etc. Moreover, the services provided by *MView modules can be nested. Currently, the working modules written for *MView and available in the source repositories are:

- **umfuse** is the File System virtualization module. Through a set of submodules, one for each file system format, it allows the user to mount (via the `mount` system call and utility) file systems of that format. This is a mount operation whose effect are limited to the processes running within the partial virtual machine. The submodules are source-code-compatible with those of the FUSE project [28], which means that only the recompilation of the module is needed for it to work with both projects. Some modules have been created by Virtual Square developers such as *fuseext2* (for ext2/ext3 file systems), *umfuseiso9660* (for CD-ROM images), and *umfusefat* (for FAT32 filesystems, often used in usb keys and other external drives). Another file system module developed by the Virtual Square Team is FSFS: an encrypted file system more scalable than NFS, because it moves the computational cost of encryption to the clients. Other modules have been ported from FUSE, like the case of cramfs, encfs and sshfs.
- **umdev** is used to create virtual devices. The processes in *MView will access the virtual devices as they were real: *umdev* implements special files that correctly manage all the specific control calls (ioctl). Like umfuse, umdev has a modular structure, so it's possible to load submodules in order to manage different kinds of virtual devices. *umdevmbr* is one of these submodules, used to create a special file

associated to a disk image. Using umdevmbr, disk partitioning is possible, and the user can work on each partition. If the disk image is mounted on `/dev/hda`, umdevmbr will define `/dev/hda1, /dev/hda2,` and so on, with the same naming convention used by the kernel. It is the possible to run any command on `/dev/hdaxx` like mkfs, chkfs or mount with *umfuse*. *umdevtap* provides a virtual tun/tap device interface, and *umdevramdisk* is used to create virtual ramdisk devices.

- **umnet** is the *MView module that virtualizes the networking support. *umnet* provides the new msocket support for multiple stack management and provides several submodiles. *umnetlwipv6* uses LWIPv6 to provide the processes with virtual network interfaces, instead of the interfaces provided by the undelying kernel. Usually, this interfaces are connected to VDE networks but it is possible to connect them to virtual interfaces (TUN/TAP) of the host system, *umnetnative* is an interface to the existing networking stack, *umnetnull* is used to deny the access to the network, *umnetlink* permits combine the services provided by several stacks for disjoint sets of protocol families.
- **ummisc** is a generic module that allows the loading of several submodules, through the `mount` system call, in order to manage some extra features. Currently, two submodules are available: *ummiscuname*, used to modify the hostname and domainname for the virtualized processes (a high number of *MView instances can be hard to tell apart: this modules assigns a different name to each instance), and *ummisctime*, that allows to change the frequency and offset of the clock.
- **umbinfmt** associates interpreters to executables and scripts depending on some properties like extensions, magic numbers or pattern matching. Using this module it's possible to run binary executables compiled for machines code-incompatible with the processor of the host computer. *umbinfmt* can be configured with the same scripts used for the *binfmt* module part of the Linux kernel.
- **viewfs** is a *MView module that virtualizes the file system structure, enabling file system operations like rearranging, moving, hiding and protecting files and directories. It is also possible to add or modify existing files using copy-on-write methods, in order to keep the original files untouched in the underlying system.

μMPS are two (μMPS and MPS) "traditional" virtual machines, designed for educational use in teaching environments. These two related virtual machines implement a complete MIPS-based system: μMPS differs from MPS because it just simplifies the MIPS architecture in order to make it more suitable for undergraduate teaching. μMPS systems can run as fully compatible hosts in a VDE network.

CHAPTER 3

What's new in Virtual Square

Virtual Square is an open lab where the research team can investigate on the limits of current virtualization models and techniques. V^2 also proposes new ideas to solve all the major issues, in short the project gives a new perspective on virtualization as a whole.

This chapter aims to give the reader an overview of some results that V^2 has reached. Some have already been introduced in the previous chapter, and all will be described in details in the following parts of this book. The purpose of this chapter is just to summarize the advances achieved by the Virtual Square Project.

3.1 ★VDE: a swiss-knife for virtual networking

Ethernet is the most used family of computer networking technologies for Local Area Network. This means that all the major networking protocols and services run over an Ethernet network.

The virtualization of the Ethernet framing and packet delivering is off-the-shelf compatible with everything running over an Ethernet, i.e. almost everything. At VirtualSquare we are used to saying that VDE is compatible, of course, with ipv4 and ipv6, but it is already compatible with also with any IPv7,8,9,... as they will have to run on Ethernet networks, too.

VDE can interconnect virtual machines, virtual interfaces of real machines, programs having embedded networking stacks, etc, like a swiss knife has several tools: screwdrivers, openers, corkscrew...

VDE it is the VirtualSquare most *popular* project not only because is the oldest project but also for the combined support of a large set of interfaceable components and an almost universal compatibility with the networking protocols.

A swiss-knife is an object able to substitute several different tools. In the same way VDE can be configured to connect Virtual Machines but also to create a Virtual Private Network, or for NAT traversing. Several functionalities currently provided by separate tools are provided by VDE.

VDE can be used instead of well-known tools but creates also new opportunities. We have used VDE as a the data-link layer for our LVIPv6 stack, permitting the implementation of programs having their network stack embedded. It means that a program (e.g. a web server) can directly communicate on a virtual network, it is possible to migrate the program on another computer keeping the ip address and the state of the communication sessions. An example of application with embedded stack is the telnet and web server of a vde switch (vdetelweb).

We have created public VDE networks where users can connect their switches. These networks are not connected through routers to the Internet: the idea is similar to a public square where friends can meet and talk. Users can join a public network to exchange data as thay were on a local area network.

This insulation property can be used to develop shooting ranges for computer security. It is possible to test attacks to virtual machines from a private close network so that the effects of the experiments cannot propagate to production systems and networks.

Like any multi-purpose tool, the range of possible application are wide and the creativity of users ofter goes beyond the idea of creators. VDE has been applied to educational tools for teaching networking like Marionnet [21], or for the interconnection of cloud computing components [23]. It has been ported to many architectures and included in many free software distributions.

3.2 msockets: Multi stack support for Berkeley Sockets

The Berkeley socket API is the *de facto* standard for network programming.

Unfortunately this API has been designed to use one single stack for each protocol family. This is the common case for many (all?)operating systems today but the ability to access and use several protocol stacks permits new applications and simplify solutions to common problems.

Networking is also an exception for *nix systems for naming. In fact the file system has been used as a unified naming method for files, devices, fifos, unix sockets, ... and almost everything but networking.

Virtual Square has designed an extension to the Berkeley API having the following features:

- multiple stacks can be used at the same time
- the file system gets used for naming the stacks
- it is backward compatible with existing applications.

The new extended API has been named *msockets* after the most important call of the API itself *msocket*, namely acronym for multi-socket.

This extension has been implemented by Virtual Square, in the View-OS module named umnet, but msockets is a clean and general solution to support several networking stacks.

The Stack special file

Each stack can be accesses by its special file.

The stack type has been defined (see stat(2)) as

```
#define S_IFSTACK 0160000
```

Execute x permission to the special file permits the configuration of the stack while the read permission r permits just the data communication.

If a user has the x permission, she can define interfaces, addresses, routes, (like CAP_NET_ADMIN capability). w permission has not been defined yet but it could be used for high level services (like binding of sockets ¡ 1024 or multicasting capabilities). A user without r permission gets an error if she tries to open a socket (using socket or msocket).

The msocket call

The syntax of msocket is:

```
#include <sys/types.h>
#include <msocket.h>

int msocket(char *path, int domain, int type, int protocol);
```

domain, *type* and *protocol* have the same meaning as in `socket(2)` (with some extensions, see over). *path* refers to the stack special file we want to use, when *path* is NULL the default stack gets used.

In fact, each process has a default stack for each protocol family. It is possible to redefine the default stack by using the tag `SOCK_DEFAULT` in the *type* field. Thus:

```
msocket("/dev/net/ipstack2",PF_INET,SOCK_DEFAULT,0);
```

defines `dev/net/ipstack2` as the default socket for the calling process for all subsequent requests for sockets of the IPv4 protocol family.

```
msocket("/dev/net/ipstack2",PF_UNSPEC,SOCK_DEFAULT,0);
```

Using msocket to define the default stack for the PF_UNSPEC family means to define `dev/net/ipstack2` as the default stack for all the protocol families it is able to manage.

When programming with *msockets* the former (system) call:

```
socket(path, domain, protocol)
```

is equivalent to:

```
msocket(NULL, path, domain, protocol)
```

The msocketpair call

msocketpair is an extension of socketpair like msocket extends socket.

The syntax is:

```
int msocketpair(char *path, int d, int type, int protocol, int sv[2]);
```

The `mstack` command

mstack is a command that defines the default stack for the mstack process itself and execute (using `exec(2)`) the a command. As a result the command uses the stack defined by mstack as its default stack.

mstack is useful to run utilities and commands designed to work on one shell (e.g. using the standard Berkeley socket API) on one of the several stacks supported by msockets support.

The syntax of `mstack` is the following:

```
mstack [-u46npbihv] [-f num,num#] stack_mountpoint command
```

where *u,4,6,n,p,b,i* stands for PF_UNIX, PF_INET, PF_INET6, PF_NETLINK, PF_PACKET, PF_BLUETOOTH, PF_IRDA respectively, *h* is help, and *v* is verbose. Protocols can be set by listing the protocol numbers (see /usr/include/linux/socket.h). When there are no options for protocol families, the stack named in the command beames the default one for all the protocol families it supports.

example:

```
$ mstack /dev/net/ipstack2 ip addr
```

executes the command `ip stack` on the stack `dev/net/ipstack2`.

```
$ mstack /dev/net/ipstack2 bash
```

thus in the subshell

```
$ ip addr
```

shows the interface addresses of the `dev/net/ipstack2` stack. opens a subshell where `dev/net/ipstack2` is the default stack.

3.3 IPv6 hybrid stacks for IPv4 backward compatibility

There are two ways to approach the virtualization. The common way is to adapt your needs to fit the features of the available tools, methods and technology. The drawback of this way is that you can be forced to give up the idea as there is not the right tool.

At VirtualSquare we are users but we are also teachers, researchers, software designers and delevopers, then we approach the problem in the reverse way: we start from the need and then we adapt or we create the software when it does not exist yet.

This is the case of LWIPv6. The implementation of an entire TCP-IP stack suite as a library is not a virtualization project as such, but we needed it in many VirtualSquare Project. We weren't able to find a library able to met our requirements. In fact we needed an implementation of a TCP-IP stack as free software, enough complete and effective to replace the kernel stack at least for the most common user activities (ssh, web browsing etc.).

Maybe nobody implemented such a library because the standard socket API does not even include the way to deal with several networking stack available at the same time.

We showed in the previous paragraph `msocket`, a new system call that extends the socket API to the management of several stacks.

The support of IPv6 is a strict requirements for our library: the virtualization of networking stack means that IP addresses can be assigned not only to interfaces as it is common today, but also to all the processes which need their own management of networking. IPv4 addresses are scarce and are not suited for this kind of usage which highly increases the number of addresses to assign.

LWIPv6 is an hybrid stack: it means that it is able to manage both IPv4 and IPv6 packets. All the common parts of the stack have been shared, like the TCP and UDP implementations while there are specific managements of version dependant parts (like ICMP). The core network layer protocol IP has only the IPv6 routing engine. The IP layers handles the packets using a pseudo IPv6 header, which contains the header data of the packet for IPv6, and is created on the fly for IPv4 packets. All the IPv4 addresses gets converted into IPv4-mapped addresses of IPv6.

This choice simplifies the library code, reduces its memory impact and simplifies its usage. In fact, it shares most of the code between IPv4 and IPv6 and provides a unified API for both families. This approach create a limited processing overhead for IPv4 packets as the header must be converted.

The LWIPv6 project has always been evolving. One goal of Virtual Square is to have LWIPv6 as our unique networking stack, even if it is used in different roles in many Virtual Square projects.

In fact, LWIPv6 is the *embedded* stack of vdetelweb, the telnet and web interface for vde switches. LWIPv6 provide the multi networking stack implementation to umnet, the netwokring virtualization of View-OS. Recently the latest experimental version of LWIPv6 is used also for the new implementation of slirp named slirpv6, as now it supports also IPv6.

LWIPv6 code is a fork of LWIP project by Adam Dunkels [14]. LWIP and LWIPv6 have a different target application: LWIP is mainly designed for embedded system, LWIPv6 is a library. LWIPv6 added several features to LWIP, not only the management of IPv6. For example LWIPv6 supports AF_PACKET for raw packet management, AF_NETLINK for network configuration and it can configured to provide network filtering, NAT, dhcp client. LWIPv6 is *multistack*: a process can run several LWIPv6 stacks at the same time.

3.4 What a process views

In modern OS, system calls and the sole and unified way for a process to interact with the outer environment. The system call interface has been designed mainly for operating system security and reliability. In fact processes run inside a controlled environment: the operating system is protected from errors and malicious behavior of processes. In the same way processes are protected from errors and attacks by other processes. If the main rule of theater is ”the show must go on”, here we can say ”the O.S. must go on”, no matter the errors and accidents our actors, the processes, can do.

Each process thus runs using an “extended language” composed by the standard instruction set of the processor and the set of system calls provided by the kernel.

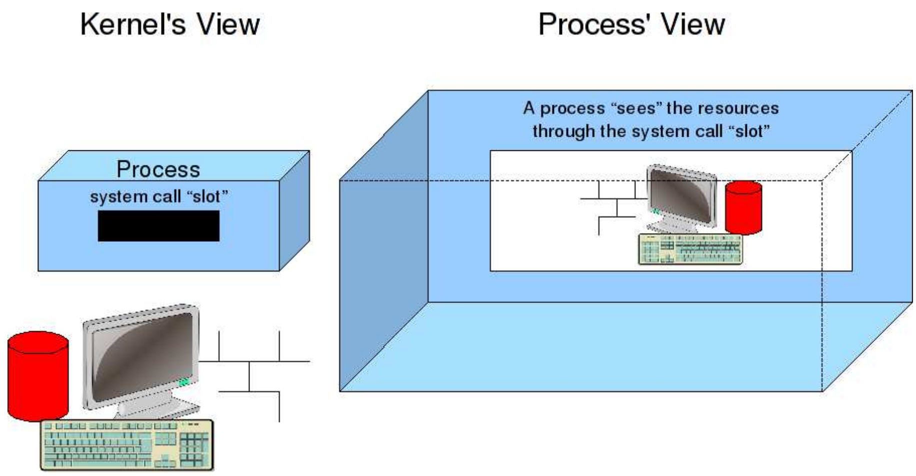

Figure 3.1: Kernel's and Process' views on the same world

Figure 3.1 shows on the left side the common perception of the confinement of the process from the kernel perspective. In the same figure on the right side there is the same situation from the process point of view. The system call interface is the only permitted slot the process can use to see the outer world.

This is the interface View-OS uses to provide a virtual perception of the environment to the processes. This approach has been already used by other projects, like User-Mode Linux, as a different support for Virtual Machines (complete VM, those requiring an O.S. to boot), View-OS extends this idea supporting a general, modular, user level support for partial virtualization.

The concept of system call virtualization can be implemented in several ways: at library level, as a tracer process, at kernel level. All these implementations are present in Virtual Square.

Most processes use dynamic libraries, more specifically the wrapper functions for system calls are part of the C library (glibc for Linux). THe glibc library does not currently provide any support for this kind of virtualization, but a different C library can include the support. Pure_libc is the V^2 implementation of a C library providing system call virtualization. Instead of a complete reimplementation of a C library, pure_libc is a library based on glibc, overriding the definition of what is needed for the new feature. In this way pure_libc is highly compatible with existing application using glibc. This virtualization is very fast but it cannot be used for security applications (e.g. to implement sandboxes). In fact, it is easily circunventable. V^2 uses pure_libc to support nested modules for View-OS. These modules are for View-OS like kernel modules for Linux, they should be reliable. Linux and View-OS run processes in protected environment, so that errors or malicious behavior should not create global failures, but consider their modules as faithful.

The second way is by the `ptrace` system call. `ptrace` has been designed for debuggers, but it can be used to implement virtual machine monitors. User-Mode Linux and V^2 View-OS implementation `umview` are based on `ptrace`. This method can be use to create protected environments and requires several context switches per virtualized system call thus it is slower than the other methods. `ptrace` has not specifically been designed for virtualization then some features cannot be virtualized (like SIGSTOP/SIGCONT signals). On the contrary this method is highly portable as the system call is provided in all Linux kernels. V^2 implements (and proposes for standardization) some patches for the Linux kernel to enhance the features of `ptrace` for a better support of virtual machines.

The kernel support is fast and creates a safe environment for virtualized processes. V^2 has a specific module `kmview` to support system call capturing and virtualizing at kernel level. `kmview` is based on Roland McGrath's `utrace` tracing feature for kernel modules.

3.5 ★Partial Virtual Machines

The literature about Virtual Machines usually defines two class of virtual machines:

- system (or platform) virtual machine which supports the boot of a complete operating system

- process (or application) virtual machine which runs one program.

In this classification User-Mode Linux, Kvm, VirtualBOX belong to the former class, being able to boot an entire operating system and giving the abstraction of an entire computer albeit software implemented. On the other side the java virtual machine is a notable example of the latter class.

The View-OS model provides a kind of virtual machine that does not fit neither of the previous categories. In fact View-OS machines do not boot any operating systems so cannot be classified as system virtual machines. At the same time View-OS allows users to start their own programs, to run shells. Apart from the boot phase, users can work on a View-OS machine as they were working on a different computer, maybe sharing some or all their resources with the hosting machine. In this sense View-OS machines cannot be classified as process machines.

We have named the new class: "Partial Virtual Machines". The focus of the definition is related to the idea that a virtual machine can be decomposed and a virtual and real machine can run providing a mix of shared and private resources.

While the development of general purpose partial virtual machines is a development of the Virtual Square lab, the idea as been applied in the past in many specific tools. The POSIX system call chroot (privileged instruction) defines the root of the file system as viewed by a process (ad its offspring). Users can run different GNU-Linux distribution on the same kernel and sharing the same networking stack just by using chroot. There are other tools that provide the virtual installation of software (klik) or mount of file systems (fuse/libguestfs). The presence of these tools, together with many others, show the effectiveness and the need for such kind of virtualization.

These tools provide specific solutions, have not been designed to interoperate, have specific interfaces and often require root access.

View-OS by the idea of general purpose partial virtual machine provides a modular, integrated solution of all these virtualizations. In fact, as it will be clear in the examples included in the following chapters, the features of the existing tools have been or can be implemented in View-OS.

The most important goal of Virtual Machines is security and Partial Virtual Machines are not an exception.

There are several perspectives on security. The most cited property of a Virtual Machine is the creation of a sandbox: a closed environment that confinates the effects of the processing running inside a virtual machine. View-OS could create sandboxes provided that all the services gets virtualized and that the host operating system provide a specific support [1].

On the other hand Partial Virtual Machines enhance the opportunities for the user. It is possible to use several networking stacks, to test applications without installing them in the system (or better, just by doing a virtual installation), to mount a file system or to use a device whose driver is not provided for the O.S. kernel in use, to run different software distributions at the same time.

But it is not just a matter of convenience, the use of a partial virtual machine gives a higher degree of security for the whole system. The definition of different roles among the system users gives to all the UNIX derivatives a good level of protection from the execution al malicious code. The least privilege principle says that a user must be guaranteed no more privileges than necessary to complete the processing he/she needs.

A root setuid executable is a danger for the system as it provides all the privileges to any user, although just to run that specific program. Programs may include bugs, and in such a case a setuid program can be a trampoline to get the full control of a system. In the same way all the time we abuse of the root access on our systems we weaken the security of the system itself.

An example is the case of "debootstrap" a program to install a Debian distribution in a chroot. Unfortunately the script must run as root, then a typo in a pathname can mess up the entire system. "febootstrap", the sibling program for Fedora, does not need root access as it uses fakeroot and fakechroot: tools of special purpose partial virtualization.

View-OS provide a structured general purpose opportunity to solve all the protection problems like the one of "debootstrap".

3.6 Microkernels and Monolithic kernels are not mutually exclusive

The debate between monolithic kernels and microkernels has been animating the Operating System community for more than two decades. There are also famous mail threads (or duels) on the topic like the one between Andy Tanenbaum and Linus Torvalds.

Pure microkernels could be extremely flexible but less efficient than microkernels. For this reason although monolithic kernels have considered as obsolete since the beginning of nineties, they are already the most used operating systems. There are also Hybrid operating systems like Windows NT and derivatives and MacOS X. Pure microkernels have not left the status of research prototypes.

[1] The kernel module of kmview is not complete yet, some features are still implemented in a non secure mode

What Virtual Square aims to add to this debate is the novel idea that an Operating System can be designed to support both servers like in microkernels and linked in modules like in monolithic kernels. We propose to study how to design operating systems where the choice between micro vs monolithic can be during the configuration phase: instead of a pre-defined architecture defined by the operating system designer, the choice is up to the system administrator and can be adapted from case to case depending on the application.

View-OS provides user-mode modules which implements device drivers, file system implementations, networking stacks. These modules are conceptually very similar to microkernel servers. The support is not complete yet, the design of the support by the host kernel must evolve in terms of performance and insulation, but the idea seems promising.

In this way the choice is flexible and back compatible. The system administrator could decide to use some microkernel-like servers on any computer architecture and using any I/O device currently supported by the Linux kernel. Eventually there will be possible to have something very similar to a microkernel by having all the device drivers and services running as View-OS modules.

3.7 Inter Process Networking: the need for multicast IPC

Berkeley sockets API was designed mainly for client-server or peer-to-peer communication. When V^2 faced the problem to write a fast kernel support for vde, we discovered that multicast is not supported for inter process communication (IPC). Multicast in this context means that messages should be delivered to some of (or none, or all) the listening processes following some policy. The means to have multicast IPC using a Linux systems are the following:

- a user-level server/dispatcher. this is the method used by VDE but also by DCOP, dbus, jack.
- `PF_NETLINK` sockets in multicast mode. These sockets have been designed fot network stack configuration, network filtering, etc. It is possible to use them for IPC but the use is restricted to the superuser as no access control is currently implemented.
- IP multicast. TCP-IP has the support for packet multicast. Using multicast channels with zero Time-to-Leave (TTL) the network packets cannot be trasmitted on real networks thus it can be used for IPC. IP-Multicast has been designed for netorking on public networks, the only way to have access control is the content encryption.

V^2 designed Inter Process Networking, a IPC support like `AF_UNIX`, but with multicast support. IPN shares the idea of `AF_UNIX` to use the file system for naming: IPN sockets appear as special files in the file system like `AF_UNIX` sockets. Standard file permissions can be used for user access control.

An IPN instance is a multicast domain for processes running on the same operating systems. THe simplest policy for an IPN is broadcasting, all the messages get received by all the processes but the sender. It is possible to define more specific delivery policies by loading kernel modules.

User processes can comminucate using IPN as this support does not require any privilege to run, like AF_UNIX sockets.

THe Berkeley socket API has been extended to support multitasking. `listen` and `accept` calls are undefined for AF_IPN sockets, as there are nither servers nor clients. Processes can define/redefine AF_IPN channels, can join a channel and can exchage data.

`bind` is used to define and administer IPN channels while `connect` is for joining a channel. Read and write permissions in the IPN socket file define the authorization to join a IPN doe receiving or sending data. The execution permission for IPN sockets stands for the ability to execute a `bind` on the channel.

IPN can be used for peer-to-peer broadcasting/multicasting, where all the peers execute both `bind` and `connect`, or there can be a one broadcaster (or some broadcsters) and several spectator processes allowed only to receive. Each process can join and leave an existing socket at any time.

Two different policies can be set up for buffer overflow: in *lossy* services processes too slow to receive can loose messages, while in *lossless* services the send operation can block the sender if there are slow receiving processes.

Part II

Virtual Square Networking

Chapter 4

VDE: Virtual Distributed Ethernet

As noted in the introduction, one of Virtual Square's primary goals is *communication*. This implies the creation of tools for interoperability and connection between machines; both virtual and real. We found it interesting to observe that while several virtual machines provide some kind of networking abstraction, others, typically Process Virtual Machines, don't provide any networking support. The aim of the Virtual Square Team's networking research was to create a communication means for *any kind* of virtual machine. Hence, to be effective our tool had to be consistent with the network abstraction provided by any type of virtual machine.

Among the set of networking-enabled VMs, the use of an Ethernet virtual interface was the most common choice. The Virtual Square networking support had to therefore be an Ethernet-compliant virtual network, able to interconnect virtual machines running on different hosting computers. These considerations led to the creation of Virtual Distributed Ethernet (VDE[1]), one of the first components of the V^2 Framework.

VDE is an Ethernet-compliant, virtual network, able to interconnect virtual and real machines in a distributed fashion, even if they are hosted on different physical hosts. Given VDE's design goal of being an effective platform for virtual machine interoperability, its main objectives are:

- The behavior must be consistent with a real Ethernet network.

- The interconnection among most types and actual implementations of virtual machines, specific networking applications, and other virtual connectivity tools must be possible. A side effect of this objective is that VDE should also enable interoperability with real networks.

[1]Virtual Distributed Ethernet is not related in any way with www.vde.com: Verband der Elektrotechnik, Elektronik und Informationstechnik, the German Association for Electrical, Electronic & Information Technologies.

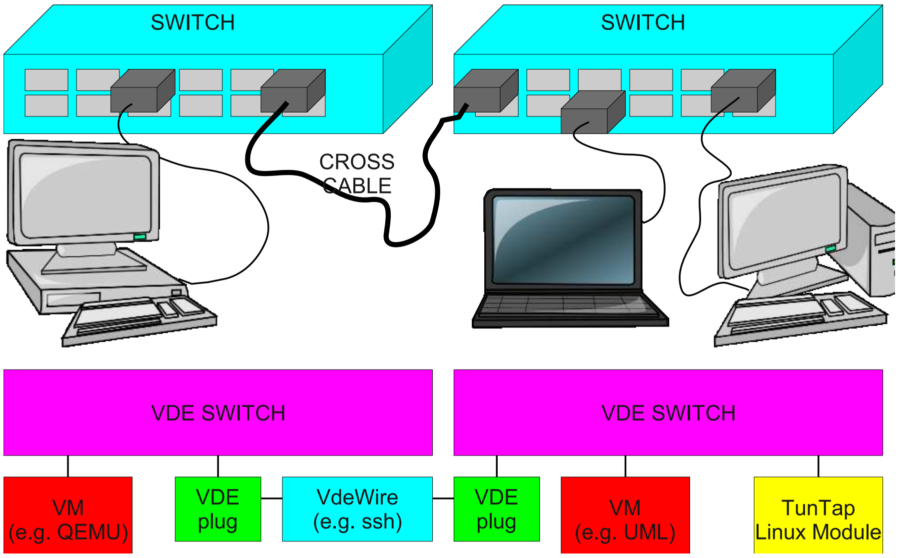

Figure 4.1: Comparison between a real Ethernet and a VDE

- Connected nodes may be geographically disperse. VDE must support distributed networking, and be able to build overlay networks on top of existing networks, with any kind of topology.

- No kernel-level addition to the operating system was to be used: VDE *must* run in user-mode. Superuser intervention is required only to connect virtual networks to real networks and only when virtual networks need public-addressing or server port numbers (< 1024).

VDE meets these objectives. Figure 4.1 illustrates VDE's consistency with real Ethernet networks.

4.1 ★VDE Main Components

The structure of a VDE network is isomorphic to a real network's. In fact, the architectural tools and devices are the same.

The basic VDE tools are:

VDE switch: The primary component used to build a virtual network. Similar to a real Ethernet switch, it has several *virtual* ports where virtual machines, applications, virtual interfaces, connectivity tools, or even other VDE switches can be plugged in.

VDE plug: The component used to plug into a VDE switch. The plug acts as a "pipe." Data streams coming *from the virtual network to the plug* are redirected to the standard output, while data streams coming *from the switch as a standard input to the plug*, are sent into the VDE network (see Fig. 4.2).

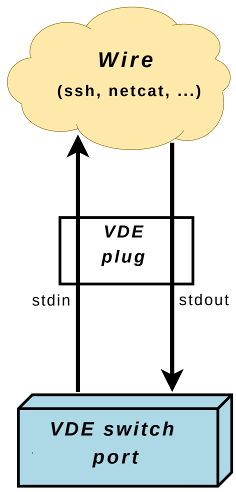

Figure 4.2: VDE plug (point-to-point)

VDE wire: Any tool able to transfer a stream connection (e.g. `cat`, `netcat`, `ssh`).

VDE cable: the conduit by which VDE switches are connected to each other. As in a real network, this device is composed of a VDE wire and two VDE plugs.

VDE cryptcab: An encrypted VDE cable. Although it is possible to use tools such as `ssh` or `cryptcat` to obtain encrypted wires connectable to VDE plugs, these existing tools work with connection-oriented streams which interfere with the underlying network stream. Hence using these existing tools results in poor performance, lost packets, and high jitter. The goal for cryptcab was to provide an encrypted cable facility in order to enable the adoption of connectionless protocols as wires.

VDE wirefilter: A program that simulates network problems. It attempts to mimic the limitations and errors of real wired connections such as noise, bandwidth limitations, etc. This tool is very useful for network testing purposes.

4.2 ★VDE Connectivity Tools

Heterogeneous virtual machine interconnectivity is accomplished through the following tools developed for use with VDE to support third party VMs.

libvdeplug: A library which provides a simple and effective programming interface to connect virtual machines and applications to a virtual network. Currently, by using this library, virtual machines including QEMU, KVM, User-Mode Linux and VirtualBox can posses virtual interfaces connectable to VDE virtual networks.

libvdetap: A pre-loadable library which uses `libvdeplug` and enables virtual machines that use the Linux tun/tap interface to connect to a VDE switch. This method has been used to connect PearPC.[2]

slirpvde: A *slirp*[3] interface to a VDE network. Our `slirpvde` shares some of the original slirp's code in addition to its basic idea. A `slirpvde` active on a VDE network acts as a NAT/masquerading gateway for the virtual network towards the real network. This way all the entities of the VDE network can be connected to the real network of the computer that hosts `slirpvde`. `slirpvde` not only provides IPv4 connectivity, but can also run a DHCP server for an easy autoconfiguration of the virtual machines.

slirpvde6: An implementation of slirp based on LWIPv6 (see 5.3). It shares the same basic ideas of slirp but supports both IPv4 and IPv6.

vdeqemu/kvm: A tool that works as a wrapper for running QEMU and KVM virtual machines, and connects them to a specified VDE switch. This tool is obsolete as native VDE support is now available in both tools.

Moreover, like the majority of professional, real network switches, VDE switches can be configured and monitored via their telnet and Web interface *vdetelweb*.

The VDE suite of programs[4] can interconnect, among others, KVM, VirtualBox, Qemu, User-ModeLinux, and μ/MPS virtual machines. Support for Xen and GXemul is under development. Furthermore, Qemu, KVM, User-Mode Linux and VirtualBox all provide native VDE support.

VDE is supported and distributed (at different versions) by Debian, Gentoo, FreeBSD, MacOSX, and is available for many other distributions. Development takes place on i386 and powerpc machines but the supported architectures are Alpha, AMD64, ARM, HPPA, Intel x86/32 and 64, IA64, Motorola M68k, MIPS, Mipsel, S/390 and Sparc.

VDE switches (starting with version 2.0) support both VLANs[5] and 802.1Q encoding of several subnets on the same port. Linux based Virtual Machines (e.g. User-Mode Linux) can define several virtual interfaces on one VDE connection. A Fast Spanning Tree protocol[6] has been implemented so that VDE networks can have cycles in their topology. Furthermore, there is an automatic reconfiguration when a link disappears.

[2] Support developed by Pierre Letouzey of the PPS laboratories, Paris VII University.

[3] Slirp was originally created by Danny Gasparovski, a student at Canberra University (Australia) in 1995. At that time, Internet providers gave different services at different prices. From a text only terminal line on one end to a full fledged Internet connection on the other. Slirp converted a terminal line to a SLIP/PPP connection so it was possible to emulate a SLIP/PPP connection over the terminal line. All the TCP connections (or UDP packets) requested from the terminal line, now converted in SLIP/PPP mode, were converted into TCP connections (or UDP packets) generated by the slirp process itself.

[4] Currently at version 2.3.1.

[5] The ability to partition available switch ports into subsets. Each subset is called a Virtual LAN or VLAN.

[6] The Fast Spanning Tree protocol is an enhancement on the Spanning Tree Protocol which cuts down on data loss and session timeouts. This protocol allows bridges used to interconnect the same two computer network segments to exchange information so that only one of them will handle a given message.

Finally, VDE supports the SNMP protocol used in network management systems to monitor network-attached devices for conditions that warrant administrative attention.

4.3 ★VDE: A Closer Look

vde_switch

While the `vde_switch` is the main component of a VDE network, its setup is both simple and intuitive. Its main features, which were briefly enumerated in the previous section, are:

VLAN: VDE switches support the creation of VLANs. Therefore it's possible to partition the set of switch ports into subsets, called Virtual LANs or VLANs. With this logical division of the virtual network it is possible to have several independent logical networks within the same virtual switch. As with real-world switches, this feature makes it possible to separate/segregate the network traffic between hosts on the same VLAN and hosts that belong to other VLANs.

Command line management: The `vde_switch` provides a command line interface for switch management; both from a socket (when running the switch as a detached process) and from standard input (when running the switch as a foreground process). The command line interface is useful to monitor switch ports, status and sockets, to create VLANs, and to enable the Fast Spanning Tree Protocol.

Fast Spanning Tree Protocol: This protocol has been implemented in the VDE switch to prevent loops. As in real switched networks, the protocol finds a spanning tree for the mesh network and disables links that are not included within the spanning tree. When FSTP is running, ports can be given roles:

- *Root:* forwarding port, elected for the spanning-tree topology.
- *Designated:* forwarding port for every lan segment.
- *Backup/Alternate:* A redundant path to a segment connected with another bridge port, or an alternate path to the root switch.
- *Edge:* port that doesn't take part in the network topology, and that doesn't influence the Spanning Tree computation.
- *Unknown:* Unidentifiable role.

A VDE switch is activated using the command `vde_switch`. Table 4.1 shows the command's customizable options. If the switch is running in the foreground (and not as a daemon with the `--daemon` option), it keeps control of the terminal and provides a console configuration interface; the prompt appears by simply typing return.

```
$ vde_switch

vde: _
```

Option	Description
`--numports N`	number of ports
`--hub`	turns off switch mode and works as a hub
`--fstp`	activates Fast Spanning Tree Protocol
`--macaddr MAC`	sets the switch MAC address
`--priority N`	priority for FST
`--hashsize N`	sets the hash table size
`--daemon`	run switch as a daemon
`--pidfile PIDFILE`	writes the pid of the daemon to PIDFILE
`--rcfile FILE`	config file, overrides /etc/vde.rc and ~/.vderc
`--mgmt SOCK`	path for the management socket
`--mgmtmode MODE`	permissions for the management socket
`--sock SOCK`	path for the communication socket
`--mod MODE`	permissions for communication sockets
`--group GROUP`	group owner for communication sockets
`--tap TAP`	sets the tap interface

Table 4.1: `vde_switch` options

Typing `help` at the prompt will display a list of possible commands and options, as shown in Fig. 4.3.

Each switch is associated with a working directory, which is also used to uniquely identify each switch. The default value is `/tmp/vde.ctl` for user activated switches, while `/var/run/vde.ctl` is the default for the VDE system daemon.

As shown in table 4.1, it is possible to run a switch with a different working directory (also known as its socket directory) with the `--sock` command:

```
$ vde_switch --sock /tmp/myvde.ctl
```

This directory structure is a new feature introduced in VDE v2.0. Earlier versions of the `vde_switch` were an extension of the `uml-switch` written by Jeff Dike, a networking feature created for User-Mode Linux. VDE v.1.0 used a socket and not a directory to name a switch and an unnamed socket for data exchange. That version had security problems because the socket could be removed by users; the secret name of the unnamed socket being easily determinable through the process id. There were also compatibility problems because unnamed sockets do not exist on several POSIX compliant kernels, e.g. FreeBSD and MacOSX.

VDE v2.0 fixes this problems: the directory can be protected to avoid Denial Of Service attacks and can keep all the communication sockets secure and reserved.

The VDE switch also supports a configuration file (using the `-f` command line option) with the same syntax of the console commands.

`vde_switch` Access Control

A `vde_switch` supports two types of access control: global access control and port access control.

```
$ vde_switch
vde$ help
0000 DATA END WITH '.'
COMMAND PATH       SYNTAX          HELP
------------       --------------  ------------
ds                 ============    DATA SOCKET MENU
ds/showinfo                        show ds info
help               [arg]           Help (limited to arg when specified)
logout                             logout from this mgmt terminal
shutdown                           shutdown of the switch
showinfo                           show switch version and info
load               path            load a configuration script
debug              ============    DEBUG MENU
debug/list                         list debug categories
debug/add          dbgpath         enable debug info for a given category
debug/del          dbgpath         disable debug info for a given category
plugin             ============    PLUGINS MENU
plugin/list                        list plugins
plugin/add         library         load a plugin
plugin/del         name            unload a plugin
hash               ============    HASH TABLE MENU
hash/showinfo                      show hash info
hash/setsize       N               change hash size
hash/setgcint      N               change garbage collector interval
hash/setexpire     N               change hash entries expire time
hash/setminper     N               minimum persistence time
hash/print                         print the hash table
hash/find          MAC [VLAN]      MAC lookup
fstp               ============    FAST SPANNING TREE MENU
fstp/showinfo                      show fstp info
fstp/setfstp       0/1             Fast spanning tree protocol 1=ON 0=OFF
fstp/setedge       VLAN PORT 1/0   Define an edge port for a vlan 1=Y 0=N
fstp/bonus         VLAN PORT COST  set the port bonus for a vlan
fstp/print         [N]             print fst data for the defined vlan
port               ============    PORT STATUS MENU
port/showinfo                      show port info
port/setnumports   N               set the number of ports
port/sethub        0/1             1=HUB 0=switch
port/setvlan       N VLAN          set port VLAN (untagged)
port/create        N               create the port N (inactive|notallocatable)
port/remove        N               remove the port N
port/allocatable   N 0/1           Is the port allocatable as unnamed? 1=Y 0=N
port/setuser       N user          access control: set user
port/setgroup      N user          access control: set group
port/epclose       N ID            remove the endpoint port N/id ID
port/resetcounter  [N]             reset the port (N) counters
port/print         [N]             print the port/endpoint table
port/allprint      [N]             print the port/endpoint table (including inactive port)
vlan               ============    VLAN MANAGEMENT MENU
vlan/create        N               create the VLAN with tag N
vlan/remove        N               remove the VLAN with tag N
vlan/addport       N PORT          add port to the vlan N (tagged)
vlan/delport       N PORT          add port to the vlan N (tagged)
vlan/print         [N]             print the list of defined vlan
vlan/allprint      [N]             print the list of defined vlan (including inactive port)
.
1000 Success
vde$
```

Figure 4.3: VDE command line interface options. Note: The actual command line interface set may differ depending upon the version of the tool and options enabled during the compilation.

Global access control permits or denies the access to the entire switch. Using the `-g` and `-m` command line options one can set user/group access permissions. The former defines the group while the latter the access mode of the communication sockets of the switch.

Switch ports can be limited to specific users or groups as well. One might use this feature to provide vlan protection. The specific command line options for this are `port/setuser` and `port/setgroup`.

The access control algorithm used is the following:

- If there is no user/group limitation for the port (port.user==NONE and port.group==NONE): allow;

- if the request comes from root or from the specified user (user==root or user==port.user): allow;
- if the request comes from a user belonging to the specified group (user belongs to port.group): allow;
- otherwise deny.

Note: While `vde_switch` is backward compatible with `uml_switch` under user mode linux, specific port access control is not supported. Since user-mode linux kernels now provide native `vde` support including port access control, older versions can only be connected to ports without access control (port.user == NONE and port.group == NONE).

vdeterm

VDE management can be accomplished using any tool that can interact with AF_UNIX stream sockets; an easy task given that the protocol is ASCII-based. For example, one can use the `socat`[7] tool available in many distributions. For a switch started with the option `-M /tmp/vde.mgmt`, the command to interact with the management console is:

```
$ socat READLINE /var/vde.mgmt
```

`vdeterm` is a terminal for VDE switches and `wirefilter`s, which like `socat` provides command history and editing, additionally provides specific VDE management features.

For the same management socket the command to start `vdeterm` is:

```
$ vdeterm /var/vde.mgmt
```

The following VDE specific features of `vdeterm` are:

- Processing and hiding of the protocol numeric codes.
- Command completion: the tab key completes the command and double tab provides a list of the available commands for a given prefix.
- Asynchronous debug messages do not interfere with standard output and command editing.

vde_plug

A `vde_plug` is designed to behave as a physical Ethernet plug, connects to a `vde_switch`. Everything that is injected into the plug from standard input is sent into the `vde_switch` it is connected to. On the other side, everything that comes from the virtual network, through the `vde_switch` to the plug, goes to the `vde_plug`'s standard output.

Two `vde_plug`s are connected with a simple but powerful tool developed to work in a VDE environment. `dpipe`, also known as a *bi-directional pipe*, is able

[7] *http://www.dest-unreach.org/socat/*

to run two or more commands diverting standard output of the first command into the standard input of the second command and vice-versa.[8]

For example:

```
$ dpipe vde_plug /tmp/vde1.ctl = vde_plug /tmp/vde2.ctl
```

shows how it is possible to connect two `vde_switch`es by running two `vde_plug`s via the bi-directional channel provided by `dpipe`. Each plug is connected to a VDE control socket.

Alternatively, the two `vde_switch`es do not need to reside on the same physical host:

```
$ dpipe vde_plug /tmp/vde.ctl = ssh foo@remote.host.org \
       vde_plug /tmp/vde_remote.ctl
```

In this example the VDE is actually distributed over a real network: a `vde_switch` running locally is connected to a remote `vde_switch` using a secure shell channel. This is done by simply running a `vde_plug` connected to the remote VDE control socket on the remote host.

Virtually any program able to provide a bi-directional channel both remotely or locally can be used as a `vde_wire` to connect VDE networking components. For example, one can use an unencrypted UDP channel built using the `netcat` utility.

An instance of `netcat` connected to a `vde_switch` waiting for incoming connections on the remote machine:

```
$ dpipe vde_plug /tmp/vde.ctl = nc -l -u -p 8000
```

Or having the `netcat` client connect with the remote `waiting-netcat`: The standard input and output are connected with `dpipe` to the local `vde_switch`:

```
$ dpipe vde_plug /tmp/vde_local.ctl = nc -u 8000 \
       remote.host.address.org
```

A Tunnel Broker via VDE

A `vde_plug` can also be used to implement a VDE tunnel broker. While public services of VDE switches can be set up, and users can join the net; users will not allowed to log-in to the remote computer.

The first step is to define `vde_plug` as a login shell. For this, add a line with the complete path of `vde_plug` at the end of `/etc/shells`:

```
# /etc/shells: valid login shells
/bin/bash
.....
/usr/bin/vde_plug
```

[8] The `dpipe` command is part of the VDE suite simply because it did not exist elsewhere and it was felt it would be a useful tool. A VDE wire connection requires that the two commands have a bi-directional communication channel in place: The output of the first command must be the input for the second and vice versa. The VDE developers have implemented the dual pipe `dpipe` command in order to meet this requirement. The two commands are separated by an "=" sign.

`vde_plug` must be set as the login shell for the user(s) of the tunnel broker service. For example it is possible to add a user named `vde` and change its shell by editing the `/etc/passwd` file. `vde`'s line should appear as the following one:

```
vde:x:1003:1003:vde,,,:/home/vde:/usr/bin/vde_plug
```

If somebody tries to log in as `vde`, typing the correct password, she will receive the following error message:

```
This is a Virtual Distributed Ethernet (vde) tunnel broker.
This is not a login shell, only vde_plug can be executed
```

While no direct log-in is permitted, it is possible for the `vde` user to join a standard switch on a remote machine. A password is still required:

```
$ dpipe vde_plug /tmp/vde.ctl = ssh vde@remote.host.org vde_plug
```

Setting up a service without a password is a bit more involved. The line in `/etc/passwd` for the `vde` user should now appear as:

```
vde::1003:1003:vde,,,:/:/usr/bin/vde_plug
```

There is now no need for a password nor a home directory for the `vde` account. The ssh daemon, though, should be configured to "allow" access for empty password accounts. (For security reasons, default ssh daemon configurations typically deny such access.) It is important to note that machines allowing remote access to empty password accounts need to be carefully controlled and updated to prevent intrusions. To permit our `vde` account to enter the system without any password on OpenSSH, edit the line in `sshd.conf`:

```
# To enable empty passwords, change to yes (NOT RECOMMENDED)
PermitEmptyPasswords yes
```

This by itself is insufficient if one is using PAM, the centralized authentication service. For those using PAM, PAM needs to also be configured to allow no-password users. In Debian, for example, this can be accomplished either by changing in the file `/etc/pam.d/common-auth` the line:

```
auth required  pam_unix.so nullok_secure
```

into

```
auth required  pam_unix.so nullok
```

or by adding "pts" lines to the file /etc/security

```
pts/0
pts/1
....
pts/255
```

It is also possible to set up a tunnel broker (on a real or virtual machine) for the users of a NIS domain. To do this, add the following to the end of `/etc/passwd`:

```
+::0:0:::/usr/bin/vde_plug
```

In this case each user can log in with her password on to the tunnel broker. Furthermore, password changes or certificate based accesses will also be granted, but the only service permitted on the tunnel broker is the connection to a VDE switch.

vde_cryptcab

In the previous examples, tools like `ssh` or `netcat` were used to interconnect remote `vde_switch`es. Although these two tools are very simple and intuitive to use, they each have their shortcomings. `netcat` creates unencrypted connections and for this reason does not protect users from traffic sniffing and intrusion. On the other hand, `ssh` provides protection from traffic sniffing because the traffic transferred with `ssh` is encrypted, but exhibits poor performances.

`vde_cryptcab`[9] was developed within the Virtual Square Project to provide a secure and efficient tool to interconnect VDE networking components distributed over different machines or different underlying networks. `vde_cryptcab` uses `ssh` to exchange a secret key and then creates an encrypted UDP connection. Validity checks have been added to each packet in order to prevent intrusions and record&playback attacks.

To start a `vde_cryptcab` server connected to a `vde_switch` on a remote machine:

```
$ vde_cryptcab -s /tmp/vde.ctl -p 12000
```

This `vde_cryptcab` server will accept UDP datagrams on port 12000 with multiple connections authenticated via `ssh`. It is also possible to connect multiple remote `vde_cryptcab` clients to the same `vde_cryptcab` server. All datagrams are sent to UDP port 12000. On the client side, one connects to the server via the following:

```
$ vde_cryptcab -s /tmp/vde_local.ctl \
       -c username@remote.host.org:12000
username@remote.host.org's password:
.blowfish.key            100%   24     0.0KB/s   00:00
```

Note that during initialization a blowfish secret key has been transferred to the remote `cryptcab` server. The key will be used to encrypt UDP datagrams *from* and *to* the server.

wirefilter

`wirefilter` is another useful tool for testing purposes. This tool simulates problems, and the limitations and errors of real wired connections, such as noise, bandwidth, etc. This VDE environment specific tool can be inserted into a bi-directional pipeline (say between two `vde_plug`s) that interconnects two `vde_switch`es, to introduce virtual errors or place limits on the line. `wirefilter` can control the connection parameters shown in table 4.2.

Since `wirefilter` works on bi-directional channels it is also possible to fine-tune the filtering by choosing which direction of the stream is affected by the `wirefilter` settings.

[9] Daniele Lacamera is the primary `vde_cryptcab` code author.

Option	Description
`--loss`	percentage of packet loss
`--lostburst`	length of lost packet burst
`--delay`	extra delay on packet transmission
`--dup`	percentage of duplicated packets
`--bandwidth`	channel bandwidth
`--speed`	interface speed
`--capacity`	maximum capacity of packet queue
`--mtu`	maximum transmission unit
`--noise`	corrupted bits per megabyte
`--fifo`	packet sorting

Table 4.2: `wirefilter` options

A typical example of `wirefilter` usage might be:

```
$ dpipe vde_plug /tmp/vde1.ctl = wirefilter \
       -M /tmp/wiremgmt = vde_plug /tmp/vde2.ctl
```

In this example `wirefilter` is in the middle of a bi-directional pipe that connects two `vde_switch`es via two `vde_plug`s. It is possible to differentiate filtering between the *left-to-right* and *right-to-left* channels. Note that like a `vde_switch`, `wirefilter` can also specify a unix socket to manage filter settings at runtime via `vdeterm`.

It is also possible to connect `wirefilter` directly to switches:

```
$ wirefilter -v /tmp/vde1.ctl:/tmp/vde2.ctl
```

This command connects the same two switches of the previous example. In this case stdin and stdout are not used for communication: instead access is provided from the console to the `wirefilter` management.

Finally, one can use `wirefilter` in combination with `dpipe` to join a remote network (In this example `-` means stdin/stdout.):

```
$ dpipe wirefilter -v /tmp/vde1.ctl:- -M /tmp/wiremgmt = ssh remote.host vde_plug
```

The various `wirefilter` options allow one to set limits on performance; e.g. bandwidth and speed. When the capacity of a channel is exceeded, packets are dropped. Each option can be set separately for each direction, e.g. a loss ratio set to `LR10` means 10% left-to-right (referring to the sides of the filter on the `dpipe` command). Furthermore, each option can be set with a statistical approximation. A delay `100+100` means a random number between 0 and 200 msec (`+` should be read as the ± sign used in Mathematics/Physics). It is also possible to add a trailing letter to specify the type of statistical distribution used: `100+50U` means uniform distribution in the range 50-150 (U can be omitted since it is the default choice), `100+50N` is a Gaussian distribution centered around 100 with more than 98% of the samples in the range 50,150.

The option `lostburst` enables the Gilbert model for bursty errors. The value is the mean length of the lost packet bursts. The Gilbert model is based on a two state Markov chain. The states are *working* and *faulty*. Naming l the loss ratio and b the mean burst length, the probability to leave the faulty state is $1/b$, the probability to enter the faulty state is $l/(b-(1-l))$. In this way the loss rate converges to the l.

`wirefilter` also provides a more complex set of parameters using a Markov chain to emulate the different states of the link and the transitions between states. Each state is represented by a node. Markov chain parameters can be set directly with management commands or via "rc" files. A command line interface, due to the large number of parameters, would quickly become unwieldy/unworkable.

markov-numnodes *n*
: Defines the number of different states. All the parameters of the connection can be defined node by node. Nodes are numbered starting from zero (to n-1).

```
delay 100+10N[4]
loss 10[2]
```

This command defines a delay of 90-110 ms (normal distribution) for node number 4 and a 10% loss for node number 2. It is also possible to resize the Markov chain at run-time. New nodes are unreachable and do not have outgoing edges to any other state. (i.e. Each new node has a loopback edge to the node itself with 100% probability). When reducing the number of nodes, the weight of the edges directing into a deleted node is added to the loopback edge. When the current node of the emulation is deleted, node 0 becomes the current node.

markov-time *ms*
: Time period (ms) for the markov chain computation. Each *ms* microseconds, a random number generator decides which is the next state (default value=100ms).

markov-name $n, name$
: Assign a name to a node of the markov chain.

markov-setnode *n*
: Manually set the current node to node n.

markov-setedge n_1, n_2, w
: Define an edge between n_1 and n_2, with edge weight w (probability percentage). The loopback edge (from a node to itself) is always computed as 100% minus the sum of the weights of outgoing edges.

showedges *n*
: List the edges from node n (or from the current node when the command has no parameters). Null weight edges are omitted.

showcurrent
: Show the current Markov state.

showinfo *n*
: Show status and information for state (node) n. If the parameter is omitted, display the status and information for the current state.

markov-debug *level*
: Set the debug level for the current management connection. In the actual implementation when the level is greater than zero each change of markov node causes the output of a debug trace. Debug tracing get disabled when level is zero or the parameter is missing.

`vde_plug2tap`

`vde_plug2tap` is another "plug" tool that can be connected to a `vde_switch`. Instead of using standard input and standard output for network I/O, everything that comes from a `vde_switch` to the plug is redirected to the specified tap interface. In the same way, everything injected into the tap interface is redirected to the `vde_switch`.

```
$ vde_plug2tap --daemon -s /tmp/myvde.ctl tap0
```

It is also possible to attach a tap interface during `vde_switch` creation to obtain the same result, with the `--tap` option.

*** I question the need for this section *** (MG)

`vdeqemu/vdekvm`

Note: New versions of qemu and kvm have already built-in vde support. There is no need of vdeq, but the syntax is the same.

These tools are wrappers for running qemu/kvm virtual machines and are used to connect them to a `vde_switch`. Basically they call qemu/kvm with the correct network parameters by re-writing the command line. The only thing to know is the path for the desired `vde_switch` to connect to: vdeqemu/vdekvm launch qemu/kvm with the desired number of emulated network interfaces connected to the specified vde_switch(es).

The only requisite is that a `vde_switch` must be running and ready to accept connections on its control socket. Note that in this case the `vde_switch` is connected to a preconfigured tap interface to make guest and host networks easier to reach. Here we see an example with vdeqemu (the usage of vdekvm is identical):

```
$ vde_switch -d -s /tmp/vde.ctl -t tap0 -M /tmp/mgmt
```

Once the `vde_switch` is started, a new instance of qemu can be connected. With qemu up to 0.9.1 and kvm up to 71 use the `vdeqemu` wrapper:

```
$ vdeqemu -hda /path/to/image.img -net nic \
      -net vde,sock=/tmp/vde.ctl
```

With newer qemu or kvm the syntax is:

```
$ qemu -hda /path/to/image.img -net nic \
      -net vde,sock=/tmp/vde.ctl
```

All the arguments following `vdeqemu` are specific for `qemu`, and can be changed according to the VM semantics. The commands `vdeq qemu` and `vdeqemu` are interchangeable. Through the `vde_switch` management console it is possible to check what is connected to the `vde_switch`, after vdeqemu/vdekvm has been launched:

```
$ vdeterm /tmp/mgmt
VDE switch V.2.3.1
(C) Virtual Square Team (coord. R. Davoli) 2005,2006,2007 - GPLv2

vde[/tmp/mgmt]: port/print
0000 DATA END WITH '.'
Port 0001 untagged_vlan=0000 ACTIVE - Unnamed Allocatable
  -- endpoint ID 0006 module tuntap      : tap0
Port 0002 untagged_vlan=0000 ACTIVE - Unnamed Allocatable
  -- endpoint ID 0007 module unix prog   : vdeqemu \
       user=fcenacch PID=15421 SOCK=/tmp/vde.15421-00000
.
Success
```

By default qemu uses the same MAC address for every virtual machine, so if the user plans to use several instances of qemu, it is necessary to set a different MAC address for each virtual machine.

slirpvde/slirpvde6

`slirpvde` and `slirpvde6` are slirp interfaces for VDE networks. As discussed above, `slirpvde` or `slirpvde6` acts like a networking router connected to a `vde_switch` and provides connectivity from the host where it is running to the virtual machines inside the virtual network. `slirpvde` is not the only way for virtual machines within a VDE network to communicate with the outside world, but it is the only way to do so that does NOT require root privileges. (e.g. A tun/tap interface.)

its main feature is that it can be run using standard user privileges.

Every connection from a machine within the virtual network to `slirpvde`'s internal address is translated, masqueraded and re-generated by `slirpvde` and redirected to host machine stack. Like most of intermediate systems, it provides basic features such as a *dhcp* service, port forwarding, and remapping *dns* requests.

```
$ slirpvde -d -s /tmp/vde.ctl -dhcp
```

Launching `slirpvde` and specifying the `vde_switch` the control socket where virtual machines are connected, is enough to provide access to the external network to all virtual machines connected to the `vde_switch`. The additional `-dhcp` option tells `slirpvde` to also provide dynamic network address assignment.

`slirpvde` provides addresses in the range 10.0.2.0/24 (configurable with the`--network` option), and the default route is 10.0.2.2. There is also a DNS forwarder on 10.0.2.3 (auto configured by *dhcp*). `slirpvde` also provides port forwarding to allow incoming connections and X window forwarding.

```
$ vdeterm /tmp/mgmt
VDE switch V.2.3.1
```

```
(C) Virtual Square Team (coord. R. Davoli) 2005,2006,2007 - GPLv2

vde[/tmp/mgmt]: port/print
0000 DATA END WITH '.'
Port 0001 untagged_vlan=0000 ACTIVE - Unnamed Allocatable
  -- endpoint ID 0006 module tuntap      : tap0
Port 0002 untagged_vlan=0000 ACTIVE - Unnamed Allocatable
  -- endpoint ID 0007 module unix prog   : vdeqemu
       user=fcenacch PID=15421 SOCK=/tmp/vde.15421-00000
Port 0003 untagged_vlan=0000 ACTIVE - Unnamed Allocatable
  -- endpoint ID 0009 module unix prog   : slirpvde:
       user=fcenacch PID=15558
 SOCK=/tmp/vde.15558-00000
.
Success
```

Using the `vde_switch` management interface one can see that `slirpvde` is connected to port 3 of the `vde_switch`. On the other ports, a qemu virtual machine and a tap interface can be seen. Additionally, the *dhcp* service provided by `slirpvde` can be used to configure the tap interfaces connected to the switch.

The same example can use `slirpvde6` instead of `slirpvde`:

```
$ slirpvde6 -d -s /tmp/vde.ctl -dhcp
```

Here, `slirpvde6` provides an IPv4 slirp service. The default gateway address is 10.0.2.1/24. `slirpvde6` uses the same interface for all its services such as the *dns* forwarder or the *dhcp* server.

In the port list on the switch `slirpvde6` appears as a LWIPv6 service:

```
Port 0002 untagged_vlan=0000 ACTIVE - Unnamed Allocatable
 Current User: renzo Access Control: (User: NONE - Group: NONE)
 IN:  pkts        482          bytes                141526
 OUT: pkts       2893          bytes                189153
  -- endpoint ID 0011 module unix prog   : LWIPv6 if=vd0
  user=renzo PID=5260  SOCK=/var/run/vde.ctl/.05260-00000
```

`slirpvde6` supports several addresses, the user may specify any mix of IPv4 and IPv6 addresses:

```
$ slirpvde6 -d -H10.0.2.1/24 -H2001::1/64 -s /tmp/vde.ctl -dhcp
```

`slirpvde6` is able to act as a stateless translator: if the VDE network is IPv6 only (no IPv4 addresses), all the IPv6 requests to IPv4 mapped hosts (::ffff:0:0/96) are converted by `slirpvde6` into IPv4. Unfortunately the standard on how to support this conversion is still unsettled; the "IPv6 Addressing of IPv4/IPv6 Translators" working group of IETF is still open. LWIPv6 supports the management of IPv4 mapped addresses which is one of the working group's proposals. Unfortunately this feature is not currently implemented in the networking stacks provided by many (all?) of the primary popular and commercial operating systems.

Both `slirpvde` and `slirpvde6` support the *plug mode* or the *remote mode*. By using the option `-s -` or `-socket -` the program uses stdin-stdout (as `vde_plug`) instead of connecting to a local switch. This is useful when connecting a VDE switch to a remote network.

```
$ dpipe vde_plug = ssh remote.machine.org slirpvde6 -D -s -
```

In this final example, the command connects the default switch on the local host to a remote slirp service. In this way the VDE network is connected through a virtual NAT/masqueraded VPN to the networking services of the remote host.

4.4 ★VDE Examples

Example 1: Connecting 4 Virtual Machines to 2 Different Switches

```
1. $ vde_switch
2. $ qemu -m 256 -k en-us -boot c -hda image.hd \
       -monitor stdio -net nic -net vde
3. $ linux mem=256 ubd0=image.uml eth0=vde
```

In line 1, the switch is started in the default directory (`/tmp/vde.ctl`). In another window, it's possible to start a virtual machine and to connect it to the switch, as in line 2 with `qemu`, and line 3 with a User Mode Linux virtual machine. This configuration is the easiest to work with given the use of the switch's default directory and the use of virtual machines with native VDE support.

```
1. $ vde_switch --sock /tmp/vde2.ctl
2. $ ln -s /bin/true scripts/ifppc_up.setuid
3. $ ln -s /bin/true scripts/ifppc_down.setuid
4. $ export VDEALLTAP=/tmp/vde2.ctl
5. $ export LD_PRELOAD=/usr/lib/libvdetap.so
6. $ ppc
7. $ kvm -net nic -net vde,sock=/tmp/vde2.ctl,port=10 \
       -m 256 -boot c -hda image2.hd -monitor stdio
```

In this example, the `vde_switch` is started (line 1), but this time one specifies the socket, in order to open a *different* switch on the same host. Now there are two switches, one in `/tmp/vde.ctl`, and the other in `/tmp/vde2.ctl`. Lines 2 to 6 are the commands for opening a PearPC virtual machine.[10] There are some environment variables that can be set up to configure the `vdetap` library. VDEALLTAP signals that all the virtual tap connections must be routed to the same switch. The man page of `vdetaplib` describes all the available options. Hence on line 6 the `ppc` command starts up the PearPC VM and connects it to the second switch – PearPC uses a tuntap interface and line 5 directs all

[10]The symbolic links of ifppc_up.setuid and ifppc_down.setuid to bin/true is a necessary trick, because PearPC (and well as several other tools) run a setuid script to set up the tuntap interface.

virtual tap connections through the newly created VDE switch. Finally, line 7 starts a `kvm` virtual machine whose networking support is not only via VDE, but through the newly created switch using the non-default socket. (`kvm` and `qemu` have the identical command line syntax.)

Example 2: Connecting 2 VDE Switches

The previous example(s) can be run either on the same computer (evidently a very powerful one given its ability to support four concurrent virtual machines) or on two different (connected) computers. These examples illustrates how to connect these two switches together.

Assuming the two switches are running on the same (powerful) computer, they can be connected by any of the following four "identical" commands:

```
$ dpipe vde_plug = vde_plug /tmp/vde2.ctl
$ dpipe vde_plug /tmp/vde2.ctl = vde_plug
$ dpipe vde_plug /tmp/vde.ctl = vde_plug /tmp/vde2.ctl
$ dpipe vde_plug /tmp/vde2.ctl = vde_plug /tmp/vde.ctl
```

Alternatively, if the two switches are running on different computers, called `vdehost1` and `vdehost2`, the VDE cable needs a *longer* wire to interconnect the two `vde_plug`s. A wire can be any tool able to send a stdin/stdout stream to a remote machine. Using `ssh` one would enter:

```
vdehost1$ dpipe vde_plug = ssh vdehost2 \
       vde_plug /tmp/vde2.ctl
vdehost2$ dpipe vde_plug /tmp/vde2.ctl = ssh \
       vdehost1 vde_plug
```

Using `netcat` one would enter:

```
vdehost1$ dpipe vde_plug = nc -l -u -p 5555
vdehost2$ dpipe vde_plug /tmp/vde2.ctl = nc vdehost1 \
       -u 5555
```

Note that in the `netcat` case the communication takes place on an unencrypted UDP channel, consequently subject to intrusions and traffic sniffing.

Example 3: tun/tap Access

It is possible to connect a `tap` interface provided by the hosting operating system to a `vde_switch`. Typically, tuntap configuration is restricted for security reasons, thus this example needs to be run by root, or by a user previously authorized by a system administrator using the `tunctl` command.

```
# vde_switch --tap tap0 --daemon --mgmt /var/run/vde.mgmt \
     --mod 777
```

The switch options used are `--tap`, that connects the switch to a tap interface, and `--mod`, to set the octal-numbered permissions for the control socket (with the same octal mode used by `chmod`). Using 700 (or omitting the option), the switch gives service only to virtual machines run by root.

When a VDE switch is connected to a tap interface, VDE become indistinguishable from any other Ethernet network as seen from the kernel of the hosting computer. Thus, any kind of packet forwarding, filtering or bridging tool can be used.

Using tap interfaces, one can create Virtual Private Networks (VPN) between computers: Each computer must run a VDE switch connected to a tap interface with the two switches connected by a cable (as in the Example 2). The two tap interfaces will see each other as if they were connected on the same LAN. Any Ethernet-compliant protocol can be used in this VPN, e.g. making IPv6 tunnels in places served only with IPv4.

Example 4: VDE Tunnel Broker

A VDE tunnel broker is a computer able to provide VDE connections to users. One creates such a service by simply running a VDE switch: This way all users allowed to log-in to the computer can start a remote `vde_plug`.

Note: A VDE tunnel broker cannot be used for remote login, but just to provide VDE connectivity. Connections and IP addresses are also logged for security reasons.

In order to run a VDE tunnel broker, a number of steps must be completed. The first is that `vde_plug` must be allowed as a login shell. To do this, add a line with the complete path of `vde_plug` at the end of `/etc/shells`:

```
# /etc/shells: valid login shells
/bin/bash
.....
/usr/bin/vde_plug
```

Next, `vde_plug` must be set as the login shell for the user(s) of the tunnel broker service. A simple way to accomplish this is to add a special user whose shell is `vde_plug`. This can be done by adding the following to the `/etc/passwd` file:

```
vdeuser:x:4242:4242::/home/vdeuser:/usr/bin/vde_plug
```

Finally, make sure that a VDE switch is running on the host server.

It is also possible to configure a VDE tunnel broker for all the users of the current NIS domain with the following line added to the `/etc/passwd` file:

```
+::0:0:::/usr/bin/vde_plug
```

Currently, the machine *vde.students.cs.unibo.it*, hosted by the University of Bologna's Computer Science Department, has been running as a VDE tunnel broker for more than two years. It is used by students and researchers interested in doing tests, experiments, or just to have a VPN on the University network.

There are also public VDE networks provided by the VirtualSquare Project for experimentation. These networks are not routed to the public Internet. Instead, all of the machines (both virtual and real) connected from anywhere on the Internet to the same public VDE network communicate on a virtual Ethernet LAN.

For example, if several users run the following command:

```
dpipe vde_plug = ssh vde0@vde2.v2.cs.unibo.it vde_plug
```

they will all connect their local VDE switch to the public network #0.[11] All the virtual and real machines plugged into the users' switches will be joined to the same virtual LAN.

4.5 VDE API: The `vdeplug` Library

VDE provides a library for virtual machine developers wishing to connect their VM's to VDE networks. The interface is composed of just six functions as illustrated in Fig.4.4.

- `vde_open` opens the connection to a switch. It requires three arguments: the pathname of the switch, a description (that will be sent to the switch to recognize the virtual machine), and an optional structure for further options. It is a macro that calls the hidden function; `vde_open_real`. This solution provides compatibility with future versions of the interface: The macro automatically inserts the interface version number in the call. `vde_open` returns a handle to the VDE connection.

- `vde_read` Receives a packet.

- `vde_write` Sends a packet.

- `vde_close` Closes the connection.

- `vde_datafd` Returns a descriptor that can be used in a *select()* system call or in a *poll()* system call to test the availability of new packets to be read.

- `vde_ctlfd` Returns a descriptor that can be used in a *select()* system call or in a *poll()* system call to test whether the switch has closed the port.

The `vdeplug` library also includes support for VDE streams; the stream encoding used by the command `vde_plug`. Figure 4.5 illustrates the API of this feature.

`vdestream_open` returns a descriptor that identifies the stream in `vdestream_send`, `vdestream_recv` and `vdestream_close`.

A `vdestream` is a bidirectional filter. All the packets passed to the library by `vdestream_send` get stream encoded and sent on the descriptor `fdout`. All the data received from a stream connection and fed into the library through `vdestream_recv` gets converted into packets and forwarded using the `frecv` upcall.

VDE uses a very simple encoding; each packet is prefixed by two bytes which indicate the length of the packet. The first byte is the most significant. There is also a heuristic to re-synchronize the stream in the unexpected case of spurious data.[12]

[11] Change the username `vde0` to `vde1`, `vde2`, etc to access the other public VDE networks.

[12] While the encoding expects the stream to be reliable, sometimes there are errors on reliable streams; e.g. error messages or system alerts in `ssh` streams.

```
#include <sys/types.h>
#define LIBVDEPLUG_INTERFACE_VERSION 1

struct vdeconn;

typedef struct vdeconn VDECONN;

/* Open a VDE connection.
 * vde_open_options:
 *   port: connect to a specific port of the switch (0=any)
 *   group: change the ownership of the communication port to a specific group
 *        (NULL=no change)
 *   mode: set communication port mode (if 0 standard socket mode applies)
 */
struct vde_open_args {
    int port;
    char *group;
    mode_t mode;
};

/* vde_open args:
 *   vde_switch: switch id (path)
 *   descr: description (it will appear in the port description on the switch)
 */
#define vde_open(vde_switch,descr,open_args) \
    vde_open_real((vde_switch),(descr),LIBVDEPLUG_INTERFACE_VERSION,(open_args))
VDECONN *vde_open_real(char *vde_switch,char *descr,int interface_version,
    struct vde_open_args *open_args);

ssize_t vde_recv(VDECONN *conn,char *buf,size_t len,int flags);

ssize_t vde_send(VDECONN *conn,const char *buf,size_t len,int flags);

/* for select/poll when this fd receive data, there are packets to recv
 * (call vde_recv) */
int vde_datafd(VDECONN *conn);

/* for select/poll. the ctl socket is silent after the initial handshake.
 * when EOF the switch has closed the connection */
int vde_ctlfd(VDECONN *conn);

int vde_close(VDECONN *conn);
```

Figure 4.4: `vdeplug` Library: libvdeplug.h

```
struct vdestream;

typedef struct vdestream VDESTREAM;

#define PACKET_LENGTH_ERROR 1

VDESTREAM *vdestream_open(void *opaque,
int fdout,
ssize_t (*frecv)(void *opaque, void *buf, size_t count),
void (*ferr)(void *opaque, int type, char *format, ...)
);

ssize_t vdestream_send(VDESTREAM *vdestream, const void *buf, size_t len);

void vdestream_recv(VDESTREAM *vdestream, unsigned char *buf, size_t len);

void vdestream_close(VDESTREAM *vdestream);
```

Figure 4.5: Vdeplug stream encoding functions: libvdeplug.h

4.6 ★`VDEtelweb`

`VDEtelweb` is the telnet and web server for remote configuration of VDE switches. `VDEtelweb` is connected to the management socket of the controlled switch

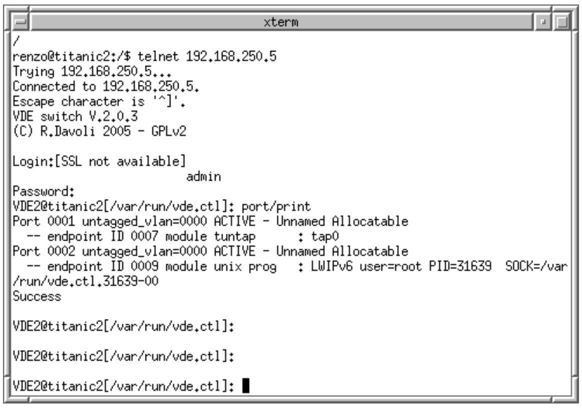

Figure 4.6: `VDEtelweb`: A sample telnet session

(the same socket used by `vdeterm`) as well as to port 0 of the same switch. `VDEtelweb` is an example of an application that uses LWIPv6 as its network stack. With `VDEtelweb` it is possible to configure VDE switches from the virtual network in a manner similar to the operation of professional non-virtual switches. `unixterm` access can be seen as the virtual counterpart of the console access to the switch.

`VDEtelweb` is version independent from VDE switches. Instead of incorporating a set of commands, web pages and fields into `VDEtelweb`, this information is downloaded by `VDEtelweb` from the VDE switch.

Through a `vdetelwebrc` file it is possible to set several options for the interface:

```
# vdetelweb rc sample
ip4=192.168.0.253/24
defroute4=192.168.0.1
password=wvde
```

To support configuration via `VDEtelweb`, the VDE switch must be started with the remote management option (-m):

```
$ vde_switch -m /tmp/vde.mgmt -daemon
```

If the VDE switch was started with remote management enabled, one can simply launch `VDEtelweb` to control the switch either via telnet or with a browser.

```
$ vdetelweb -t -w -f vdetelwebrc /tmp/vde.mgmt
```

4.7 ▲Plugin Support for VDE Switches

VDE switches are implemented using an architecture to support plugins. For example, packet processing like dumping or filtering can be implemented via plugins.

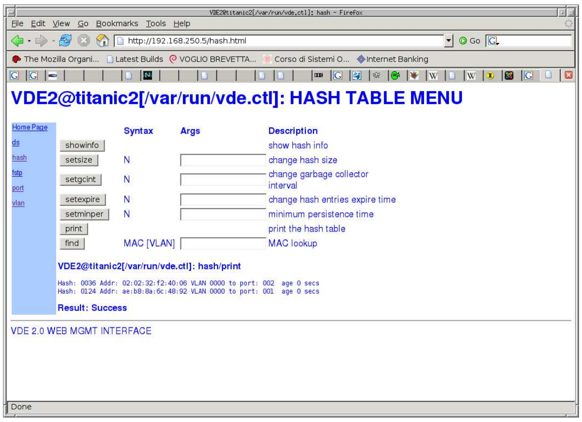

Figure 4.7: `VDEtelweb`: A sample web session

The API to create VDE plugins is described in the file `vdeplugin.h`. See the directory `src/vde_switch/plugins/` in the source code hierarchy for some plugin examples.

A plugin is a dynamic library. Its constructor must initialize plugin's features. A destructor must be provided for cleaning up plugin's data structures. Furthermore, a VDE plugin *must* define a plugin struct variable named `vde_plugin_data`.

```
struct plugin vde_plugin_data={
  .name="test",
  .help="a simple plugin for vde",
};
```

VDE plugins use the event driven paradigm. Functions can be activated by commands or by switch related events.

A plugin can also add its own management commands. All the plugin commands (or menu definition) must be defined in a `comlist` struct array, loaded by the macro `ADDCL` and unloaded by `DELCL`. The example in Fig. 4.8 is a simple (though useless) plugin. It manages an integer value. The user can store a new value or print the current value.

The fields of the `comlist` struct are:

- the command path
- a syntax help (============ for the menu definition)
- a description
- the pointer to the implementation function
- a flag field, implementation functions have different arguments depending on tags.

`NOARG` (or 0),`INTARG` and `STRARG` are used to forward the command parameters to the implementation function. `INTARG` means that there is one integer

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <vdeplugin.h>

int testglobal;

struct plugin vde_plugin_data={
  .name="test",
  .help="a simple plugin for vde",
};

static int testvalue(int val)
{
  testglobal=val;
  return 0;
}

static int testprint(FILE *f)
{
  fprintf(f,"Test Plugin value %d\n",testglobal);
  return 0;
}

static struct comlist cl[]={
  {"test","============","test plugin",NULL,NOARG},
  {"test/change","N","change value",testvalue,INTARG},
  {"test/print","","change value",testprint,WITHFILE},
};

static void
__attribute__ ((constructor))
init (void)
{
  ADDCL(cl);
}

static void
__attribute__ ((destructor))
fini (void)
{
  DELCL(cl);
}
```

Figure 4.8: A minimal VDE plugin

parameter, with `STRARG` all the characters up to the end of line get passed as a string. Splitting up multiple parameters as well as parsing the syntax of the parameters is the responsibility of the implementation function.

The last argument of the implementation function must be an `int` for `INTFUN` and a `char *` for a `STRARG`. If a function needs to return a `long` value then `WITHFILE` must be set. The first argument of the implementation function is a `FILE` * variable in this case. This (virtual) file is used to write the output that is sent at the end of the function using the `0000 DATA END WITH '.'` rule of the management protocol. All the output is buffered and sent when the implementation function exits to avoid interleaving problems with debug outputs. The use of `WITHFD` is rare. When this flag is set the file descriptor of the management session is passed to the implementation function as a parameter (after the `WITHFILE` argument and before the parameter `INTARG` or `STRARG`). This integer file descriptor should be never used to send or receive data (it would cause interleaving problems and protocol misalignments), but it can be used to identify the connection.

The example above can also be compiled as a shared library:

```
gcc -shared -o testplugin.so testplugin.c
```

and loaded into a running VDE switch using the `plugin/add` command::

```
vde$ plugin/add ./testplugin.so
```

In this case the plugin is in the working directory of the switch, otherwise one must specify the complete path or just `testplugin.so` if the library is in a standard directory or in a directory named in `LD_LIBRARY_PATH`.)

```
vde$ plugin/list
0000 DATA END WITH '.'
NAME                  HELP
------------          ----
test                  a simple plugin for vde
.
1000 Success

vde$ help
0000 DATA END WITH '.'
COMMAND PATH      SYNTAX          HELP
------------      --------------  ------------
.....
test              ============    test plugin
test/change       N               change value
test/print                        change value
.
1000 Success
```

At this point the plugin has been loaded and the its commands are available. One may now, for example, enter:

```
vde$ test/print
0000 DATA END WITH '.'
Test Plugin value 0
.
1000 Success

vde$ test/change 42
1000 Success

vde$ test/print
0000 DATA END WITH '.'
Test Plugin value 42
.
1000 Success
```

A plugin can also subscribe to switch event notifications. These functions are used by plugins to subscribe or unsubscribe event notifications:

```
PATH          FLAGS                  VARARG PARAMETERS
-------------------------------------------------------------------------
port/+        D_PORT|D_PLUS          int portno
port/-        D_PORT|D_MINUS         int portno
port/descr    D_PORT|D_DESCR         int portno, int fd, char * descr
port/ep/+     D_EP|D_PLUS            int portno, int fd
port/ep/-     D_EP|D_MINUS           int portno, int fd
packet/in     D_PACKET|D_IN          int portno, char *packet, int len
packet/out    D_PACKET|D_OUT         int portno, char *packet, int len
hash/+        D_HASH|D_PLUS          char *extmac
hash/-        D_HASH|D_MINUS         char *extmac
fstp/status   D_FSTP|D_STATUS        int portno, int vlan, int status
fstp/root     D_FSTP|D_ROOT          int portno, int vlan, char *extmac
fstp/+        D_FSTP|D_PLUS          int portno, int vlan
fstp/-        D_FSTP|D_MINUS         int portno, int vlan
```

Figure 4.9: VDE built-in events

```
int eventadd(int (*fun)(struct dbgcl *event,void *arg,va_list v),
             char *path,void *arg);
int eventdel(int (*fun)(struct dbgcl *event,void *arg,va_list v),
             char *path,void *arg);
```

An event causes function `fun` to be called. The `vararg` parameters depends on the event. VDE switches support the built-in events listed in Fig. 4.9.

The path argument for `eventadd/eventdel` is the kind of event the plugin needs to handle, as listed in the first column of the table above. If the path is just a prefix, all the events matching the prefix gets subscribed. The `arg` parameter is an opaque argument passed to the function (to keep the internal state of the plugin). The `packet/in` and `packet/out` event management functions can drop packets and/or change the packet contents. This is the recommended way to implement packet filtering plugins. For `packet/{in,out}` management functions (the `fun` passed to `eventadd`), the return value is the length of the packet. If the return value is less than or equal to zero, the packet is dropped. It is also possible to rewrite the packet: The buffer is large enough to store MTU bytes long packets.

Plugins can also register their own debug/event items. Each item is described by a `struct dbgcl` element of an array. The method to register or unregister debug/event items is similar to what has been described for commands above. The plugin should define the following fields of `struct dbgcl`:

- path: The path of the event.
- help: The comment line shown by `debug/list`. If help==NULL this is just an event item for other plugins. It cannot be directly used by the management interface (link built in `packet/in`, `packet/out`).
- tag: A numerical tag to speed up the discovery of the event type (to avoid strcmp).

When a plugin needs to send an event notification it uses:

```
EVENTOUT(CL, ...)
```

for debugging output:

```
DBGOUT(CL, FORMAT, ...)
```

where CL is the `dbgcl` struct item. `DBGOUT` has a similar syntax of `fprintf`. While the signature of `EVENTOUT` is open, the sequence of parameters must match those retrieved by the event management function of the client plugin. `EVENTOUT` should never include newline chars (`'\n'`) and should be called only once per notification.

Figure 4.10 code (dump.c) uses a combination of all the support described above. This plugin implements a simple hexadecimal packet dumping.

4.8 ♦vde_switch Internals

libvdeplug and vde_plug Protocols

The protocols used in VDE are very simple and light. The `libvdeplug` protocol is used between VDE client processes, such as between a virtual machine and a switch. This protocol was derived from the one used by the `um_switch`, included in the User-Mode Linux toolset.

The `libvdeplug` protocol uses two PF_UNIX sockets; a control stream and a datagram stream for data. In the initial set-up phase, the client sends a request through the control stream. The fields of this request are defined by the following structure:

```
struct request_v3 {
  uint32_t magic;
  uint32_t version;
  enum request_type type;
  struct sockaddr_un sock;
  char description[MAXDESCR];
};
```

The magic number must be the constant `0xfeedface`, `version` is 3 (for backward compatibility with `um_switch`es), `sock` is the address of the data socket, and `description` is a comment to identify this connection when listing ports on the switch. The low order 8 bits of `type` can be either `REQ_NEW_CONTROL` or `REQ_NEW_PORT0`: `REQ_NEW_CONTROL` to allocate a standard port, and `REQ_NEW_PORT0` to connect to port #0, which is reserved for a management network client (e.g. `vdetelweb`). The upper 24 bits contain the port number. If the value is 0, the client gets connected to the first unused allocatable port, otherwise the switch connects the client to the specified port.

`vde_open` sends the request to the switch on the control stream and receives, from the same stream, the address of the switch data socket. The client data socket is connected to the switch data socket (through the `connect` system call). At this stage the communication channel has been set up. No more data gets sent or received on the control stream, except that "end of stream" is used to test if the switch shuts down the port. Data is send and received on the data socket. Each datagram on this communication is an Ethernet packet, without any encoding or protocol envelope.

`libvdeplug` is also compatible with `kvde`, a new implementation of VDE based on IPN (See Chapter 6) with all packet dispatching occurring at kernel level. In this latter case there is no need for a control connection and the

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <vdeplugin.h>

static int testevent(struct dbgcl *tag,void *arg,va_list v);
static int dump(char *arg);

struct plugin vde_plugin_data={
  .name="dump",
  .help="dump packets",
};

static struct comlist cl[]={
  {"dump","============","DUMP Packets",NULL,NOARG},
  {"dump/active","0/1","start dumping data",dump,STRARG},
};

#define D_DUMP 0100
static struct dbgcl dl[]= {
  {"dump/packetin","dump incoming packet",D_DUMP|D_IN},
  {"dump/packetout","dump outgoing packet",D_DUMP|D_OUT},
};

static int dump(char *arg) {
  int active=atoi(arg);
  int rv;
  if (active)
    rv=eventadd(testevent,"packet",dl);
  else
    rv=eventdel(testevent,"packet",dl);
  return 0;
}

static int testevent(struct dbgcl *event,void *arg,va_list v) {
  struct dbgcl *this=arg;
  switch (event->tag) {
    case D_PACKET|D_OUT:
      this++;
    case D_PACKET|D_IN:
      {
        int port=va_arg(v,int);
        unsigned char *buf=va_arg(v,unsigned char *);
        int len=va_arg(v,int);
        char *pktdump;
        size_t dumplen;
        FILE *out=open_memstream(&pktdump,&dumplen);
        if (out) {
          int i;
          fprintf(out,"Pkt: Port %04d len=%04d ", port, len);
          for (i=0;i<len;i++)
            fprintf(out,"%02x ",buf[i]);
          fclose(out);
          DBGOUT(this, "%s",pktdump);
          free(pktdump);
        }
      }
  }
  return 0;
}

static void __attribute__ ((constructor)) init (void) {
  ADDCL(cl);
  ADDDBGCL(dl);
}

static void __attribute__ ((destructor)) fini (void) {
  DELCL(cl);
  DELDBGCL(dl);
}
```

Figure 4.10: VDE dump.c plugin

data socket is an IPN datagram socket. The protocol policy is defined as `IPN_ANY` so that `libvdeplus` is able to join any kind of service: The standard `IPN_BROADCAST` (hub), `IPN_SWICTH` for `kvde`, or for future services like layer 3 switches.

The command `vde_plug` encodes the Ethernet packets in an ASCII stream so that any application able to provide a bidirectional stdin/stdout stream communication, like `cat`, `netcat` or `ssh`, can be used as a wire between two `vde_plugs`.

Each Ethernet packet has a two bytes header on the ASCII stream. The header contains the length of the packet. In this way the packets can be reread from the ASCII stream. In fact, a `read` operation from the stream receives a number of bytes not necessarily aligned with packet boundaries. In case of errors on the communication channel, `vde_plug` does some basic heuristics to re-synchronize the stream. In this case some packets can get lost, as happens in real Ethernet networks.

vde_switch Source Code

The file `vde_switch.c` implements the call option parsing, starts up all the sub-modules and then enters the main event-processing loop. With regard to the main event-processing loop, each sub-module can register/deregister file types (`add_type, del_type`) defining the module's responsibility for incoming data matching file descriptors for each file type. File types can have high priority (prio=1) or low priority (prio=0). Usually data sockets have high priority, while control and management sockets have low priority. The main event-processing loop has an optimization in the management of the `poll` system call: At each loop iteration, the high priority descriptors pass through a single bubblesort step, migrating the most recently used sockets towards the first elements of the socket array. In this way the most frequently used sockets stay towards the front of the socket array. Furthermore, at each iteration of the main loop, just after the `poll` call, there is a sub-loop to call the handling function for all the file descriptors with pending events. Due to the very bursty nature of Ethernet traffic, this latter, inner loop is likely to terminate after a few iterations.

The only connection between `vde_switch.c` and the other modules is the module startup function `start_modules` which appears at the end of the file.

`consmgmt.c` implements the console, the interface to the management and plugins facilities, the logging facility, and the management of the daemon mode.

`qtimer.c` is a general timer used to schedule delayed actions.

`hash.c` manages the hash table for Ethernet switching. The hashing key is an extended MAC address combining the MAC address and the VLAN number.

`port.c` is the abstraction of an Ethernet port; it uses the hash module functions to switch incoming packets.

`fstp.c` is the implementation of the fast spanning tree protocol.

`datasock.c` code is the interface between the port module and the clients using `libvdeplug` protocol.

`tuntap.c` provides the interface between a VDE switch (port module) and the networking stack of the hosting operating system by tap interfaces.

`packetq.c` implements the queues of unsent packets. A VDE switch tries to deliver each packet several times before to drop it.

Each module uses the function `add_swm` to add its command line options, section of "usage message," constructor, destructor, and input handling function. Each module uses the macros `ADDCL`, and `ADDDBGCL` to define its management commands and its hooks for both debuggers and plugins. The modularity of the switch can be seen also in the output of `vde_switch -h`.

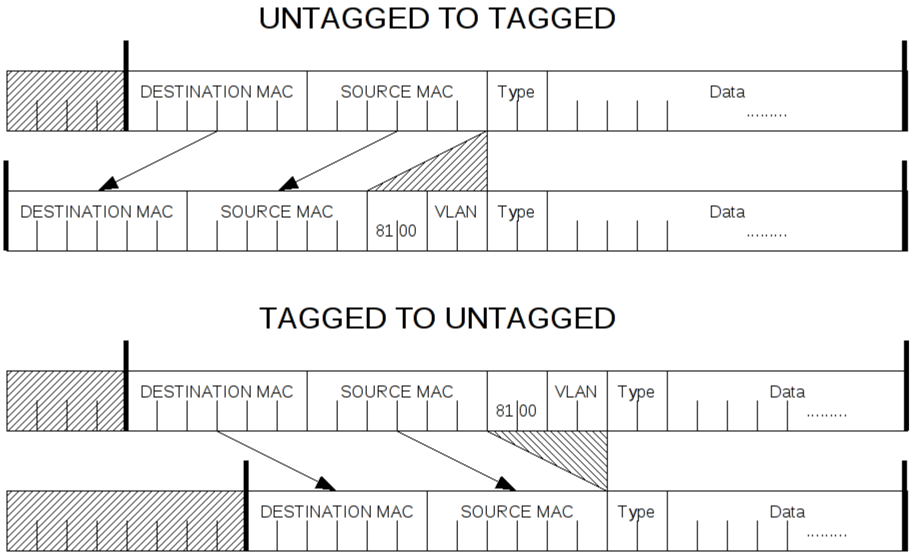

Figure 4.11: Conversions between untagged to untagged packets and viceversa

`port.c` is the core module that supports packet switching. A detail worth some additional description is the management of tagged and untagged packets. Each packet is stored in a structure of type `struct bupacket`. More exactly it is read into that structure with a four bytes offset. See Figure 4.11. This leading unused space is used to convert untagged packets into tagged packets. This shift left of the Ethernet header by four bytes allows for the addition of the extra header data required by IEEE 802.1Q. In the case of converting a tagged packet into an untagged packet, the Ethernet header is right shifted by four bytes, leaving eight leading bytes unused. In this way, either type of conversion only requires the copying of twelve bytes instead of the whole packet.

When broadcasting a packet on a VLAN, the packet gets sent to all the ports. If the original packet was tagged, the packet is first sent out on all the tagged ports, then converted to an untagged packet and then finally sent out to all the untagged ports. A similar operation occurs when broadcasting an untagged packet. This way, at most one conversion is required per broadcast packet.

Each port can have several entry points, which can be used to temporarily set up several cables between two ports of two switches. This facility is used to manage "hand-offs" between two cables. For example if two switches are first connected by a VDE cable using `ssh` on a wired network, it is possible to close the cable and reconnect them using a new VDE cable using a wireless connection. From the point of view of the clients connected to the VDE, all the connections should stay in place, but some time is required to detach the first cable, connect the second, and recompute the switching table. By supporting multiple entry points it is possible to connect the new cable on the same port of the original cable before disconnecting the first cable. In this way the "hand-off" is faster. Note: Some packets may be delivered twice when two (or more) cables interconnect the same pair of ports.

All the VLAN-relevant mapping of ports (belonging, tagged/untagged) as well as port status (not-learning, learning, forwarding) are recorded using bitarray structures. There are four bitarrays (one bit per port) for each VLAN:

- table: If a bit is set it means that the port belongs to the VLAN.
- bctag: The map of forwarding tagged ports.
- bcuntag: The map of forwarding untagged ports.
- notlearning: When set, all packets associated with the fast spanning tree computation, except BPDU-messages, get dropped.

Bitarrays are processed by the macro library in `bitarray.h`.

The status of each port are encoded as follows:

- "not learning" mode means notlearning true, bctag and bcuntag false.
- "learning" mode is when all nonlearning, bctag, bcuntag are false
- "forwarding" mode means nonlearning is false while either bctag or bcuntag is set true.

In some sense notlearning encodes the inability to receive and bctag and bcuntag encode the ability to send.

4.9 ▼VDE in Education

VDE creates a virtual networking infrastructure that can be useful in many educational applications.

A Shooting Range for Networking

For safety, a networking laboratory needs to be a physical space where all the resident computers, and while connected together via switches to form a network, is not connected to any other network. (i.e. The Internet) This way, any experiment conducted, even the most dangerous, (e.g. Denial of Service attacks and defenses) cannot propagate any kind of damage outside of the laboratory. The networking laboratory experience can be replicated in a cheaper and more flexible way by employing VDE: a VDE switch, or set of switches, when not connected to any other network creates a closed network isomorphic to a closed physical networking laboratory.

For example, consider:

```
$ vde_switch -sock /public/net -m 777 -M /public/mgmt.net -daemon
```

This is a switch to which every user can connect his/her virtual machine to. If the host running the switch is a server with 24/7 `ssh` access, students can perform their experiments whenever and wherever. (e.g. 2:00 a.m. from a dormitory room.) Alternatively, instead of running their virtual machines on the server, students can launch their virtual machines on their own machines and use `ssh` as a VDE wire to the server.

A closed network is a wonderful "shooting range" or "sandbox" to test any kind of network service; both benevolent and malevolent. Students may run their sandboxed internet and set up mailing services, DNS (including the defining of new root name servers), web servers, etc. On the other hand they can also try to intrude systems and test if a intrusion detection system can be circumvented. Students can test attacks like man-in-the-middle and arp-spoofing and use tools like OpenVAS[24] and nmap[22] to find vulnerabilities. Students can also try network protocol analyzers like `wireshark` on an experimental network, without any danger of violating the privacy of the other users.

A VDE Based VPN

The only difference between a "shooting range" and a VPN is that with a VPN the VDE switch is connected to the enterprise's production networks.

For example, consider:

```
$ vde_switch -sock /public/vpn -m 777 -tap tap0 -M /public/mgmt.vpn -daemon
```

As discussed in Section 4.4, the `tap0` interface is a virtual interface of the host operating system and must have been authorized by the sysadm in advance.

```
# tunctl -u teacher -t tap0
```

The sysadm configures the forwarding and the packet filtering for the new virtual interface in the same way she would configure a real interface.

This VPN can also be useful for experiments. For example it is possible to be directly connected to an IPV6 network at home (albeit through a virtual connection), even if your Internet provider only supports IPv4.

Some exercises and projects

- Set up and configure a network; the hubs, switches, router (implemented on virtual machines, e.g. by quagga[25]).
- Set up an application level firewall (e.g. with a DMZ, two routers and a bastion host as described in [5]) and test its features.
- Set up a VDE network with cycles and VLANs. Configure the fast spanning tree protocol to use different paths when possible.
- Implement a load balancing algorithm.
- Configure a network monitoring tool like zenoss[4]; test the alarms, and create network faults via `wireshark`.
- Design and implement simple protocols on an Ethernet LAN.
- Configure and test other networking protocols supported by the Linux kernel but rarely used on production networks.
- Design and implement a switch alternative to `vde_switch` having different features.
- Create a *virtual embedded system*: A small application having a small TCP-IP or just UDP-IP embedded. For example students can use uIP[15].

CHAPTER 5

LWIPv6

VDE creates a communication infrastructure for Virtual Machines provided with an Ethernet interface, like System Virtual Machines or other Virtual Machines that boot a virtual kernel and emulate virtual hardware. On the contrary, virtualization as implemented in Process Virtual Machine doesn't have the abstraction of a virtual Ethernet, therefore these VMs cannot use VDE, but they have to use a specific communication library: the Berkeley socket API.

In GNU/Linux, as in many other operating systems, the socket library forwards all the requests to the kernel which implements a correspondent set of system calls. The network stack implementation is part of the kernel, and is shared by all the processes running on that system. In order to use the network with regular user privileges, PVMs need their own network stack. If the PVM is a user-mode virtual machine, then this network stack must be implemented entirely at the user-level.

LWIPv6 is a network stack implemented as a library entirely in user-mode that allows a process or a PVM to directly connect to a virtual network. This implies that, with LWIPv6, every process has its own personal IP address.

LWIPv6 was created as a fork project of LWIP[16, 14, 13]. LWIP is a stack for embedded systems that provides a complete IPv4 protocol stack. LWIPv6 is an IPv6/IPv4 hybrid stack. It has only one packet dispatching engine and only one implementation of UDP and TCP: it is not dual stack. The core part of the stack processes only IPv6 packets; IPv4 addresses are converted into IPv6/IPv4 embedded addresses (chapter 2.5.4 of RFC2373), thus making it more efficient when dealing with IPv6 networks than IPv4 ones. This is by design: as said before, LWIPv6 allows each process to have its own personal IP address; this is more meaningful in the wide address space provided by IPv6 than in the narrow 4 byte wide address space of IPv4.

For example, the IPv4 address 130.136.1.1 (0x82880101) is converted into ::ffff:130.136.1.1 (0x0000 0000 0000 0000 0000 ffff 8288 0101) by the input interface, and a packet delivered to an embedded address is converted back in IPv4 by the output interface.

The network side of LWIPv6 (as opposed to the process side) can be connected to VDE virtual interfaces (`vd0, vd1, ...`) and to tun and tap interfaces (`tn0, tn1, ...` and `tp0, tp1, ...`); a typical loopback interface is also provided (`lo0`). The only way to make processes' IP addresses public on a real network is to connect LWIPv6 (either directly, or through VDE) to a tuntap interface, thus requiring administrator privileges. It is also possible to use slirpvde and obtain process level private IP addressing inside the virtual network and to NAT towards the real network.

LWIPv6 supports also slirp interfaces. A slirp interface (`sl0, sl1, ...`) converts TCP/UDP traffic routed to it into requests to the hosting system TCP-IP stack by the process running LWIPv6.

5.1 LWIPv6 API

The interface of LWIPv6 can be seen in Fig 5.1 and 5.2. These function are generally called by the program prior to start its operations. It is possible to set up several stacks and for each stack the interfaces, to manage IP addresses and the routing table.

The second part of the interface is the complete set of Berkeley socket calls and the other system calls that can be used to communicate with sockets. They are named after the corresponding calls provided by the standard library, with a `lwip_` prefix and have exactly the same syntax.

`lwip_{select,poll,pselect,ppoll}` are able to manage file descriptors of LWIPv6 sockets and descriptors of other files, devices and sockets at the same time.

LWIPv6 currently supports the following protocol families:

- `PF_INET, PF_INET6`: for IPv4 and IPv6 connectivity;
- `PF_PACKET`: for direct access to the Data-Link layer;
- `PF_NETLINK`: for interface and routing configuration.

LWIPv6 supports IPv6 autoconfiguration (RFC2462) and an internal DHCP client has been recently added in the library. The latest development of the project also include a support for packet filtering (similar to iptables) including a Network Address Translation feature for both IPv4 and IPv6.

5.2 An LWIPv6 tutorial

This is a short guide of the LWIPv6 library. It is intended for programmers wishing to write programs using LWIPv6.

LWIPv6 implements an entire LWIPv4/v6 stack as a library, thus when a program uses LWIPv6 it can interoperate using its own TCP-IP stack (or even multiple LWIPV6 stacks, the library supports many stacks at the same time).

LWIPv6 stacks communicate using four different types of interfaces:

- `tap` (access to /dev/net/tun required) it uses a point to point layer 2 (ethernet) virtual interface with the hosting machine;

Constructor/destructor: *do not call these functions unless you are writing a statically linked program*

```
void lwip_init(void);
void lwip_fini(void);
```

Define a new stack, terminate an existing stack, set stack flags:

```
struct stack *lwip_stack_new(void);
void lwip_stack_free(struct stack *stack);
#define LWIP_STACK_FLAG_FORWARDING 1
unsigned long lwip_stack_flags_get(struct stack *stackid);
void lwip_stack_flags_set(struct stack *stackid, unsigned long flags);
```

Set/Get the current default stack (for `lwip_socket\verb`).

```
struct stack *lwip_stack_get(void);
void lwip_stack_set(struct stack *stack);
```

Define new interfaces:

```
struct netif *lwip_vdeif_add(struct stack *stack, void *arg);
struct netif *lwip_tapif_add(struct stack *stack, void *arg);
struct netif *lwip_tunif_add(struct stack *stack, void *arg);
struct netif *lwip_slirpif_add(struct stack *stack, void *arg);
```

Add/delete addresses:

```
int lwip_add_addr(struct netif *netif,struct ip_addr *ipaddr, struct ip_addr *netmask);
int lwip_del_addr(struct netif *netif,struct ip_addr *ipaddr, struct ip_addr *netmask);
```

Add/delete routes:

```
int lwip_add_route(struct stack *stack, struct ip_addr *addr, struct ip_addr *netmask,
             struct ip_addr *nexthop, struct netif *netif, int flags);
int lwip_del_route(struct stack *stack, struct ip_addr *addr, struct ip_addr *netmask,
             struct ip_addr *nexthop, struct netif *netif, int flags);
```

Turn the interface up/down:

```
int lwip_ifup(struct netif *netif);
int lwip_ifdown(struct netif *netif);
```

Figure 5.1: LWIPv6 API: Interface definition

- `tun` (access to /dev/net/tun required) similar fo the previous one, it uses a point to point layer 3 (IP) virtual connection;
- `vde` it gets connected to a Virtual Distributed Ethernet switch.
- `slirp` it is a virtual interface which uses the TCP-IP stack of the hosting system.

Loading and Linking LWIPV6

A program can use LWIPv6 in three different ways.

- By linking statically the library.

  ```
  gcc -o static static.c /usr/local/lib/liblwipv6.a  -lpthread -ldl
  ```

  in this case the constructor/destructor must be explicitely called in the code:

LWIPv6 implementation of comm syscalls:

```
int lwip_msocket(struct stack *stack, int domain, int type, int protocol);
int lwip_socket(int domain, int type, int protocol);
int lwip_bind(int s, struct sockaddr *name, socklen_t namelen);
int lwip_connect(int s, struct sockaddr *name, socklen_t namelen);
int lwip_listen(int s, int backlog);
int lwip_accept(int s, struct sockaddr *addr, socklen_t *addrlen);
int lwip_getsockname (int s, struct sockaddr *name, socklen_t *namelen);
int lwip_getpeername (int s, struct sockaddr *name, socklen_t *namelen);
int lwip_send(int s, void *dataptr, int size, unsigned int flags);
int lwip_recv(int s, void *mem, int len, unsigned int flags);
int lwip_sendto(int s, void *dataptr, int size, unsigned int flags,
        struct sockaddr *to, socklen_t tolen);
int lwip_recvfrom(int s, void *mem, int len, unsigned int flags,
        struct sockaddr *from, socklen_t *fromlen);
int lwip_shutdown(int s, int how);
int lwip_setsockopt (int s, int level, int optname, const void *optval, socklen_t optlen);
int lwip_getsockopt (int s, int level, int optname, void *optval, socklen_t *optlen);
int lwip_sendmsg(int fd, const struct msghdr *msg, int flags);
int lwip_recvmsg(int fd, struct msghdr *msg, int flags);
int lwip_write(int s, void *dataptr, int size);
int lwip_read(int s, void *mem, int len);
int lwip_writev(int s, struct iovec *vector, int count);
int lwip_readv(int s, struct iovec *vector, int count);
int lwip_ioctl(int s, long cmd, void *argp);
int lwip_close(int s);
int lwip_select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
        struct timeval *timeout);
int lwip_pselect(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset,
        const struct timespec *timeout, const sigset_t *sigmask);
int lwip_poll(struct pollfd *fds, nfds_t nfds, int timeout);
int lwip_ppoll(struct pollfd *fds, nfds_t nfds,
        const struct timespec *timeout, const sigset_t *sigmask);
```

Management of asynchronous events:

```
int lwip_event_subscribe(lwipvoidfun cb, void *arg, int fd, int how);
```

Figure 5.2: LWIPv6 API: socket library and I/O

```
main(int argc,char *argv[])
{
    lwip_init();
    /* core of the application */
    lwip_fini();
}
```

- Using a dynamic linking.

  ```
  gcc -o dynamic dynamic.c -llwipv6 -lpthread -ldl
  ```

  `lwip_init`, `lwip_fini` are automagically called when the library is loaded. Do not call them in the code.

- Dynamically loading the dynamic library. The code appears like this:

  ```
  void *handle
  ...
  handle=loadlwipv6dl();
  ....
  /* handle==NULL in case of errors; to unload the library use: dlclose(handle) */
  ```

This application should be compiled in this way:

```
gcc -o dynload dynload.c -lpthread -ldl
```

The advantage of this approach is the lack of direct dependence (requirement) for the lwipv6 library. It is possible to write programs able to run both on systems where lwipv6 is installed and on system where lwipv6 does not exist. The choice of features can be done at run time.

How to start a stack (or several stacks)

A stack descriptor is defined as a (opaque) stucture:

```
struct stack *stackd;
```

The program can start a stack by calling:

```
stackd=lwip_stack_new();
```

If something goes wrong, `lwip_stack_new` returns *NULL*. A program can call `lwip_stack_new` several times to define several TCP-IP stacks.

It is also possible to shut down a stack in this way:

```
lwip_stack_free(stackd);
```

An LWIPv6 stack does not route packets between different interfaces in its default configuration. The forwarding of IP packets can be enabled when required in this way:

```
lwip_stack_flags_set(stackd,LWIP_STACK_FLAG_FORWARDING);
```

How to use a Hybrid Stack

LWIPv6 is a Hybrid stack. In a raw and intuitive definition, it means that it has only one packet engine (lwipv6) and it is backward compatible with IPv4 using some exceptions in the code where the management is different.

LWIPv6 internal engine uses exclusively IPv6 addresses. All the calls to set up the addresses and routes use addresses defined as:

```
struct ip_addr {
    uint32_t addr[4];
};
```

This data structure contains an IPv6 address. IPv4 address are stored as IPv4 mapped address, i.e. in the following form: the first 80 bits set to zero, the next 16 set to one, while the last 32 bits are the IPv4 address.

There are macro in the lwipv6 include file to help programmers to define IPv4 and IPv6 addresses and masks.

```
IP6_ADDR(addr,0x2001,0x760,0x0,0x0,0x0,0x0,0x0,0x1)
```

defines addr as 2001:760::1. IP6ADDR can be used both for address and masks, e.g.

```
IP6_ADDR(mask,0xffff,0xffff,0xffff,0xffff,0x0,0x0,0x0,0x0)
```

is a /64 mask.

For IPV4 there are two different macros:

```
IP64_ADDR(addr4,192,168,1,1);
IP64_MASKADDR(mask4,255,255,255,0);
```

define addr4 e mask4 the IPv4 mapped adress 192.168.1.1 and a /24 mask (255.255.255.0) respectively.

How to define interfaces, addresses, routes

Once a stack has been created, it is useless until it has a non trivial interface. The loopback `lo0` interface is the only one automatically defined in a new stack.

```
struct netif *lwip_vdeif_add(struct stack *stack, void *arg);
struct netif *lwip_tapif_add(struct stack *stack, void *arg);
struct netif *lwip_tunif_add(struct stack *stack, void *arg);
struct netif *lwip_slirpif_add(struct stack *stack, void *arg);
```

The three functions above define new interfaces. For tun and tap interfaces, the argument is a string that will be used as the name of the virtual interface. `lwip_vdeif_add` argument is the path of the `vde_switch`.

```
struct netif *tunnif,*vdenif,*vde2nif;
tunnif=lwip_tunif_add(stackd,"tun4");
vdenif=lwip_vdeif_add(stackd,"/var/run/vde.ctl");
vde2nif=lwip_vdeif_add(stackd,"/var/run/vde.ctl[4]");
```

In this example three interfaces get added to the stack defined by *stackd*. The first is the tun interface named *tun4*, the second a vde connection to a switch, the third another connection to the port #4 to the same switch. In fact the square brackets syntax is commonly used in vde to indicate a specfic port of a switch.

Interfaces (except slirp ones) must be assigned TCP-IP addresses to communicate.

```
int lwip_add_addr(struct netif *netif,struct ip_addr *ipaddr,
         struct ip_addr *netmask);
int lwip_del_addr(struct netif *netif,struct ip_addr *ipaddr,
         struct ip_addr *netmask);
```

for example the following chunk of code sets the address 192.168.1.1/24 for vdenif.

```
struct ip_addr addr4, mask4;
IP64_ADDR(&addr4,192,168,1,1);
IP64_MASKADDR(&mask4,255,255,255,0);
lwip_add_addr(vdenif,&addr4,&mask4);
```

An interface can have several IPv4 and IPv6 addresses. IPv6 supports stateless address autoconfiguration.

In a similar manner it is possible to define routes.

```
int lwip_add_route(struct stack *stack, struct ip_addr *addr,
         struct ip_addr *netmask, struct ip_addr *nexthop,
         struct netif *netif, int flags);
int lwip_del_route(struct stack *stack, struct ip_addr *addr,
         struct ip_addr *netmask, struct ip_addr *nexthop,
         struct netif *netif, int flags);
```

`addr/netmask` is the destination address for the route. `nexthop` is the next hop destination address and `netif` is the network interface where the packet must be dispatched. To define a default route, use `IPADDR_ANY` both for address and for netmask.

e.g.

```
struct ip_addr gwaddr4;
IP64_ADDR(&gwaddr4,192,168,1,254);
lwip_add_route(stackd, IPADDR_ANY, IPADDR_ANY, &gwaddr4, vdenif, 0);
```

defines the default route to be 192.168.1.254 on interface vdenif.

Remember to turn on the interfaces!

All the interfaces added by `lwip_vdeif_add` `lwip_tunif_add` or `lwip_tapif_add` are disabled upon creation. `lwip_ifup` turns on an interface, `lwip_ifdown` turns it off. e.g.

```
lwip_ifup(vdeif);
```

Remember to turn on the interfaces otherwise the stack won't work!

How to use a stack (or several stacks)

`lwip_msocket` is similar to the msocket call defined by the multiple stack exstension of the Berkeley socket API definition (msockets). The sole difference between the signature of msocket and `lwip_socket` is that the socket decriptor gets used instead of the pathname of the stack special file.

```
int lwip_msocket(struct stack *stack, int domain, int type,
         int protocol);
```

For example, a TCP (V4) socket on the lwip stack `stackd` gets created by the following call.

```
fd=lwip_msocket(stackd, AF_INET, SOCK_STREAM, 0);
```

fd can be used in Berkeley Sockets API like calls: `lwip_bind`, `lwip_connect`, `lwip_accept`, `lwip_recv`, `lwip_send` ... that correspond to bind, accept, recv, send, etc.

"sockaddr" parameters (like in bind, connect, etc) use the standard definitions (`sockaddr_in`, `sockaddr_in6`).

For application using only one stack (or at least one stack at a time) it is possible to define the default stack:

```
lwip_stack_set(stackd);
```

If the default stack has been already defined the call

```
lwip_socket(AF_INET, SOCK_STREAM, 0);
```

implicitely refers to `stackd` The default stack gets defined for the whole library, thus the use of default networks is discouraged on multithreaded applications working on several stack concurrently.

A complete example

The following code is a simple TCP terminal emulator working on LWIPv6. It works like the utility `nc` used as a TCP client. In fact our utility (say it is named lwipnc):

```
lwipnc 192.168.250.1 9999
```

has the same behavior of netcat:

```
nc 192.168.250.1 9999
```

One way to test this program is by starting a tcp server on the other end of the network link:

```
nc -l -p 9999
```

The source code of lwipnc.c is in Figure 5.3.

Compile it using lwipv6 as a dynamic library in this way:

```
gcc -o lwipnc lwipnc.c -ldl -lpthread -llwipv6
```

or as a run-time dinamically loaded library in this way:

```
gcc -o lwipnc lwipnc.c -D LWIPV6DL -ldl -lpthread
```

It is possible to run the same example on a tun or on a tap interface just by changing the source code line:

```
if((nif=lwip_vdeif_add(stack,"/var/run/vde.ctl"))==NULL){
```

into

```
if((nif=lwip_tunif_add(stack,"tun1"))==NULL){
```

or

```
if((nif=lwip_tapif_add(stack,"tap1"))==NULL){
```

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <lwipv6.h>
#include <poll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define BUFSIZE 1024
char buf[BUFSIZE];

int main(int argc,char *argv[])
{
  struct sockaddr_in serv_addr;
  int fd;
  void *handle;
  struct stack *stack;
  struct netif *nif;
  struct ip_addr addr;
  struct ip_addr mask;
#ifdef LWIPV6DL
  if ((handle=loadlwipv6dl()) == NULL) { /* Run-time load the library (if requested) */
    perror("LWIP lib not loaded"); exit(-1);
  }
#endif
  if((stack=lwip_stack_new())==NULL){ /* define a new stack */
    perror("Lwipstack not created"); exit(-1);
  }
  if((nif=lwip_vdeif_add(stack,"/var/run/vde.ctl"))==NULL){ /* add an interface */
    perror("Interface not loaded"); exit(-1);
  }
  IP64_ADDR(&addr,192,168,250,20); /* set the local IP address of the interface */
  IP64_MASKADDR(&mask,255,255,255,0);
  lwip_add_addr(nif,&addr,&mask);
  lwip_ifup(nif); /* turn on the interface */
  memset((char *) &serv_addr,0,sizeof(serv_addr));
  serv_addr.sin_family      = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
  serv_addr.sin_port        = htons(atoi(argv[2]));
  if((fd=lwip_msocket(stack,PF_INET,SOCK_STREAM,0))<0) { /* create a TCP lwipv6 socket */
    perror("Socket opening error"); exit(-1);
  }
  /* connect it to the address specified as argv[1] port argv[2] */
  if (lwip_connect(fd,(struct sockaddr *)(&serv_addr),sizeof(serv_addr)) < 0) {
    perror("Socket connecting error"); exit(-1);
  }
  while(1) {
    struct pollfd pfd[]={{STDIN_FILENO,POLLIN,0},{fd,POLLIN,0}};
    int n;
    /* wait for input both from stdin and from the socket */
    lwip_poll(pfd,2,-1);
    if(pfd[1].revents & POLLIN) { /* copy data from the socket to stdout */
      if((n=lwip_read(fd,buf,BUFSIZE)) == 0)
        exit(0);
      write(STDOUT_FILENO,buf,n);
    }
    if(pfd[0].revents & POLLIN) { /* copy data from stdin to the socket */
      if((n=read(STDIN_FILENO,buf,BUFSIZE)) == 0)
        exit(0);
      lwip_write(fd,buf,n);
    }
  }
}
```

Figure 5.3: The source code of `lwipnc.c`

Another example: a tiny router

The entire code of a router running a vde, a tap and a slirp interface is in Figure 5.4. The code creates the stack, adds the interfaces, defines addresses and routes, activates the interfaces and enters a pausing loop.

In fact, when the `LWIP_STACK_FLAG_FORWARDING` flag is set, all the packet forwarding is managed by the stack itself.

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <lwipv6.h>

int main(int argc,char *argv[])
{
  struct stack *stack;
  struct netif *vdeif, *slirpif, *tapif;
  struct ip_addr addr, mask, routeaddr, routemask;
  if((stack=lwip_stack_new())==NULL){ /* define a new stack */
    perror("Lwipstack not created");
    exit(-1);
  }
  lwip_stack_flags_set(stack,LWIP_STACK_FLAG_FORWARDING);
  if((vdeif=lwip_vdeif_add(stack,"/tmp/vde1"))==NULL){ /* add a vde interface */
    perror("VDE Interface not loaded");
    exit(-1);
  }
  if((tapif=lwip_tapif_add(stack,"tpx"))==NULL){ /* add a tap interface */
    perror("TAP Interface not loaded");
    exit(-1);
  }
  if((slirpif=lwip_slirpif_add(stack,NULL))==NULL){ /* add a slirp interface */
    perror("SLIRP Interface not loaded");
    exit(-1);
  }
  IP64_MASKADDR(&mask,255,255,255,0);
  IP64_ADDR(&addr,10,0,2,1); /* set the local IP address of the vde interface */
  lwip_add_addr(vdeif,&addr,&mask);
  IP64_ADDR(&addr,10,0,3,1); /* set the local IP address of the tap interface */
  lwip_add_addr(tapif,&addr,&mask);
  IP64_ADDR(&routeaddr,0,0,0,0); /* add a default route to slirp */
  IP64_MASKADDR(&routemask,0,0,0,0);
  if(lwip_add_route(stack,&routeaddr,&routemask,&addr,slirpif,0)<0){
    perror("lwip_add_route err");
    exit(-1);
  }
  lwip_ifup(vdeif);
  lwip_ifup(tapif);
  lwip_ifup(slirpif);
  while(1)
    pause();
}
```

Figure 5.4: The source code of `tinyrouter.c`

A different model for asynchrony: `event_subscribe`

LWIPv6 provides `lwip_select`, `lwip_pselect`, `lwip_poll`, `lwip_ppoll` having the same semantics of the correspondent system call (those without the prefix `lwip_`). These calls are useful when porting applications using the standard Berkeley socket API to LWIPv6.

There is however in LWIPv6 another way to deal with asynchronous events generated by the stack:

```
typedef void (*lwipvoidfun)();
int lwip_event_subscribe(void (*cb)(void *), void *arg, int fd,
          int how);
```

`cb` is the address of a callback function (or NULL), `arg` is the argument that will be passed to the callback function, `fd` is a LWIPv6 file descriptor, `how` is an event code. `how` gets the same encoding of events as in *poll(2)*. The return value is a bitmask filled in with the event that actually occured. (The return value always reports a subset of events with respect to those encoded in `how`. This function has three different meanings:

- If *cb==NULL* and *arg==NULL*, it tests which events(s) already happened. e.g.

  ```
  rv=lwip_event_subscribe(NULL,NULL,fd,POLLIN);
  ```

  rv is non-zero if there is data to read.

- if *cb!=NULL* LWIPv6 tests which events(s) among those defined in `how` already happened. If *rv==0*, i.e. no one of the event happened, it subscribes for a notification. When an event of `how` happens LWIPv6 calls *cb(arg)*.

- If *cb==NULL*, `lwip_event_subscribe` checks again to see which event(s) happened. If there is a pending notification request with the same arg, it is cancelled.

5.3 ♦LWIPv6 Internals

Stacks architecture and software layers

LwIPv6 is an IPv4/IPv6 hybrid stack and its architecture is based on the logical model called one process for message. In this model all the operations are performed by single thread and network protocols are rappresented by a set of API used during I/O operations.

On LwIPv6, Network Layer protocols (e.g.: IP, ICMP) and Transport Layer protocols (TCP,UDP) are handled by a single main thread which is separated by the application thread.

The stack sends and receive data throw several virtual network interfaces. Each network device has got its driver and its execution thread: the first one implements I/O functions and takes care about the phisical layer and the Datalink layer; the interfaces thread must use drivers functions to read incoming data from the virtual network.

All stacks threads (main thread, interfaces threads) and the applications thread comunicate with each other by using Message-Passing APIs, Semaphores and Call-back functions. To make the interaction between the application and the Stack easier, LwIPv6 comes with two different Application Level API: the Netconn library and an implementation of the BSD Sockets Library.

In figure 5.5 you can see the LwIPv6s global architecture and the several stacks layers and exectucion threads.

The abstraction layer

In order to make LwIPv6 portable, the specific function calls and data structures provieded by the operating system are not used directly in the code. Instead, when such functions are needed the operating system emulation layer is used. The operating system emulation layer provides a uniform interface to operating system services such as timers, process synchronization, and message passing mechanisms. In principle, when porting LwIPv6 to other operating systems only an implementation of the operating system emulation layer for that particular operating system is needed.

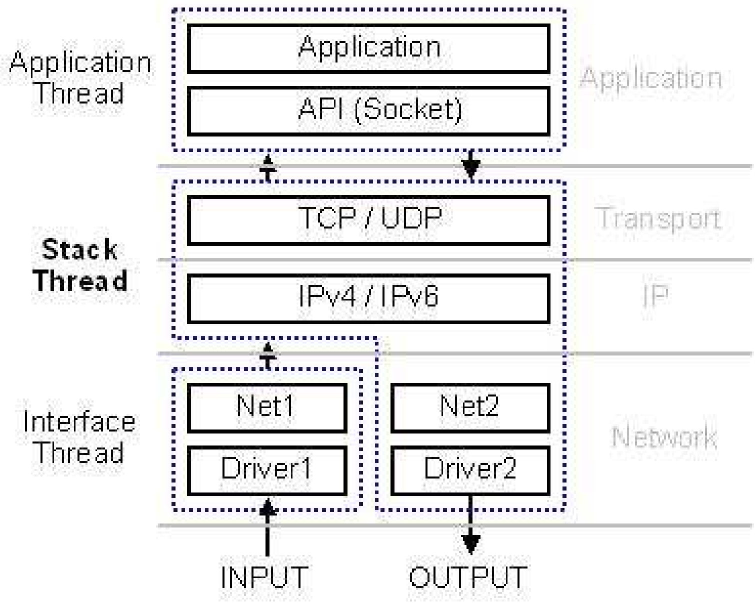

Figure 5.5: LWIP architecture

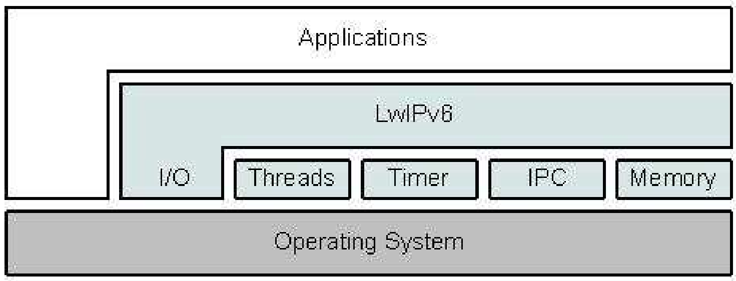

Figure 5.6: LWIP and the other components of an operating system

The Operating Systems memory management is masked too by using very few API. The only operating systems functions used directly without a wrapping API is rappresented by the I/O mechanisms.

On a unix-like operating system, this abstraction layer could be implemented by using the standard Posix API for thread and syncronization primitives and the standard C library for memory management.

I/O

LwIPv6 uses the operating systems I/O primitives only inside virtual network device drivers. Inside the driver code the stack could launch operating systems specific functions like open(), send(), recv(), ecc...

There are few LwIPv6 features that use particular system-calls in other points of the stacks code, like the support for the UMView `select()` mechanism, but these are very particular cases.

Multi-threading

The abstraction layer provides only one functions for creating new threads:

```
sys_thread_t sys_thread_new(void (* thread)(void *arg),
                            void *arg, int prio);
```

The functions sets up a new execution thread and launchs the function thread with arg as input parameter. If the function terminates successfully, it returns a new thread descriptor. This functions MUST be used only inside the stack code. LwIPv6 doesnt provide any other functions for thread handling and nobody can stop or kill the new thread.

Semaphores

Semaphores are used inside LwIPv6 for thread syncronization. Each semaphore is identified by a sys_sem_t descriptor. To create a new semaphore call this function:

```
sys_sem_t sys_sem_new(u8_t count);
```

Only two operations are allowed on a semaphore: Signal (V) e Wait (P) and are performed by calling these API:

```
void sys_sem_signal(sys_sem_t sem);
void sys_sem_wait(sys_sem_t sem);
int sys_sem_wait_timeout(sys_sem_t sem, u32_t timeout);
```

The function `sys_sem_wait_timeout()` blocks the calling thread on the semaphore until a signal occurs or the timeout expires.

Message-passing

The comunication between threads is implemented by using a simple message-passing mechanism based on message queues or mail boxes. Mailboxes allow only two basic operations: the insertion (Post) and the removal (Fetch) of messages into and from the mailbox (Post). A new mailbox can be created with the following function:

```
sys_mbox_t sys_mbox_new(void);
```

Messages exchanged by threads are basically memory pointers. Communication is performed by using the following I/O functions:

```
void sys_mbox_post(sys_mbox_t mbox, void *msg);
void sys_mbox_fetch(sys_mbox_t mbox, void **msg);
```

If a thread attempts to read from an empty mailbox with `sys_mbox_fetch()`, it will block until an other thread pushes at least one new message inside the box.

Timers

A timer is a sequence of instructions (a function) executed only one time when a timeout expires. Each thread has its own timers and there is no limit to number of timers anybody can register for each thread. Well, this is not really true because there is limit to the number of timer descriptors the user can allocate in memory.

Different threads can not access to the timers of the others threads. When a thread sets up a new timer, a new timer descriptor is stored inside a list of pending timers. The elements of this list are declared as follows:

```
struct sys_timeout {
  struct sys_timeout *next;
  u32_t time;
  sys_timeout_handler h;
  void *arg;
};
```

After `time` milliseconds, the function `h` is launched with input parameter `arg`. All the timers of the same thread are stored respecting the expiring time. The functions for creating or removing timers are declared as follows:

```
void sys_timeout(u32_t msecs,sys_timeout_handler h,
                 void *arg);

void sys_untimeout(sys_timeout_handler h, void *arg);
```

A new timer is identified by both the functions `h` and the argument `arg`. If a thread calls `sys_untimeout()` on a timer created by an other thread, the call fails and returns immediately.

This peace of code shows how to set up an auto-respawing update timer:

```
#define TIMEOUT 1000

/* This is the timeout handler */
void tcp_tmr(void *arg)
{
    char *data = (char *) arg;

    ...call your update function...

    /* set up the next timer */
    sys_timeout(TIMEOUT, tcp_tmr, arg);
}

char *dummydata = ...;

int main(int argc, char* argv[])
{
    ...
    /* Set up the timer */
```

```
    sys_timeout(TIMEOUT, tcp_tmr, dummydata);
    ...
}
```

Problems with timers Timers handling is not performed in a separeted thread and it's triggered only inside the Abstraction Layer's API. What does this means? This means that a pending timer will expire only if any semaphore o message-passing functions is called after the timer's setup procedure.

For example, if you set up a new timer, the stack will check for its execution only at the first `sys_*()` function call.

This is a very important point because this influences also the real execution time of a timer function. If you set up a 10 seconds timeout at time T1 and, for any reason, you execute a `sys_*()` function after 60 seconds, your timeout function handler will be called only after those 60 seconds, regardless of the original timeout.

Memory management

LwIPv6 provides a set of API for the dynamic memory management:

```
void *mem_malloc(mem_size_t size);
void mem_free(void *mem);
void *mem_realloc(void *mem, mem_size_t size);
void *mem_reallocm(void *mem, mem_size_t size);
```

Input parameters are different but the semantic and the return values are the same of `malloc()`, `free()`, `realloc()` functions. LwIPv6 comes with two different implementations: a wrapper for the standard C library functions and a dynamic memory manager which uses an hidden static RAM buffer. Under unix-like systems the first one implementation is preferred. The second one should be used only on those embedded systems coming without a dynamic memory manager.

These function are thread safe under unix-like system and when the standard C library wrapper is used.

Main data structures

The two main data structures used inside LwIPv6 are: IP Addresses and Packet buffers (sent or received). Its very importat how they are manipulated and stored in memory.

IP Addresses

LwIPv6 can handle both IPv4 and IPv6 packets, but internally, every data structure stores IP addresses in the IPv6 (128 bit) format: IPv4 addresses are converted in the IPv4-Mapped IPv6 format; IPv6 are stored unchanged. Network netmasks are converted in the 128 bit format too, but the first 80 bit are set to 1. For example, the netmask `255.255.255.0` (`0xffffff00`), is converted in the following 128 bit netmask:

```
0xfffffff.ffffffff.ffffffff.ffffFFFF.ffffff00
```

LwIPv6 stored IPv4 and IPv6 addresses inside these two structures:

```
struct ip4_addr {
  u32_t addr;
};

struct ip_addr {
    u32_t addr[4];
};
```

The convertion from 128 bit back to the 32 bit rappresentation occurs only in few point for the stacks code. For example inside the ARP protocol code and in some functions of the Socket API where IPv4 addresses are needed (e.g.: `getpeename()`).

Packet Buffers

IP packets are rappresented inside LwIPv6 by using special data structures called PBuf (Packet Buffer). This data type is very similar to those data structures used inside other operating systems like the Mbuf structure (BSD systems) or Skbuff structures (GNU/Linux). The PBuf structure is defined in this way:

```
struct pbuf {
    struct pbuf *next;
    void *payload;
    u16_t tot_len;
    u16_t len;
    u16_t flags;
    u16_t ref;
};
```

The field `payload` points to the buffer of length `len` where data is stored. An IP packet can be splitted in several no-contiguous memory buffers linked together as a simple list by using the field `next`.

This special linked list is called Pbuf Chain and the total amount of used memory is saved inside the field `tot_len`. If the chain contains only one element, `tot_len` e `len` store the same value. The field `ref` specifies the number of active references (memory pointers) pending on the the packet.

There exist four different types of Pbuf structures: `PBUF_RAM`, `PBUF_ROM`, `PBUF_POOL` and `PBUF_REF`. The first three are used to access to different types of memories (RAM, ROM or a to statically allocated buffer). The `PBUF_REF` type is used to mantain a reference to a memory buffer not handled by the stack's memory sub-system (e.g: the thread's stack memory).

In figure 5.7 you can see a IP packet stored inside a Pbuf Chain composed by several type of Pbuf element.

To allocate a new Pbuf structure you must call the following function:

```
struct pbuf *pbuf_alloc(pbuf_layer layer, u16_t size,
                        pbuf_flag flag);
```

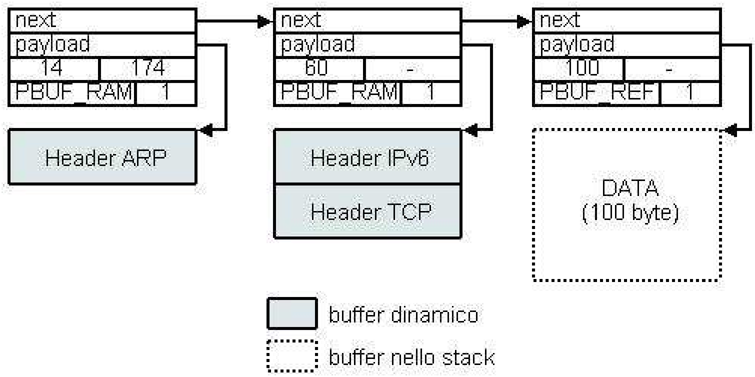

Figure 5.7: Pbuf chain

The parameters `size` and `flag` specify the dimension in bytes and the type of new Pbuf to create..

The `layer` parameter The `layer` parameter specifies which kind of network headers will be encapsulated inside the new buffer. There are four level: `PBUF_TRANSPORT`, `PBUF_IP`, `PBUF_LINK` and `PBUF_RAW`. The `PBUF_TRANSPORT`, for example, is used to allocate enough space for the data payload plus the link layer's header (Ethernet) plus the network packet's header (Ipv4 or Ipv6) plus the transport packet's header (TCP or UDP). This parameter is very important because everytime new data have to be sent, the stack performs the protocol encapsulation process. For each step of the encapsulation new space for an other protocol packet header have to be allocated. These buffers can be allocated by using several Pbuf packets, one for each new header, but this solution is not optimal and it can cause memory fragmentation.

With this special parameter, each new packet can be stored inside a single Pbuf element instead of using a Pbuf chain. The following function can be used to shift the `payload` pointer and thereby to access to the memory locations reserved to each network header:

```
u8_t pbuf_header(struct pbuf *p, s16_t header_size)
```

In Figurelwippbufheader you can see a `PBUF_TRANSPORT` Pbuf structure and the consecutive calls to `pbuf_header()` (from top to bottom) needed to access to the different segments of the packet.

Drivers and Network Interfaces

LwIPv6 can handle an unbouned number of network interfaces at the same time. Each network device is rappresented by a special structure called `netif`:

```
struct netif {
  struct netif *next;

  char name[2];
```

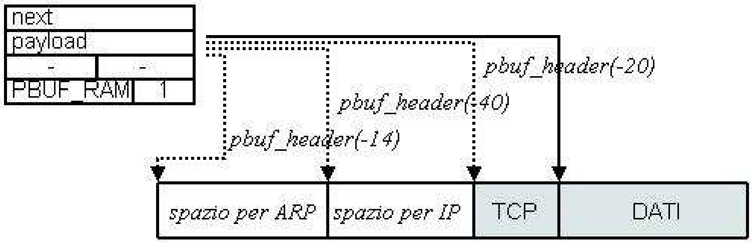

Figure 5.8: Pbuf header segments

```
  u8_t num;
  u8_t id;

  unsigned char hwaddr_len;
  unsigned char hwaddr[NETIF_MAX_HWADDR_LEN];
  u16_t mtu;
  u8_t link_type;
  u16_t flags;
  void *state;

  struct ip_addr_list *addrs;

  err_t (* input)     (struct pbuf *p, struct netif *inp);
  err_t (* output)    (struct netif *netif, struct pbuf *p,
                       struct ip_addr *ipaddr);
  err_t (* linkoutput)(struct netif *netif, struct pbuf *p);
  err_t (* cleanup)   (struct netif *netif);
  void  (* change)    (struct netif *netif, u32_t type);
};
```

The network driver must initialize all the structure and must launch the interfaces thread. All the interfaces created by the stack are linked together in a simple list structure by using the next field. Each interface is identified either by its logical name, composed by the fields `name` and `num` (eg. et0, wl0, bt2) or its `id`, which is an unique interger number assigned by the stack at initialization time.

The `netif` structure stores several informations like the network link type (`link_type`), the supported MTU (`mtu`), the physical address for the device (`hwaddr`) if supported and a set of flags (`flag`) used to save the current state of the interface (`UP`, `DOWN`, `PROMISQUOSE MODE`, ecc...). The field state is used by the device driver to save private data useful for the driver only.

Interfaces addresses Each interface can use several IPv4 and IPv6 addresses, and they are stored inside the list addrs. Each entry of the list contains the IP address, the netmask and some additional flags used mainly by the IPv6 layer. The entry structure `ip_addr_list` is defined as follow:

```
struct ip_addr_list {
```

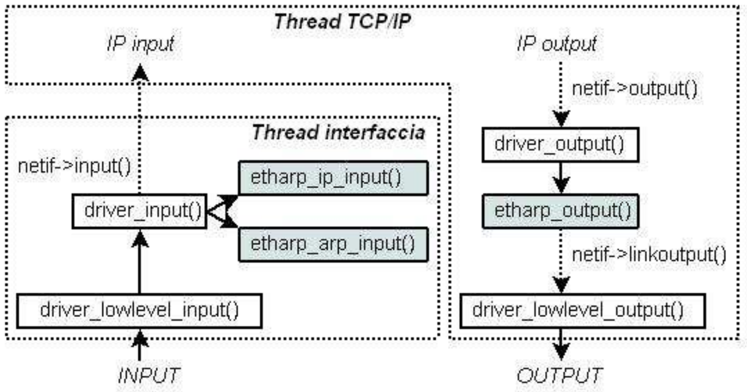

Figure 5.9: Structure of a netif driver

```
struct ip_addr_list *next;
struct ip_addr ipaddr;
struct ip_addr netmask;
struct netif *netif;
char flags;
};
```

The `netif` field keeps the reference to the interface the address belongs to.

Comunication with the stack The network interfaces thread interacts with the stacks thread by using the function pointers `input()`, `output()`, `linkoutput()`, `change()` stored inside the netif structure at initialization time .

For each incoming packet (eg. ARP o IP, it depends on the link type), the interfaces thread delivers it by calling the `input()` function. This function is usually the function `tcpip_input()`. When a new packet is read from the network link, the thread calls this function which simply sends the packet to the main thread by using message-passing. N.B: The interfaces thread handles (read from the link) incoming packets only.

When the stacks need to send outgoing packets, it calls the function `output()`. The routine associated with the `output()` pointer is implemented by the interfaces driver and usually perform link-specific operations before calling the low level function `linkoutput()`. Its duty of `linkoutput()` to "phisically" send (eg. Call `write()` on a pipe or a socket) the outgoing packets.

N.B: LwIPv6 comes with a set of drivers fo ARP protocol handling, but each driver has the job to use these APIs to implement the `output()` function.

Every time its necessary to change the interface state and perform special operations (e.g. flush a cache associated with the link) the `change()` functions is called

Figure 5.9 shows an example of functions called by the interfaces thread and the stacks thread while sending and receiveing packets. In this example the interface driver handles a ethernet linnk and uses the ARP API of LwIPv6.

IP Layer

With some execptions, LwIPv6 handles IPv4 and IPv6 packets inside the same set of functions and witht the help of the same data structures. In the following sections we will show the main steps performed by the stack when a new IP packet is sent or received.

IP Input

Incoming packets are read from the link by the interfaces thread and sent to the main thread throw message-passing. The main thread checks its message queue, pops the packet and calls the `ip_input()` function. For IPv4 packets, the function peforms the checksum validation. The packets destination address is compared with all the incoming network interface. If the destination address and anyone of the interfaces addresses match, che packet is passed to the `ip_inpacket()` function. This function performs IP fragmens reassemblation, if needed, and then delivers the IP packet to the transport layer of the stack.

IP Output

When anyone of the transport protocols needs to send data (TCP segments or UDP datagrams), the function `ip_output()` is called. This function tries to identify the outgoing interface for the given destination and then calls `ip_output_if()`. If the transport layer already knows the outgoing interface, then `ip_output_if()` is called directly. The `ip_output_if()` function performs the IP encapsulation and send the packet on ouput by calling the `netif->output()` function. If the destination address belongs to anyone of the stacks interfaces, the function pushes the packet in the stacks message queue and the packet will be processed as a incoming packet. When the destination IP idetifies a remote host and the packets length its larger than the links MTU, then IP fragmentation is performed.

IP Forwarding

IP Forwarding is an optional feature and can be eigher enabled or disabled at compilation time. This feature is useful only if the stack acts as a network router and uses several interfaces conneceted to different links. If an incoming IP packet needs to be forwarded, the stacks checks the rouring table and then calls the `ip_forward()` function. This routine decrease the `TimeToLive` (TTL) field, or the `Hop-Limit` field (in IPv6), compares the packets length with the ourgoing links MTU and then calls the `ip_output()` function.

Figure 5.10 shows a simplified scheme of the IP layers operations sequence.

IPv6: Missing data structures

The RFC documents about IPv6 propose several data structure the correct management of input and output packets, for example the Neighbour Cache, the Prefix List), the Destination Cache and the Default Router List. All of these are used at the same time by many sub-protocols like the Neighbour

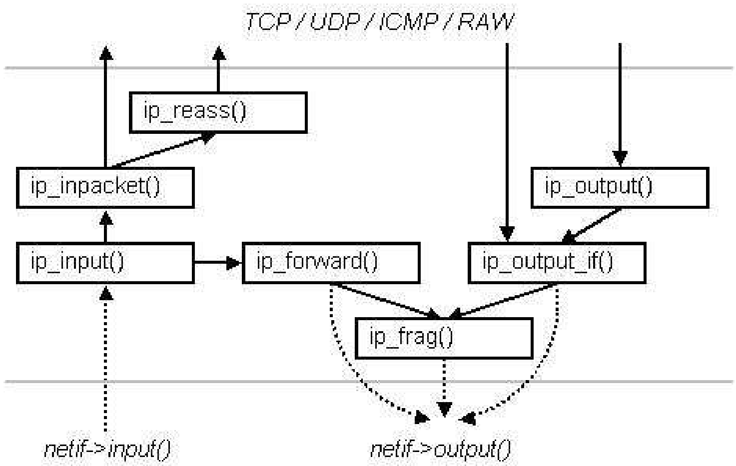

Figure 5.10: LWIP IP layers

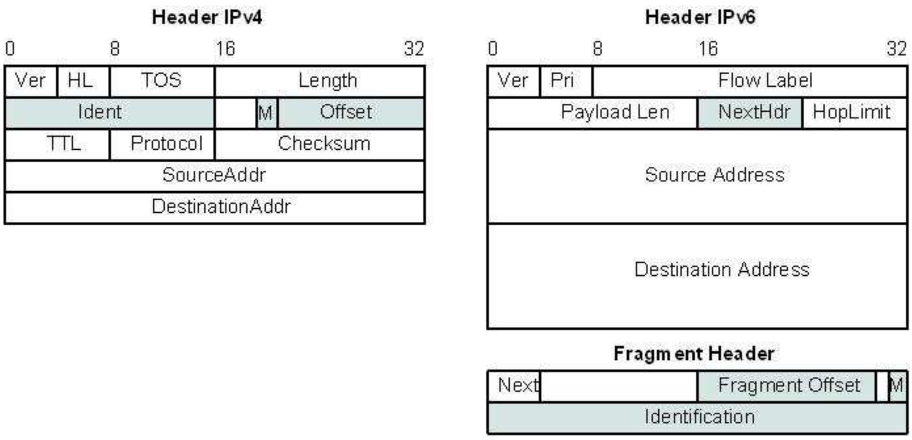

Figure 5.11: Headers for packet fragmentation

Discovery, PMTU Discovery, ecc... To make the stack as simple as possible, LwIPv6 does NOT explicitly implements all these internal data structures. The many informations stored inside these caches are saved and extracted from the existing structures like the ARP table, the routing table, ecc....

IP: Reassembling and Fragmentation

LwIPv6 supports both IPv4 and IPv6 reassembling and fragmentation of datagrams. In Figure 5.11 you can see the headers and fields involved during the packet fragmentation.

Even if IPv4 and IPv6 protocols implement this feature in very different ways, LwIPv6 uses the same data structure for supporting both protocols: a memory buffer and a bit mask used to remember the holes in the IP datagram (see Fig. 5.12.

The maximum number of fragmented datagrams the stack can reassmebly

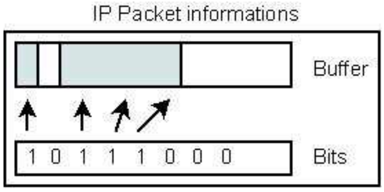

Figure 5.12: Reassembly of a packet

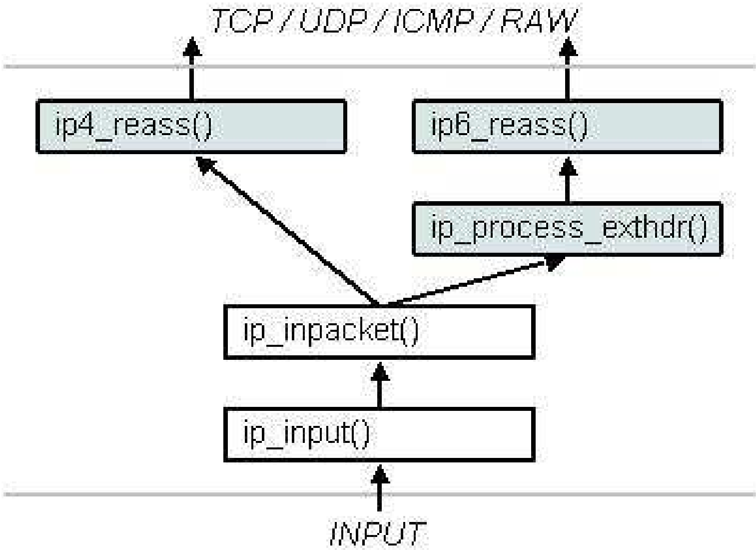

Figure 5.13: Function calls in IP packet reassembly

at the same time is defined by the costant `IP_REASS_POOL_SIZE`. If a packet is not reassembled before `IP4_REASS_MAX_AGE` or `IP6_REASS_MAX_AGE` seconds, then the stack discards every information about that datagram . For each incoming IPv6 packet, the stack calls the `ip_process_exthdr()` which processes IPv6 optional headers, looking for the Fragmentation Header. Reassembling is implemented by two different functions:

```
struct pbuf *ip4_reass(struct pbuf *p);

struct pbuf *ip6_reass(struct pbuf *p,
                       struct ip6_fraghdr *fragext,
                       struct ip_exthdr *lastext);
```

These functions return NULL if no received datagram can be fully reassembled. (See Fig. 5.13)

ICMP

LwIPv6 manages the ICMP layer as well as it manages the IP layer protocols, both ICMPv4 and ICMPv6 are handled by the same stack code.

Incoming ICMP packets, no matter what their version is, are passed to the `icmp_input()` function. This function perform ICMP fields validation and, if its necessary, sends a ICMP response on output.

Up to know, LwIPv6 supports only ECHO and ECHO REPLY messages for both ICMPv4 and ICMPv6. The stack provides a simple working implementation of the ICMPv6 layer: Neighbor Discovery, Router Discovery and Address Autoconfiguration protocols. Their implementation is not complete yet.

Transport Layer

Every connection of the Transport Layer (TCP, UDP) is identified by a connection descriptor which is special data structure called PCB (Protocol Control Block). A PCB saves all the informations about a connection and it is used to handle the protocol session in the proper way.

Each Transport Protocol uses a different and very specific PCB structure, but there are few informations that are common to all PCBs. These informations are: the source and destination IP addresses of the connections, a TOS (Type Of Services) parameter, the TTL (Time to Live) of the outgoing IP packets for that protocol and few additional flags (socket options) used by the Application Level APIs.

All these informations will henceforth be referred to as `IP_PCB` informations.

Protocol Callback functions

Every PCB stores also an other type of information, which is vital for the correct management of a transport connection: a reference (a pointer) to the protocol callback functions. These functions are called by the stack code everytime the connection state changes: the incoming of new data, the establishment of a new connection with a remote host, ecc...

The number and the type of the functions depends to the transport protocol; these differences between every PCB will be analized in the following paragraphs.

UDP

The UDP PCB structure is defined as follow:

```
struct udp_pcb {

  IP_PCB;

  struct udp_pcb *next;

  u8_t flags;
  u16_t local_port, remote_port;
  u16_t chksum_len;

  void (* recv)(void *arg,
              struct udp_pcb *pcb, struct pbuf *p,
              struct ip_addr *addr, u16_t port);
  void *recv_arg;
```

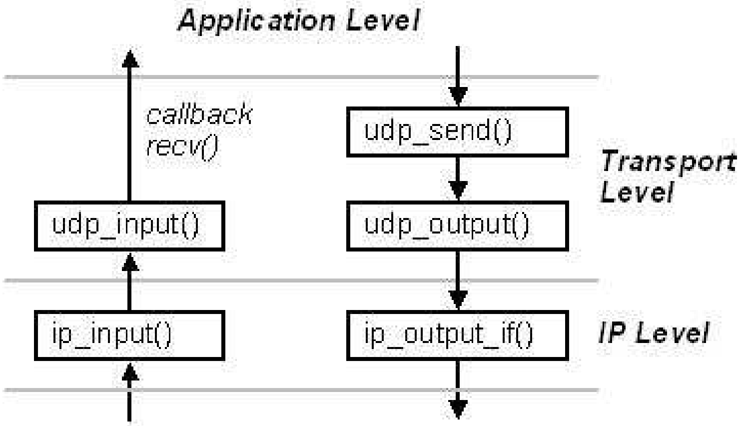

Figure 5.14: Function involved in UDP packets management

```
}
```

In the structure declaration the reader will find the common connection data (`IP_PCB`) and the other fields needed by the UDP layer, in particular the source and destination ports. The function pointer `recv<code> and the <code>recv_arg` field are necessary for the incoming traffic management.. The recv function is the callback used by the application to process all the new incoming UDP datagrams.

The IP Layer passes all the UDP datagrams to the `udp_input()` function and this one scans all the PCBs looking for the right connection and then calls the callback `recv`. When the application need to send new data on output, it calls the `udp_send()` routine which creates the UDP packet and starts the protocol encapsulation mechanism. In Figure 5.14 is showed the global structure of the UDP layer, with both receiving and sending actions.

The application must allocate and initialize a new connection descriptor for each UDP connection. All these operation can be performed by the application itself, but LwIPv6 comes with a implementation of the Socket library which does the hard job and hides a the details.

TCP

The TCP layer looks almost like the UDP layer, but its implementation is more complex, of course. The PCB descriptor for a TCP connection is defined as follow:

```
struct tcp_pcb {

  IP_PCB;

  struct tcp_pcb *next;
  enum tcp_state state;
  u8_t prio;
  void *callback_arg;
```

```
  u16_t local_port;
  u16_t remote_port;
  u8_t flags;

  u32_t rcv_nxt;
  u16_t rcv_wnd;
  u32_t tmr;
  u8_t polltmr, pollinterval;
  u16_t rtime, mss;
  u32_t rttest, rtseq;
  s16_t sa, sv;
  u16_t rto;
  u8_t nrtx;
  u32_t lastack;
  u8_t dupacks;
  u16_t cwnd, ssthresh;
  u32_t snd_nxt, snd_max, snd_wnd, snd_wl1, snd_wl2,
        snd_lbb;
  u16_t acked;
  u16_t snd_buf;
  u8_t snd_queuelen;
  struct tcp_seg *unsent, *unacked, *ooseq;
  u32_t keepalive;
  u8_t keep_cnt;

  err_t (* sent)      (void *arg, struct tcp_pcb *pcb,
                       u16_t space);
  err_t (* recv)      (void *arg, struct tcp_pcb *pcb,
                       struct pbuf *p, err_t err);
  err_t (* connected)(void *arg, struct tcp_pcb *pcb,
                       err_t err);
  err_t (* accept)    (void *arg,
                       struct tcp_pcb *newpcb,
                       err_t err);
  err_t (* poll)      (void *arg, struct tcp_pcb *pcb);
  void  (* errf)      (void *arg, err_t err);
};
```

As the reader can read in the previews structure, several informations are needed by the TCP state machine code. All the fields are used to implement the Congestion Control, the Fast Recovery/Fast Retrasmit mechanism and the Round-Trip Time Estimation.

RAW Connections/Protocols

LwIPv6 can handle RAW connections. This means that the Application Level can send hand-crafted IP datagrams and must read and process all incoming packets. The stacks manages these special connections as like as any other TCP or UDP connection. The PCB used for RAW connections is defined as:

```
struct raw_pcb {

  IP_PCB;

  struct raw_pcb *next;
  u16_t in_protocol;

  void (* recv)(void *arg, struct raw_pcb *pcb,
              struct pbuf *p,
              struct ip_addr *addr, u16_t protocol);
  void *recv_arg;
};
```

It looks like the UDP descriptor: there is a callback function (`recv()`), but the structure keeps informations only about the Transport protocol (`in_protocol`) the application want to manage.

When a new IP datagram is received, the stacks calls the `raw_input()` function before passing the packet to the transport layer. This functions checks if anyone of the active RAW connections matches with the new packet and then calls the callback functions. On output, the application level sends raw data with the `raw_sendto()` and then the IP layer encapsulates the application datagram with the `ip_output_if()`.

Other Images

Figure 5.15 is an other rapresentation of the function calling sequence inside the stack. The picture shows a stack with two virtual interfaces: a VDE interface and a TUNTAP interface. Figure 5.16 is a very streamlined view of an application running LwIPv6.

5.4 ▼LWIPv6 in education

Two different kinds of exercises and projects can use LWIPv6: the first type uses LWIPv6, the second is based on modifying LWIPv6 code.

LWIPv6 is a networking stack as a library, so it may appear useless for projects and exercirses as the students can use the networking stack provided by the kernel. The main feature that the students can test only by LWIPv6 is the ability write programs which operates on multiple networking stacks at the same time.

Exercises may include:

- design and implement an application tunnel between two stacks: the program should receive service requests from one stack and forward them to the other. Some kind of connection tracking should permit the reply messages (for connection free services) to return back to the requester. Connection based services need that a new connection to the server is created for each one accepted from a client.
- write a "partial slirp6" service. The slirpv6 command is able to define only slirp interfaces as default route while LWIPv6 is able to use slirp

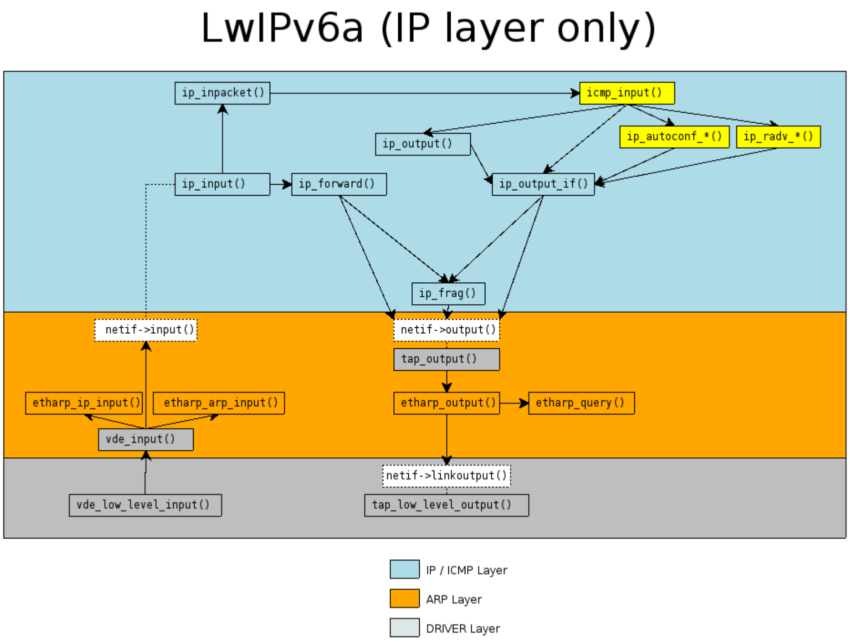

Figure 5.15: Sequence of calls (limited to the management of the IP protocol)

as any other interface routing to slirp just some networks. The project consists in designing and implementing an *extended* slirpv6 able to use several interfaces and a more complete management of routing.

While it is common to have a networking stack to use, the ability to modify a networking stack is not common. LWIPv6 enables several projects and exercises.

- The size of the TCP window can be a bottleneck for *long fat networks*[19], network having high bandwith/high latence links, like satellite networks. The students can test this problem using wirefilter to create delays and changing the size of the TCP window in the code of LWIPv6.
- Create a ppp interface for the stack. Lwipv6 should connect to a sibling lwipv6 stack on a bidirectional stream or to a real system over a serial line.
- Design and implement different queueing disciplines and/or Quality of Service features in LWIPv6.
- Design and implement a (minimal) support for IP multicast on LWIPv6.

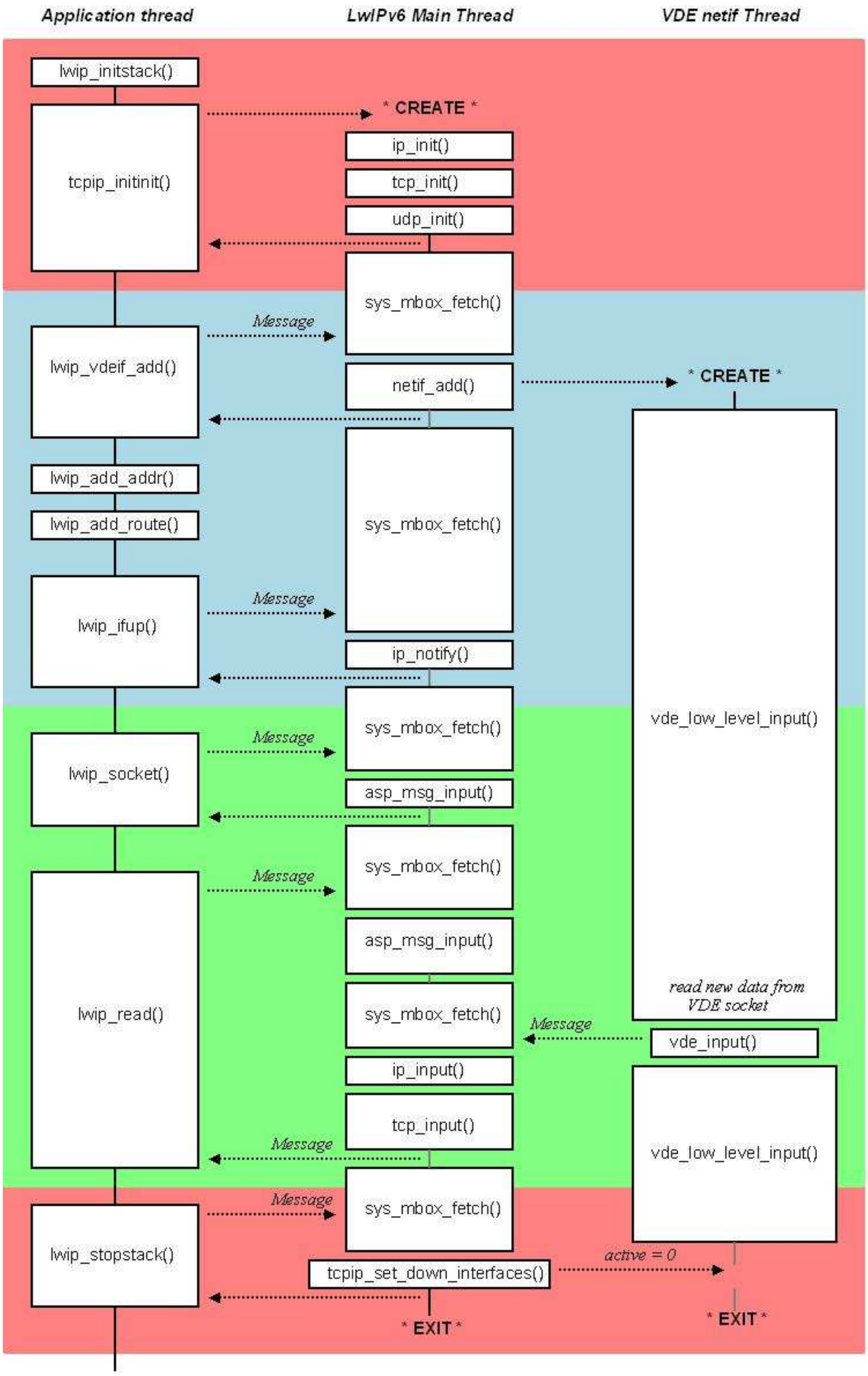

Figure 5.16: Interactions between an application and the LWIPv6 library

CHAPTER 6

Inter Process Networking

Inter Process Networking exploits the networking methodologies in the context of Inter Process Communication[1]: a process acts like a network host, "talking" with the other processes through routines typical of a networking system. IPN supports kvde_switch, a very efficient vde switch which dispatches the packets entirely in kernel space, but IPN can be effectively used for many other uses like multimedia Mpeg-TS stream multiplexing or service state multicasting.

IPN[2] is an Inter Process Communication service that uses the same programming interface and protocols used for networking: processes using IPN are connected to a "network" (many to many communication), and "talk" to each other like network hosts do.

The messages or packets sent by a process on an IPN network can be delivered to many other processes connected to the same IPN network, potentially to *all* the other processes. Several protocols can be defined on the IPN service:

broadcast (level 1) protocol: all the packets are received by all the processes except the sender

ethernet/internet protocols: IPN sockets can dispatch packets using the Ethernet protocol (like a Virtual Distributed Ethernet - VDE switch), or the Internet Protocol (like a layer 3 switch)

These are only examples, but the basic idea is open to the implementation of more sophisticated protocols and policies.

[1]IPC is a set of techniques for the exchange of data among two or more threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory and remote procedure calls (RPC).

[2]http://wiki.virtualsquare.org/index.php/IPN

6.1 IPN usage

The Berkeley socket Application Programming Interface (API) was designed for client server applications and for point-to-point communications. There is not a support for broadcasting/multicasting domains.

IPN updates the interface introducing a new protocol family (PF_IPN or AF_IPN). PF_IPN is similar to PF_UNIX but for IPN the Socket API calls have a different, extended behavior.

```
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/ipn.h>

sockfd = socket(PF_IPN, int socket_type, int protocol);
```

This example shows how a communication socket is created: the only `socket_type` defined is `SOCK_RAW`, while other socket_types can be used for future extensions. A socket cannot be used to send or receive data until it's connected (using the "connect" call). The `protocol` argument defines the policy used by the socket:

IPN_ANY (0): this protocol can be used to connect or bind a pre-existing IPN network regardless of the policy used, because the policy has been defined by somebody else;

IPN_BROADCAST (1): this protocol is the basic policy, a packet is sent to all the receipients except the sender;

IPN_SWITCH (2): Layer 2 switching, VLANs;

IPN_L3 (3): not implemented yet. Layer 3 (network) routing.

The address format is the same one of PF_UNIX (a.k.a PF_LOCAL), as described in the unix(7) manual. In order to create an IPN network (when it doesn't exist), the bind() call must be used:

```
int bind(int sockfd, const struct sockaddr *my_addr,
               socklen_t addrlen);
```

If the IPN exists already, the call can be used to join an existing network (only for management). The policy of the network must be consistent with the protocol argument of the `socket()` call. The policy for a new network is already defined in the socket `sockfd`, while the `bind()` and `connect()` calls invoked on an existing network will fail if the policy of the socket is neither IPN_ANY nor the same of the existing network. Note that a network should not be started with an IPN_ANY socket.

An IPN network appears in the file system as a Unix socket. Execution permission (x) on this file is required for the `bind()` call to succeed (otherwise EPERM is returned); similarly, the read/write permissions (rw) allow the `connect()` call respectively to receive (read) and send (write) packets. When a socket is bound (but not connected) to an IPN network, the process does not receive or send any data but it can call `ioctl()` or `setsockopt()` to configure the network.

The call required to connect a socket to an existing IPN network is:

```
int connect(int sockfd, const struct sockaddr *serv_addr,
               socklen_t addrlen);
```

The socket could be still unbound, or already bound through the `bind()` call: unbound connected sockets receive and send data, but they cannot configure the network. The read/write permissions on the socket (rw) are required to "connect" to the channel and send/receive (read/write) packets.

When the `connect()` call succeeds, and the socket is provided with appropriate permissions, the process can send packets and receive all the packets sent by other processes and delivered according the network policy. The socket can receive data at any time, like in a network interface: this requires the process to be able to handle continuously incoming data (using `select()/poll()` or multithreading).

Obviously, higher lever protocols could prevent the reception of unexpected messages by design: it is the case of networks used with with *exactly one* sender, where all the other processes can simply receive the data, while the sender will never receive any packet.

It is also possible to have sockets with different roles assigning reading permission to some, and writing permissions to others. If data overrun occurs, there can be data loss or the sender can be blocked, depending on the policy of the socket (LOSSY or LOSSLESS). The `bind()` call must be invoked before the `connect()` call. The correct sequences are:

- socket+bind: only for management
- socket+bind+connect: management *and* communication
- socket+connect: communication *without* management

Since there isn't any server, the calls `accept()` and `listen()` are not defined for AF_IPN: all the communication takes place among peers.

Data can be sent and received using `read()`, `write()`, `send()`, `recv()`, `sendto()`, `recvfrom()`, `sendmsg()`, `recvmsg()`.
The socket options and flags are:

IPN_FLAG_LOSSY: in case of network overloading, or data overrun (when some processes are too slow in consuming the packets from the buffer), packets are dropped from the buffer;

IPN_FLAG_LOSSLESS: in case of network overloading or data overrun, the network blocks the sender until the buffer make space for new entries;

IPN_SO_NUMNODES: max number of connected sockets;

IPN_SO_MTU: maximum transfer unit (maximum size of packets);

IPN_SO_MSGPOOLSIZE: size of the buffer (number of pending packets);

IPN_SO_MODE: this option specifies the permission to use when the socket is created on the file system. It is modified by the process' umask in the usual way;

IPN_SO_PORT: this option specifies the port number where the socket must be connected. When IPN_PORTNO_ANY is set, the port number is decided by the service. This is the default option. On some network services, such as VLANs on virtual Ethernet switches, different ports may have different definitions;

IPN_SO_DESCR: this is the description of the node as a string, with maxlength IPN_DESCRLEN. This option is used only for debugging.

IPN sockets can be connected to virtual and real network interfaces using specific ioctl, but only if the user has the permission to configure the network (e.g. the CAP_NET_ADMIN POSIX capability). A virtual interface connected to an IPN network is similar to a tap interface: a tap interface appears as an ethernet interface to the hosting operating system, where all the packets sent and received through the tap interface are sent and received by the application which created that interface. The IPN virtual network interface acts in the same way: packets are received and sent through the IPN network and delivered consistently with the defined policy (IPN_BROADCAST acts as a basic HUB for the connected processes).

It is also possible to *grab* a real interface: in this case the closest example is the Linux kernel ethernet bridge. When a real network is connected to an IPN, all the packets received from the real network are injected also into the IPN and all the packets sent by the IPN through a defined "port" are sent to the real network. The function `ioctl()` is used for creation and control of TAP or GRAB interfaces.

```
int ioctl(int d, int request, .../* arg */);
```

The currently supported `request` options are:

IPN_CONN_NETDEV: this request needs an argument of type `struct ifreq *arg`, and creates a TAP interface or implements a GRAB on an existing interface and connects it to a bound IPN socket. The field `ifr_flags` can be IPN_NODEFLAG_TAP for a TAP interface, or IPN_NODEFLAG_GRAB to grab an existing interface. The field `ifr_name is` the desired name for the new TAP interface or is the name of the interface to grab (e.g. eth0). In the case of TAP interfaces, `ifr_name` can be an empty string, while in the GRAB case, the interface is named *ipn* followed by a number (e.g. ipn0, ipn1, ...). This `ioctl` must be used on a bound but unconnected socket: when the call succeeds, the socket gains the connected status, but the packets are sent and received through the interface. Persistence apply only to interface nodes (TAP or GRAB);

IPN_SETPERSIST: this request needs an argument of type `int arg`. If `(arg != 0)` the interface gains the persistent status: the network interface survives and remains connected to the IPN network when the socket is closed. If `(arg == 0)` the standard behavior is resumed: the interface is deleted or the grabbing is terminated when the socket is closed;

IPN_JOIN_NETDEV: this request needs an argument of type `struct ifreq *arg`, and reconnects a socket to an existing persistent node. The interface can be defined either by name (`ifr_name`) or by index (`ifr_index`).

If there is already a socket controlling the interface this call fails (EADDRNOTAVAIL).

Some other `ioctl` requests can be used by a system administrator to give/clear persistence on existing IPN interfaces. These calls apply only to unbound sockets:

IPN_SETPERSIST_NETDEV: this request needs an argument of type `struct ifreq *arg`, and sets the persistence status of an IPN interface. The interface can be defined either by name (`ifr_name`) or by index (`ifr_index`);

IPN_CLRPERSIST_NETDEV: this request needs an argument of type `struct ifreq *arg`, and clears the persistence status of an IPN interface. The interface is specified as in the same way as IPN_SETPERSIST_NETDEV. In the TAP case, the interface is deleted, while in the GRAB case the grabbing is terminated when the socket is closed, or immediately if the interface is not controlled by a socket. If the specified node was the only node in the IPN network, the IPN network is terminated too.

IPN_REGISTER_CHRDEV: It is possible to define character devices connected to IPN networks. When a process opens a character device connected to an IPN network, read and write operations on the device operate like receive or send operations on a socket. Protocol submodules can identify when the network nodes communicate by device, and also which is the device involved. In this way protocol submodules provide different services on specific devices.

The ioctl tag IPN_REGISTER_CHRDEV can be used to define or allocate one or more character devices. The argument of IPN_REGISTER_CHRDEV is a pointer to a structure chrdevreq:

```
struct chrdevreq {
  unsigned int major;
  unsigned int minor;
  int count;
  char name[64];
};
```

Major and minor identify the device or the first device of the range. If major is zero IPN dynamically allocate a major number for the device (the field is updated by ioctl, it is possible to read the assigned major number after the call). When major is nonzero IPN register that device, an error occurs in case the major is already registered by another device. count is the number of devices requested: IPN assigns a range of minor numbers from minor to minor+count-1. name is the name of the device. IPN_REGISTER_CHRDEV works on a IPN socket already bound to an IPN network, requires the CAP_MKNOD capability, defines the sysfs nodes, and it is compatible with udev.

IPN_UNREGISTER_CHRDEV: is the tag for unregistering a device range, it has no arguments, the device or the device range of the IPNN (must run on a bound socket) get released. It is not possible to unregister just some of the devices, all the allocated range must be unregistered as a whole.

Normally when the last process of an IPN network close the socket (or the descriptor related to a device of the IPN), the network gets deleted. It is possible to define an IPN network (with devices) to be "persistent": the IPN network will survive even when no processes are connected. IPN_CHRDEV_PERSIST is the tag that allow to set/unset the "persistency" of an IPN network, it requires an "int" argument: the network becomes persistent if the argument is non zero, non-persistent otherwise.

IPN_JOIN_CHRDEV is the ioctl tag to bind the IPN network associated with a character device. IPN_JOIN_CHRDEV works on a unbound IPN socket bound and requires the CAP_MKNOD capability. The argument for IPN_JOIN_CHRDEV is a pointer to a struct chrdevreq. It is the only way to change the persistence of a IPN network when the socket has been removed from the file system.

When unloading the *ipnvde* kernel module, all the persistent flags of the interfaces are cleared.

Finally, it's possible to write programs that forward packets between different IPN networks running on the same or on different systems, extending the IPN in the same way as cables extend ethernet networks, connecting switches or hubs together. VDE cables are examples of such a kind of programs.

6.2 ★Compile and install IPN

IPN is not yet included in any distribution nor in the kernel source. IPN needs some dedicated structures in the kernel: it needs the attribution of a Protocol Family and some fields for the interface grabbing feature. IPN cannot currently run on standard kernels. For this reason IPN can be compiled in *stealing* mode. If the constant "IPN_STEALING" is defined during the compilation, IPN *steals* the Protocol Family from AF_NETBEUI (which is not implemented in the kernel) and the fields for interface grabbing from the ones defined for the kernel bridge.

IPN is available as kernel patch or as external module. Both versions currently use AF_NETBEUI as AF_IPN has not been officially assigned yet. The kernel patch includes the IPN code, too.

If IPN is compiled in *stealing* mode it is not possible to load IPN and the kernel bridge at the same time as they share the same fields.

To compile IPN in real mode download the IPN kernel patch consistent with your kernel version from the VDE repository. Apply the patch, from the root directory of the linux kernel tree:

```
# patch -p 1 < patch-linux-2.6.25.4-ipn
```

Use your favourite configuration tool, like menuconfig or xconfig; in the networking menu there is a new option:

```
IPN domain sockets (EXPERIMENTAL)
```

Select the item, compile and install the kernel.

For the *stealing* mode, download the IPN source tree. The makefile in the root directory is already configured for the *stealing* mode. In fact, it includes the following definition:

```
EXTRA_CFLAGS += -DIPN_STEALING
```

Compile and install in the usual way (`make` and `make install`).

The IPN kernel module `ipn.ko` can be loaded as any module by `insmod` or `modprobe`.

The system log will report the new protocol loaded:

```
Aug 24 19:49:08 v2host kernel: NET: Registered protocol family 13
Aug 24 19:49:08 v2host kernel: IPN: Virtual Square Project, University of Bologna 2007
```

(family 13 is for the *stealing* mode).

It is a good idea for application programmers to set up the code to test both protocol families, AF_NETBEUI for the stealing mode and AF_IPN for the real mode, when trying to join an IPN network. AF_IPN has not been officially assigned yet, but the code in this way is ready and backwards compatible. libvdeplug source code is an example of IPN client compatible with real and stealing mode.

6.3 IPN usage examples

This section shows some notes on how to use IPN sockets. All the examples require:

```
#include <net/af_ipn.h>
```

If the IPN module has been compiled using the *stealing* mode define `IPN_STEALING` before including the header file:

```
#define IPN_STEALING
#include <net/af_ipn.h>
```

It is possible to write applications able to use either the real IPN mode or the *stealing* mode. When `IPN_STEALING` is *not* defined `AF_IPN_STOLEN` is defined as the stolen family of addresses.

```
int ipn_af=AF_IPN;
int s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST);
if (s< 0) {
  ipn_af=AF_IPN_STOLEN;
  s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST);
  if (s< 0)
    error....
}
```

The address family `ipn_af` must be used also for defining addresses (`struct sockaddr`).

Communication among peers

Each process execute the same code to join the network:

```
int ipn_af=AF_IPN;
struct sockaddr_un sun={.sun_family=ipn_af,.sun_path="/tmp/sockipn"};
int s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST); /* or a different protocol */
err=bind(s,(struct sockaddr *)&sun,sizeof(sun));
err=connect(s,NULL,0);
```

Please note that here and throughout all the examples of this section, the error management has been skipped for the sake of clearness. In the code here above the values s and err when negative should activate error management actions.

Using `IPN_BROADCAST` each process receives all the messages sent by the others. The socket is closed by close. When the last process leaves the network, the network itself is terminated (the socket in the file system must be removed using unlink). If the socket gets unlinked when it is already in use, the socket survives: all the processes using the socket at the deletion time continue to use it, the socket will cease to exist only when the last process closes it. It is not possible binding or connecting to an unlinked socket. The management is similar to unlinking open files, commonly used for temporary files. If a process binds a socket to the same name of an already existing but unlinked socket it creates a new IPN network unrelated with the previous one.

Broadcast IPN, one sender, many receivers

The sender runs the following code:

```
int ipn_af=AF_IPN;
struct sockaddr_un sun={.sun_family=ipn_af,.sun_path="/tmp/sockipn"};
int s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST); /* or a different protocol */
err=shutdown(s,SHUT_RD);
err=bind(s,(struct sockaddr *)&sun,sizeof(sun));
err=connect(s,NULL,0);
```

The receivers do not need to manage the network, thus they skip the bind and execute:

```
int ipn_af=AF_IPN;
struct sockaddr_un sun={.sun_family=ipn_af,.sun_path="/tmp/sockipn"};
int s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST); /* or a different protocol */
err=shutdown(s,SHUT_WR);
err=connect(s,(struct sockaddr *)&sun,sizeof(sun));
```

For a protected service it is possible to create a user or a group and apply appropriate access mode.

if the sender add this statement (e.g. just after the `SHUT_RD` shutdown):

```
int mode=0744;
err=setsockopt(s,IPN_SO_MODE,&mode,sizeof(mode));
```

the receivers belonging to other users cannot send anything to the IPN network. The shutdown statement for those receivers is useless and can be deleted. (the actual access mode takes into account umask in the usual way, all the bits set in the mask will be cleared in the access mode).

`Lossless service`

To create a lossless service add these statements before bind:

```
int flags=IPN_FLAG_LOSSLESS;
int size=64; /* standard size is 8, it is too small for LOSSLESS */
err=setsockopt(s,0,IPN_SO_FLAGS,&flags,sizeof(flags));
err=setsockopt(s,0,IPN_SO_MSGPOOLSIZE,&size,sizeof(size));
```

The value of `IPN_SO_MSGPOOLSIZE` must be increased. While for LOSSY service it is the number of pending message per node, for LOSSLESS service this is the overall number of pending messages in the network.

IPN network connected to a TAP interface

```
int ipn_af=AF_IPN;
struct sockaddr_un sun={.sun_family=ipn_af,.sun_path="/tmp/sockipn"};
int s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST);
struct ifreq ifr;
err=bind(s,(struct sockaddr *)&sun,sizeof(sun));
memset(&ifr, 0, sizeof(ifr));
strncpy(ifr.ifr_name, "test", IFNAMSIZ);
ifr.ifr_flags=IPN_NODEFLAG_TAP;
err=ioctl(s, IPN_CONN_NETDEV, (void *) &ifr);
```

Using `IPN_BROADCAST` all the packets sent on the IPN networks will appear as incoming packets on the TAP interface and viceversa: all the packets sent by the TAP interface will be received by all the nodes connected to the IPN network.

IPN network connected to a GRAB interface

```
int ipn_af=AF_IPN;
struct sockaddr_un sun={.sun_family=ipn_af,.sun_path="/tmp/sockipn"};
int s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST);
struct ifreq ifr;
err=bind(s,(struct sockaddr *)&sun,sizeof(sun));
memset(&ifr, 0, sizeof(ifr));
strncpy(ifr.ifr_name, "eth1", IFNAMSIZ);
ifr.ifr_flags=IPN_NODEFLAG_GRAB;
err=ioctl(s, IPN_CONN_NETDEV, (void *) &ifr);
```

The code is very similar to the one used for the TAP interface. In this case the interface must already exist. All the packets received by the interface (in this case from the real ethernet eth1) will be sent over the IPN network, and all the packets sent on the GRABBED port by the IPN network will be sent also an the real ethernet. In this way it is possible to extend an Ethernet with an IPN network.

Out of Band notification of the number of senders/receivers

```
int ipn_af=AF_IPN;
struct sockaddr_un sun={.sun_family=ipn_af,.sun_path="/tmp/sockipn"};
int s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST);
int want=1;
int nreaders;
struct pollfd pfd[...]={{0,POLLPRI,0},...};
err=setsockopt(s,0,IPN_SO_HANDLE_OOB,&want,sizeof(want));
err=setsockopt(s,0,IPN_SO_WANT_OOB_NUMNODES,&want,sizeof(want));
err=shutdown(s,SHUT_RD);
err=connect(s,(struct sockaddr *)&sun,sizeof(sun));
pfd[0].fd=s;
while (poll(pfd,...,...) >= 0) {
  if (pfd[0].revents & POLLPRI) {
    struct numnode_oob *nn = (struct numnode_oob *) buf;
    len=read(s,buf,256);
    nreaders= nn->numreaders;
  }
  /* the code here can decide different behavior depending on nreaders.
  maybe the program stops to send data on the net where there are no
  receivers */
}
```

IPN sends notifications from protocols as OOB messages. OOB messages are delivered first (preempting all waiting normal messages) together with normal messages, if necessary they can be recognized using recvmsg system call: `MSG_OOB` is set on `msg_flags`. The example above is a sender (maybe a broadcasting daemon) stopping to send data when no receivers are registered on the network, and resuming its service as soon as a new receiver join the net.

A minimal VDE hub using IPN broadcast

The basic IPN service IPN_BROADCAST implements an ethernet hub. All the dispatching of packets is managed by the kernel. The user mode code stays idle just to keep the socket open.

The source code of a minimal VDE hub is in Figure 6.1.

This hub can be launched by the command:

```
$ ./minihub /tmp/vdetest
```

All the VDE tools (like vdei_plug, qemu, kvm, user-mode linux) can use the network defined by `/tmp/vdetest`.

6.4 ★kvde_switch, a VDE switch based on IPN

`kvde_switch` is a user level application and an IPN submodule.

If the IPN module has been correctly compiled installed and loaded either in real on in *stealing* mode, the `kvde_switch` module can be compiled and installed (`make`, `make install`).

`kvde_switch` generates two modules: `ipn_hash.ko` and `kvde_switch.ko`.

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <net/af_ipn.h>

main(int argc,char *argv[])
{
  int ipn_af=AF_IPN;
  int s=socket(ipn_af,SOCK_RAW,IPN_BROADCAST);
  int err;
  struct sockaddr_un sun;
  if (s< 0) {
    ipn_af=AF_IPN_STOLEN;
    s=socket(AF_IPN_STOLEN,SOCK_RAW,IPN_BROADCAST);
    if (s< 0)
      perror("socket");
  }
  strcpy(sun.sun_path,argv[1]);
  sun.sun_family=ipn_af;
  err=bind(s,(struct sockaddr *)&sun,sizeof(sun));
  if (err<0)
    perror("bind");
  while (pause())
    ;
  close(s);
}
```

Figure 6.1: minihub.c: A minimal VDE hub based on IPN.

The `kvde_switch` application needs `ipn.ko`, `ipn_hash.ko` and `kvde_switch.ko` loaded.

`kvde_switch` and `vde_switch` have the same syntax. `kvde_switch` has far fewer feature than the user mode vde switch.

```
$ kvde_switch -s /tmp/ks

vde$ help
0000 DATA END WITH '.'
COMMAND PATH       SYNTAX          HELP
------------       --------------  ------------
ds                 ============    DATA SOCKET MENU
ds/showinfo                        show ds info
help               [arg]           Help (limited to arg when specified)
logout                             logout from this mgmt terminal
shutdown                           shutdown of the switch
showinfo                           show switch version and info
load               path            load a configuration script
.
1000 Success
```

```
vde$
```

kvde_switch has the following command line options:

```
$ kvde_switch -h
Usage: vde_switch [OPTIONS]
Runs a VDE switch.
(global opts)
  -h, --help                 Display this help and exit
  -v, --version              Display informations on version and exit
(opts from datasock module)
  -s, --sock SOCK            control directory pathname
  -s, --vdesock SOCK         Same as --sock SOCK
  -s, --unix SOCK            Same as --sock SOCK
  -m, --mod MODE             Standard access mode for comm sockets (octal)
  -g, --group GROUP          Group owner for comm sockets
  -t, --tap TAP           Enable routing through TAP tap interface
  -G, --grab INT          Enable routing grabbing an existing interface
(opts from consmgmt module)
  -d, --daemon               Daemonize vde_switch once run
  -p, --pidfile PIDFILE      Write pid of daemon to PIDFILE
  -f, --rcfile               Configuration file (overrides ...)
  -M, --mgmt SOCK            path of the management UNIX socket
      --mgmtmode MODE        management UNIX socket access mode (octal)
```

There is the grab feature (`-G`) to include a real interface in a KVDE network. Using KVDE all the dispatching of packets takes place in the kernel code.

A complete implementation of `vde_switch` features into `kvde_switch` is under way.

6.5 ▲IPN protocol submodules

IPN provides the standard broadcast policy, and there is already a module providing `IPN_SWITCH`. IPN can be used for many other application that may have specific requirements and switching policies. As a test IPN was used to split a MPEG-TS stream: using one satellite DVB-S board different applications (mplayer it the test) were able to receive one of the programs broadcast in the stream.

This section explains how to write your own policy module that will be loaded as a kernel module after ipn.ko.

```
struct ipn_protocol {
  int refcnt;
  int (*ipn_p_newport)(struct ipn_node *newport);
  int (*ipn_p_handlemsg)(struct ipn_node *from,
              struct msgpool_item *msgitem, int depth);
  void (*ipn_p_delport)(struct ipn_node *oldport);
  void (*ipn_p_postnewport)(struct ipn_node *newport);
  void (*ipn_p_predelport)(struct ipn_node *oldport);
```

```
  int (*ipn_p_newnet)(struct ipn_network *newnet);
  int (*ipn_p_resizenet)(struct ipn_network *net,int oldsize,int newsize);
  void (*ipn_p_delnet)(struct ipn_network *oldnet);
  int (*ipn_p_setsockopt)(struct ipn_node *port,int optname,
  char __user *optval, int optlen);
  int (*ipn_p_getsockopt)(struct ipn_node *port,int optname,
  char __user *optval, int *optlen);
  int (*ipn_p_ioctl)(struct ipn_node *port,unsigned int request,
            unsigned long arg);
};

int ipn_proto_register(int protocol,struct ipn_protocol *ipn_service);
int ipn_proto_deregister(int protocol);

void ipn_proto_sendmsg(struct ipn_node *to, struct msgpool_item *msg,
               int depth);
```

A protocol (sub) module must define its own `ipn_protocol` structure (maybe a global static variable).

`ipn_proto_register` must be called in the module init to register the protocol to the IPN core module. `ipn_proto_deregister` must be called in the destructor of the module. It fails if there are already running networks based on this protocol.

Only two fields must be initialized in any case: `ipn_p_newport` and `ipn_p_handlemsg`.

`ipn_p_newport` is the new network node notification. The return value is the port number of the new node. This call can be used to allocate and set private data used by the protocol (the field `proto_private` of the struct `ipn_node` has been defined for this purpose).

`ipn_p_handlemsg` is the notification of a message that must be dispatched. This function should call `ipn_proto_sendmsg` for each recipient. It is possible for the protocol to change the message (provided the global length of the packet does not exceed the MTU of the network). Depth is for loop control. Two IPN can be interconnected by kernel cables (not implemented yet). Cycles of cables would generate infinite loops of packets. After a pre-defined number of hops the packet gets dropped (it is like EMLINK for symbolic links). Depth value must be copied to all `ipn_proto_sendmsg` calls. Usually the handlemsg function has the following structure:

```
static int ipn_xxxxx_handlemsg(struct ipn_node *from,
                     struct msgpool_item *msgitem, int depth)
{
   /* compute the set of receipients */
   for (/*each receipient "to"*/)
        ipn_proto_sendmsg(to,msgitem,depth);
}
```

It is also possible to send different packets to different recipients.

```
struct msgpool_item *newitem=ipn_msgpool_alloc(from->ipn);
/* create a new contents for the packet by filling in
```

```
   newitem->len and newitem->data */
ipn_proto_sendmsg(recipient1,newitem,depth);
ipn_proto_sendmsg(recipient2,newitem,depth);
....
ipn_msgpool_put(newitem);
```

The packet (`msgpool_item`) passed as a parameter is automatically freed by IPN while for the packets allocated by the protocol submodule `ipn_msgpool_put` after all the sendmsg calls.

`ipn_p_delport` is used to deallocate port related data structures.

`ipn_p_postnewport` and `ipn_p_predelport` are used to notify new nodes or deleted nodes. newport and delport get called before activating the port and after disactivating it respectively, therefore it is not possible to use the new port or deleted port to signal the change on the net itself. `ipn_p_postnewport` and `ipn_p_predelport` get called just after the activation and just before the deactivation thus the protocols can already/still send packets on the network.

`ipn_p_newnet` and `ipn_p_delnet` notify the creation/deletion of a IPN network using the given protocol.

`ipn_p_resizenet` notifies a number of ports change

`ipn_p_setsockopt` and `ipn_p_getsockopt` can be used to provide specific socket options.

`ipn_p_ioctl` protocols can implement also specific ioctl services.

A complete example

The example presented in Figure 6.2 is a test policy submodule that splits MPEG-TS streams.

Each program can register to receive two streams (usually video and audio), and the two program identifiers (PID) are stored in the 16 bit halves of the `proto_private` field used as an integer. A client program registers its pid calling an ioctl 4444 with an integer parameter containing the two pids. `ipn_kmpegts_handlemsg` delivers each packets to all those clients requiring the PID of the packet or 8192 which is the wildcard for any pid.

The source code can be compiled with a standard Makefile for kernel modules like the following one:

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
#KERNELDIR = "/path/of/linux-2.6.xx-ipn"

ifneq ($(KERNELRELEASE),)
  mpegts-objs := mpegts_main.o
  obj-m += mpegts.o
# Otherwise we were called directly from the command
# line; invoke the kernel build system.
else
  KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

modules:
```

```
#include <linux/module.h>
#include <linux/if_ether.h>
#include <sys/af_ipn.h>
MODULE_LICENSE("GPL");
MODULE_AUTHOR("VIEW-OS TEAM");
MODULE_DESCRIPTION("MPEG TS switch Kernel Module");
#define IPN_MPEGTS 8

static int ipn_kmpegts_newport(struct ipn_node *newport) {
  newport->proto_private = (void *) 0; return 0;
}

static int ipn_kmpegts_handlemsg(struct ipn_node *from,
    struct msgpool_item *msgitem){
  struct ipn_network *ipnn=from->ipn;
  struct ipn_node *ipn_node;
  if (msgitem->len > 4 && msgitem->data[0] == 0x47) {
    int mypid=((msgitem->data[1] << 8) | msgitem->data[2]) & 0x1fff;
    list_for_each_entry(ipn_node, &ipnn->connectqueue, nodelist) {
      int pid1 = ((unsigned int) ipn_node->proto_private) >> 16;
      int pid2 = ((unsigned int) ipn_node->proto_private) & 0xffff;
      if (ipn_node != from) {
        if (pid1 == 8192 || pid1 == mypid || pid2 == mypid)
        ipn_proto_sendmsg(ipn_node,msgitem);
      }
    }
  }
  return 0;
}

static int ipn_kmpegts_newnet(struct ipn_network *newnet) {
  if (!try_module_get(THIS_MODULE))
    return -EINVAL;
  return 0;
}

static void ipn_kmpegts_delnet(struct ipn_network *oldnet) {
  module_put(THIS_MODULE);
}

static int ipn_kmpegts_ioctl(struct ipn_node *port,unsigned int request,
    unsigned long arg) {
    if (request == 4444) {
      port->proto_private = (void *) arg;
      return 0;
    }
    return -EOPNOTSUPP;
}

static struct ipn_protocol mpegts_proto={
  .ipn_p_newport=ipn_kmpegts_newport,
  .ipn_p_handlemsg=ipn_kmpegts_handlemsg,
  .ipn_p_newnet=ipn_kmpegts_newnet,
  .ipn_p_delnet=ipn_kmpegts_delnet,
  .ipn_p_ioctl=ipn_kmpegts_ioctl
};

static int kmpegts_init(void) {
  return ipn_proto_register(IPN_MPEGTS,&mpegts_proto);
}

static void kmpegts_exit(void) {
  ipn_proto_deregister(IPN_MPEGTS);
}

module_init(kmpegts_init);
module_exit(kmpegts_exit);
```

Figure 6.2: MPEG TS policy submodule: mpegts_main.c

```
  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif

clean:
  rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
```

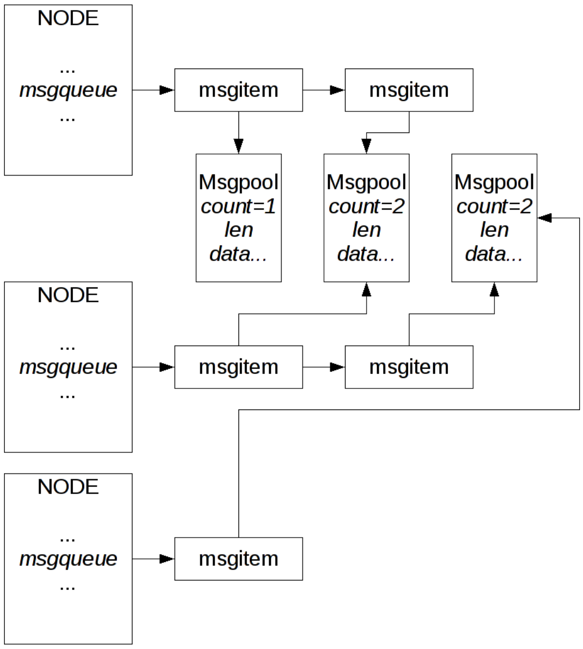

Figure 6.3: IPN: Data stuctures used for message queuing

6.6 ♦IPN internals

IPN has been designed as a fast modular multicast service for inter process communication. IPN code aims to use the parallelism of the architecture where it runs. The critical sections have been minimized, some receivers can dequeue packets while the sender is already queing the same messages for other processes. All the data structures use slabs for dynamic allocation.

The main part of the code is in `af_ipn.c`. `ipn_netdev.c` contains the code for tap and grabbed interfaces and `ipn_msgbuf.c` manages different slubs for messages data (struct msgpool_item) one for each MTU of networks.

The main data structures of IPN are:

- `ipn_node`. Each endpoint of communication is an IPN node. Each node has its msglog, a spinlock just for the time to queue/unqueue a message in its msgqueues. There is a queue for messages and one for oob messages. The `pbp` (pre bind parameters) field, temporarily stores some configuration to configure the network when bind creates or joins it.

- `ipn_sock` has been defined for compatibility with the socket implementation interface of the Linux kernel. The node struture is not part of the `ipn_sock` definition as nodes can persist without any socket connected to it.

- `ipn_network` This is the core structure of the module. Each instance of this data structure keeps the data of an IPN network. There is a hash table to find an `ipn_network` from its name (the pathname). Each `ipn_network` has two queues of nodes, connected nodes and unconnected nodes. There is also an array (`connport`) to access directly nodes given the port number, this array is dynamycally created and has the size of `maxports`. `msgpool_size` is the maximum of pending messages per node in case of lossy services or the global maximum number of pending messages in case for losslessy netowrks.

In the lossy service (provided by default) messages are not queued for a receiver if its queue is already full. On the contrary for the lossless service there is a limit on the global number of pending messages: when the sender tries to allocate a message it blocks (on `send_wait`) until some messages get received and deallocated. The global number of pending message is limited also for the lossy service, in fact when all the receivers have full queues, all the new messages cannot be sent to anybody thus immediately deallocated.

When a process sends a message on an IPN, the message is stored in a msgpool buffer (waiting for one if the service is lossless and the buffer is fulle), the reference count of the msgpool is set to one, and it is passed to the protocol submodule (broadcast protocol is considered as an "internal defined" submodule). The module's handlemessage function calls `ipn_proto_sendmsg` for each receipient of the message. `ipn_proto_sendmsg` create a msgitem element, increases the number of reference count for the msgpool, and enqueue the element for the receipient using the spinlock to prevent wrong interactions with a concurrent dequeue operation. The msgpool reference count is descreased when a destination process receives the message and when `ipn_proto_sendmsg` completes to enqueue a message. Reference count is an atomic counter: when the counter decreases to zero the msgpool item gets deallocated.

All the send and receive operation do not involve any search loop in the IPN code, the only spinlock involved is the critical section between enqueue and dequeue of a message for a specific destination. No synchronizations take place between different destination. Core IPN code is O(1) for sending and receiving. The submodules may scan the node array to forward the message to several destinations. For example the standard broadcast module forwards the message to all the nodes but the sender. In this case the complexity is linear on the number of connected nodes.

The file `ipn_netdev.c` include at the beginning the management of tap interfaces directly connected to an IPN network (callback functions for the `net_device` structure of the linux kernel. Please note that the `ipn_net_xmit` function gets called when a packet is sent by a process through a tap interface, thus it is an input event for ipn. The packets received from a grabbed interface instead activate the `ipn_handle_hook` function. Both `ipn_net_xmit` and `ipn_handle_hook` allocate a msgpool item, call `ipn_proto_injectmsg` to inject the message in the IPN network and call `ipn_msgpool_put` to deallocate the msgpool item if and when all the destination processes have received the message.

The interface from the IPN core to `ipn_netdev.c` is composed by four functions: `ipn_netdev_alloc`, `ipn_netdev_activate`, `ipn_netdev_close` and `ipn_netdev_sendmsg`. The startup phase of an IPN tap o grabbed interface invonves two phases: allocation and activation. During the allocation phase, some consistency checks take place (e.g. loopback interfaces cannot be grabbed) and data structure gets allocated. After the allocation phase, IPN core sets up the new IPN node and finally it activates the IPN netdevice. Data starts flowing through the tap or grabbed device only after the activation phase. This two step actiocation prevent protocol modules from receiving packets from already unknown interfaces.

6.7 IPN applications

IPN is able to give a unifying solution to several problems and creates new opportunities for applications. Several existing tools can be implemented using IPN sockets:

- VDE: Level 2 service implements a VDE switch in the kernel, providing a considerable speedup
- Tap (tuntap) networking for virtual machines
- Kernel ethernet bridge
- All the applications that need multicasting of data streams, like tee or jack

A continuous stream of data (like audio/video/midi/...) can be sent on an IPN network and several applications can receive the broadcast simply by joining the channel.

6.8 ▼IPN in education

The main advantage of IPN is that it permits to use broadcast/multicast in inter process communication, thus it is possible to make experiments on uncommon messaging pattern.

- design and implement a publish and subscribe service.
- create a submodule for MIDI events dispatching, so that IPN can work as a virtual pactchbay for MIDI players. The user can decide which channels are routed to each player. This project requires students to write kernel code, so it is warmly suggested to test the module on a virtual machine to avoid kernel panics and to use an unpriveleged user account.

Part III

View-OS

Chapter 7

View-OS

So far, in the context of a computer system, the concept of *global view* has always been applied: every process shares the same uniform vision of the system (network, devices, file system, ...) where it is running.

Nowadays modern operating systems provide several tools that give some kind of specific partial virtuality, like *chroot*, *fakeroot*, or `/dev/tty` (viewed as a different device depending on the controlling terminal), or `/proc/self` that act partially modifying the view that a process has of the operating system. A wider application of this partial virtuality is not appliable with these tools, because they have been created for specific needs.

We can say that *every process has a view of the environment* it is running in, that defines:

- filesystem namespace, and the related ownership and permission information
- networking configuration
- system name
- current time
- devices
- other global information

Considering the views of two processes running on two different computers, the same pathname generally addresses different files (like /etc/hostname), they have a different perception of the network (they would use different IP addresses), and different perception of resource ownership and permission.

On the contrary, two processes running on the same computer have an implicit *global view assumption* of the system: the same pathname means the same file, all the processes share the same network stack, they see and use the same IP addresses, the same filesystem, the same routing policies and so on.

However, it is possible to give completely different views to processes running on the same physical computer by running them inside virtual machines. For example, System Virtual Machines as User-Mode Linux, Qemu or GXemul can be used to run processes in completely different environments.

Unfortunately, this operation is extremely expensive in terms of time and resources: an entire operating system kernel must be loaded and booted, a large chunk of the main memory must be allocated to emulate the memory of the Virtual Machine, the VM needs a disk image on the file system, and so on.

7.1 The global view assumption and its drawbacks

The global view assumption is almost always true in the POSIX standard, but it has several drawbacks. There is a strict boundary between the system calls used to change the global view, which are restricted for system adminitration use, and the other "non-dangerous" calls. These restrictions are often due to the global view assumption rather than to a real security concern.

It is the case of the `mount` system call. `mount` changes the single global mount table, thus it changes the view on the file systems for all the processes running in the system. Mount is then an extremely dangerous call that can modify in an unexpected way the running environment of all the processes, possibly leading some to abnormal termination. At the same time, a malicious use of `mount` can override security sensible information in the system (say `/etc/passwd` and create security breaches in the system.

Since `mount` is too restrictive for many applications, several exceptions have been added. Users can be given the ability to mount the contents of their own removable devices (CD, DVD, USB-keys) by using the *user* options in the `/etc/fstab` file. Users can mount their removable devices, but only if the system administratr decided so and set the proper permissions. This way, the user's removable devices become part of the global view of the file system. Thus, users have also to give convenient permissions to their files to avoid possible disclosure of sensible data.

However, we can safely state that when a user owns a file containing a disk image or she inserts some removable media, the mount operation is no more than a sophisticated access to her own personal data. It is possible to mount a file system *inside* a virtual machine without any security issue, while it's not possible to mount the same image or removable media *outside* a virtual machine because of the *global view assumption*, but this appears to be a lack in operating system design.

Networking support is prone to similar problems. A user can define her own Virtual Private Network using a Virtual machine. As in the `mount` case, the *global view assumption* on networking prevents users to define the same VPN outside a Virtual Machine. Global IP addressing and a unique shared network stack provided by the kernel are other aspects of this problem.

Exceptions and workarounds have been added to overcome the restrictions of the global view in networking as well as in file system. Consider the case of a system running both an ftp server and a webserver: if the two processes had different IP addresses it would be simpler to migrate the services independently on other systems. It is possible to create such a configuration by setting IP source parameters in the configuration files, but if the two addresses needed different routing it would be necessary to create policy routing tables and maybe custom `iptables` filters.

The point is that both addresses are in the global view of both servers, thus the above configuration effort is needed for the servers to avoid using the wrong address.

7.2 The ViewOS idea

View-OS aims to free the processes from this assumption. The idea is that each process should be allowed to have its own view on the running *environment*: this doesn't mean that processes must live in *completely* different environments, but it can be useful to keep in their view a subset of the of the real system (maybe empty, maybe the major part of it), while part of it is virtual and different for every process.

In particular, a View-OS process can redefine each system call behavior and define new system calls while keeping the original syntax. This way it's possible to run existing executables in different scenarios, possibly enhancing their features. Finally, since View-OS is part of the Virtual Square Framework, it shares all the V^2 design guidelines like modularity and user-mode implementation.

The concept of View-OS is strictly based on the idea of Virtual Machine, and should be seen as a configurable, modular and general purpose Process Virtual Machine.

View-OS support can be provided by a View-OS tailored kernel: this implementation would resemble the Kernel Virtual Server model. The kernel mantains information about the view of each process and evaluate the system calls in the context of the view. A Kernel Virtual Server (like OpenVZ) could act in the same way, keeping the mapping between each process and the VM they belongs to, and evaluating each call in the correct context.

View-OS can also be implemented at user-level using a System Call VM. A virtualizing software can give the processes different views by intercepting the system calls generated by each process. Each system call will be evaluated in the context of the calling process view.

Since User Mode Linux is the canonical SCVM, it is interesting to compare this approach to the ViewOS idea with User Mode Linux. User-Mode Linux loads an entire kernel used as virtualizing software and gives to all the processes running in the VM the same global view, hiding completely the pre-existing running environment (seen by the processes outside the VM). On the contrary, View-OS is implemented as a System Call virtual machine that can load just the modules needed for the view change. This way it is able to provide several views to different processes, and the unchanged parts of the running environment can be shared with the other processes running outside the VM. All the system call interposition methods would be regarded as View-OS "applications".

A System Call virtual machine implementation of View-OS has poorer performance than a kernel implementation. However, a System Call virtual machine can run at user-level with user permissions, and doesn't need a complete reconfiguration of the hosting operating system in order to run. Moreover, a SCVM implementation doesn't require the expensive debugging that a kernel implementation requires to be effective.

On the contrary, a System Call Virtual Machine implementation is effective on existing systems, and allows a safe testing of the ViewOS principles.

7.3 Goals and applications

The reasons that guided the creation of the ViewOS project are based on this concept: adding possibilities and flexibility on "what can be done" and "who can do it", and are better elaborated in the following points.

- **System call redefinition:** the most general way to change the system view for a process is to modify the semantics of the system calls being invoked. ViewOS allows to entirely redefine a system call behaviour and apply the new results to a process or a process tree.

- **New system calls:** In addition to "old" system calls, it is possible to define new calls for several reasons: it may be necessary for processes to interact with new features provided by the virtualization layer, or a user may want to use ViewOS to achieve compatibility with software that works for other architectures, that use different sets of system calls. The same mechanism used for system call redefinition can be used to create new system calls.

- **Binary compatibility:** A very important objective of the project is to achieve binary compatibility with existing software, so it wouldn't be necessary to recompile a program in order to use it within ViewOS, but the virtualization must be as transparent as possible. This is achieved keeping the system call syntax intact.

- **Non priviledged use:** ViewOS aims to overcome the traditional limit imposed in Unix-like system about the privileges granted to users. With ViewOS many operations (e.g. mount a file system from a disk) can be performed from a user in its own view, without affecting stability and security of the system. ViewOS doesn't run privileged code: in the mount case, the only requirement is that the user has the correct rights to read the file that contains the image.

- **Modularity and componibility:** As already shown, there are partial solution to solve specific problems, but ViewOS aims to be a generic and extensible object. ViewOS is composed of a central core and several modules, each implementing a specific kind of virtualization.

ViewOS is appliable in several contexts and cases, of which nowadays there is no good solution, or no best solution. As already noted, in many cases the use of complete virtual machines like User Mode Linux can solve the same problems, but a very high cost in terms of waste of resources, speed and flexibility. Some of the main application of ViewOS are:

- **Emulation of privileged operations:** the Unix authorization model provides a strict distinction between the superuser and normal users. In a traditional system the superuser can perform operations that a normal user can't, like mounting filesystems, configuring networking interfaces, and so on. ViewOS allows non-privileged users to perform such operations, without breaking the specific rules and permissions for the user.

- **Security:** the isolation of a server (or other) process is a known issue, in order to avoid access to protected data. The common solution is the use of a *chroot cage*, but aside the waste of resources, this method has several drawbacks: the process can't access other processes information, the use of some network interfaces is not allowed, and these operations can be performed only by the superuser. ViewOS can redefine the view of a process, hiding some portion of the system, while keeping others. Another interesting application regarding security is that ViewOS can test and run an unknown and potentially dangerous software in an isolated and safe environment, protecting the critical parts of the system.
- **Mobility:** with IPv6 it is possible to assign an IP address to *every process*, other than to the entire host computer. This, combined with the use of tools for VPN creation or other virtual networks, can make the process independent from the underlying network configuration. This way it is possible to stop the process and continue it in another moment and another place, keeping his view of the network intact.
- **Prototyping:** one of the reasons behind the creation of User Mode Linux was the necessity to test kernel changes in a quick way, without need to use a real computer or a complete virtualization (Qemu or VMWare). With ViewOS it is possible to extend this concept to network services prototyping: a user can define a set of virtual networks and virtual views that allow the processes to communicate, be them in the same machine or different machines.

7.4 ViewOS implementation

In the next section, the tools that actually realized the ViewOS concepts will be described in detail:

- **pure_libc:** this is not an implementation of ViewOS itself, but a support library needed to implement an effective system call overriding, because the current implementation of the standard GNU lib C does not allow it.
- **UMView:** this is a ViewOS implementation as a System Call Virtual Machine, more specifically a Partial Virtual Machine. UMView is implemented entirely in user space and doesn't require any modification to the running kernel. It is based on the *ptrace()* mechanism to trace the system calls of a process.
- **KMView:** like the former, this is a ViewOS implementation as a System Call Virtual Machine, more specifically a Partial Virtual Machine. Unlike the former, it depends on a Linux kernel module and on the *utrace()* process tracking mechanism.

Since UMView and KMView share the same principles, concepts and modules, but differ only in the above implementation mechanisms, they can be grouped together in the term *MView.

CHAPTER 8

Pure_libc

The standard GNU C library is one of the most complex libraries included in GNU-linux distributions: this core library is an unified container for several sub-libraries like `stdio, threads, resolv, libio` and many others. The GNU C library also includes all the functions that interface to system calls: those interfaces are wrappers in order to give the system calls the same syntax as ordinary C functions.

GNU libC has not been designed for system call overriding: it is not possible for a process to redefine its system calls. In fact, while it is possible to redefine the system call interfacing functions, the calls generated by other sub-libraries included in the GLIBC would keep using the GLIBC implementation of the system call. For example, if a program defines a function named `write` or a pre-loaded library defines such a function, all the reference to `write` system call will be diverted to the new `write` function. The system call `printf` has a call to `write` when its buffer is flushed: with the current implementation of GLIBC, a call to `printf` will keep using the old `write` system call instead of the new `write`.

`pure_libc` is not an alternative implementation of the standard library, but it's designed to be an add-on to GLIBC: `pure_libc` overrides the functions using system calls, either directly (wrapping functions) or indirectly (like in the `printf` example).

8.1 Pure_Libc API

The interface to the library is very compact (see Fig. 8.1), `_pure_start` is the only function call needed.

It has the following arguments:

_pure_syscall: it is the pointer of the system call management functioni, when a process invokes a system call (direct or indirect), `pure_libc` invokes this function. The first argument of `_pure_syscall` is the system

```
typedef long int (*sfun)(long int __sysno, ...);

#define PUREFLAG_STDIN (1<<STDIN_FILENO)
#define PUREFLAG_STDOUT (1<<STDOUT_FILENO)
#define PUREFLAG_STDERR (1<<STDERR_FILENO)
#define PUREFLAG_STDALL (PUREFLAG_STDIN|PUREFLAG_STDOUT|PUREFLAG_STDERR)

sfun _pure_start(sfun pure_syscall,sfun pure_socketcall,int flags);

long _pure_debug_printf(const char *format, ...);
```

Figure 8.1: PURE_LIBC API (pure_libc.h)

call number as defined in `unistd.h`, while the following argument are the system call arguments.

_pure_socketcall this is only a performance shortcut: the Linux kernel uses one system call `__NR_socketcall` for every socket call, when compiled for some common architectures (like i386, ppc, ppc64, while separate calls are used in x86_64). `__NR_socketcall`, when defined, has two arguments: the number of the socket call (as defined in `/usr/include/linux/net.h`) and a pointer to an array with the real call arguments. `pure_libc` calls `_pure_syscall` for *all the architectures*. texttt_pure_socketcall is the address of a socketcall management function. If `_pure_socketcall` is NULL, the socket calls are converted into system calls and can be captured as system calls (unfortunately all the system calls like `socket`, `accept`, `connect`, etc. will be received as generic `__NR_socketcall` calls on some architectures), `_pure_socketcall` redefinition is more efficient because it avoids the processing on the arguments. When defined `_pure_socketcall` is used on all architectures, for a better portability of programs.

flags: Standard files are already open when purelibc starts, thus PureLibc needs to reopen them to trace all the system call on the three standard files. There are many uses of PureLibc where the virtualization of standard files is inconvenient. So it is up to the user to decide using this `flag` argument which standard file should be virtualized. The constants named `PUREFLAGS_STD{IN,OUT,ERR}` are three flags that can be set to reopen the respective standard file.

The return value of `_pure_start` is a pointer that can be used to call the native system call provided by the kernel. Without this function, since the system call wrapping functions are redefined by `pure_libc` (including the generic `syscall` function), there would be no way to run the system calls provided by the kernel. **_pure_debug_printf** is not part of the interface, but it's just an utility function for debugging: it bypasses `pure_libc` and writes on `stderr`. This function avoids loops to debug printf inside the management functions.

With `pure_libc` it is possible to write applications that need to process all the system calls generated during their execution. It is also possible to use pre-compiled libraries and shared objects. UMview uses `pure_libc` to provide

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdlib.h>
#include <purelibc.h>

static sfun _native_syscall;

static char buf[128];
static long int mysc(long int sysno, ...){
  va_list ap;
  long int a1,a2,a3,a4,a5,a6;
  va_start (ap, sysno);
  snprintf(buf,128,"SC=%d\n",sysno);
  _native_syscall(__NR_write,2,buf,strlen(buf));
  a1=va_arg(ap,long int);
  a2=va_arg(ap,long int);
  a3=va_arg(ap,long int);
  a4=va_arg(ap,long int);
  a5=va_arg(ap,long int);
  a6=va_arg(ap,long int);
  va_end(ap);
  return _native_syscall(sysno,a1,a2,a3,a4,a5,a6);
}

main() {
  int c;
  _native_syscall=_pure_start(mysc,NULL,PUREFLAG_STDALL);
  while ((c=getchar()) != EOF)
    putchar(c);
  printf("hello world\n");
}
```

Figure 8.2: A first PureLibc example.

nested virtuality: UMview does not depend on `pure_libc`, but it checks for its exitence and loads `pure_libc` dinamically when possible.

PureLibc captures also all the calls to the generic system call interface `syscall(2)`.

8.2 Pure_Libc Tutorial

The program in Fig. 8.2 prints the number of the system call before actually calling it (it is a `catr` like stdin to stdout copy, when EOF is sent it prints `hello world`):

To run this example just compile it and link it together with the library in this way:

```
$ gcc -o puretest puretest.c -lpurelibc
```

if the purelibc library is installed in /usr/local/lib this directory must be added to the linker search path:

```
$ setenv LD_LIBRARY_PATH /usr/local/lib
```

Unfortunately purelibc does not work when loaded as a dynamic library by dlopen.

The example in Fig.8.3 solves the problem. More specifically:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <purelibc.h>

static sfun _native_syscall;

static char buf[128];
static long int mysc(long int sysno, ...){
```

... the same as in Fig.8.2.

```
}

main(int argc,char *argv[]) {
  int c;
  sfun (*_pure_start_p)();
  void *handle;
  /* does pure_libc exist ? */
  if ((_pure_start_p=dlsym(RTLD_DEFAULT,"_pure_start")) == NULL &&
      (handle=dlopen("libpurelibc.so",RTLD_LAZY))!=NULL) {
    char *path;
    dlclose(handle);
    /* get the executable from /proc */
    asprintf(&path,"/proc/%d/exe",getpid());
    /* preload the pure_libc library */
    setenv("LD_PRELOAD","libpurelibc.so",1);
    printf("pure_libc dynamically loaded, exec again\n");
    /* reload the executable */
    execv(path,argv);
    /* useless cleanup */
    free(path);
  }
  if ((_pure_start_p=dlsym(RTLD_DEFAULT,"_pure_start")) != NULL) {
    printf("pure_libc library found: syscall tracing allowed\n");
    _native_syscall=_pure_start_p(mysc,NULL,PUREFLAG_STDALL);
  }
  while ((c=getchar()) != EOF)
    putchar(c);
  printf("hello world\n");
}
```

Figure 8.3: A PureLibc example using dynamic loading

- It is possible to use purelibc to track the calling process and all the dynamic libraries loaded at run time.
- The code does not depend on purelibc. If you run it un a host without purelibc, it will not be able to track its system calls but it works.

To run this example just compile it and link it with the dl library in this way:

```
$ gcc -o puretest2 puretest2.c -ldl
```

PureLibc can be used to implement virtualization. View-OS uses it to virtualize the system calls generated by the modules and the libraries used by the modules. In this way *mview (the programs that actually implement View-OS) have an efficient implementation of module nesting.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdlib.h>
#include <purelibc.h>
#include <dlfcn.h>

static sfun _native_syscall;
static char hosts[]="/etc/hosts";

static char buf[128];
static long int mysc(long int sysno, ...){
  va_list ap;
  long int a1,a2,a3,a4,a5,a6;
  va_start (ap, sysno);
  a1=va_arg(ap,long int);
  a2=va_arg(ap,long int);
  a3=va_arg(ap,long int);
  a4=va_arg(ap,long int);
  a5=va_arg(ap,long int);
  a6=va_arg(ap,long int);
  va_end(ap);
  if (sysno == __NR_open) {
    char *path=(char *)a1;
    if (a1 && strcmp(path,"/etc/passwd")==0)
      a1=(long int) hosts;
  }
  return _native_syscall(sysno,a1,a2,a3,a4,a5,a6);
}

void
__attribute ((constructor))
init_test (void)
{
  _native_syscall=_pure_start(mysc,NULL,PUREFLAG_STDALL);
}
```

Figure 8.4: A simple virtualization based on PureLibC

Figure 8.4 shows a simple virtualization based on PureLibc This source code virtualizes the file /etc/passwd, when loaded the file /etc/hosts will be given instead of the /etc/passwd. This is the file xchange.c

This source can be compiled in this way:

```
gcc -shared  -o xchange.so xchange.c
```

This virtualizer can be tested by preloading it:

```
export LD_PRELOAD=libpurelibc.so:/tmp/xchange.so
```

Please change /tmp with the absolute path of the shared library of the virtualizer.

After the `LD_PRELOAD` the shell works as usual but when a process tries to open `/etc/passwd` it gets `/etc/hosts` instead. This behavior can be tested with commands like `cat /etc/passwd` or `vi /etc/passwd`. Note that `cat < /etc/passwd` prints the real file as it is the shell to open the file, in a subshell the virtualization applies also to this file opened by redirection.

8.3 ♦Pure_Libc design choices

Pure_Libc concepts can be implemented as a complete library, alternative to glibc, a patch for glibc or as a library based on glibc which redefines some of glibc functions.

Virtual Square team decided for the latter implementation. In fact the implementation of a complete C library is a considerable effort (on the contrary the entire Pure_Libc is currently less than 2500 lines of C code, headers included). Furthermore a different implementation of the C library creates compatibility issues and even a patch to the existing glibc requires the management of a branch that need to include all the updates of the main project and to cope with all the details of the different architecures.

Pure_Libc is based on the interface of glibc. Several functions have been redefefined but in no cases Pure_Libc functions use glibc internal variables. In this way Pure_Libc is easily portable, is highly compatible with the existing applications and libraries and does not require a daily alignment with the code of glibc.

Pure_Libc is in this way a very efficient virtualization mean as it uses almost all the optimization already implemented in glibc. On the other hand, Pure_Libc cannot be used to create a sandbox for unsafe code, as its virtualization is easily circunventable and a fatal error in the system call implementation closes the entire application (application and virtualization code share the same address space and privileges).

Pure_Libc should be completely transparent to applications, but bugs, already unimplemented features, or some open issues (like the `freopen` implementation problem, discussed in the next section) can lead to unpredicted results.

Pure_Libc has been proved to be a stable and effective support for *Mview nested modules virtualization.

glibc could provide a virtualization feature like the one provided by Pure_Libc but it does not. If eventually glibc decides to provide a way to virtualize system calls generated by glibc itself and by the other libraries, purelibc will have no more reasons to exist. It is not too easy to integrate purelibc into glibc as all the glibc functions are tightly linked to the syscall-send-to-the-kernel function. This latter function is implemented in assembler and it is architecture dependent.

8.4 ♦Pure_Libc internals

Pure_Libc redefines some of the calls of glibc. Several Pure_Libc functions are just system call and socket call interfaces (files `syscalls.c` and `socketcalls.c`).

Unfortunately this is not enough to *purify* glibc. In fact, when a function in glibc requires a system call, e.g. when `printf` calls `write`, the syscall implementation internal to glibc is directly linked. Then a redefinition of all the system and socket calls captures all the direct invocations of the process. All the indirect calls cannot be capturedin this way.

Thus `dir.c` `exec.c` and `stdio.c` include implementations of many library functions which require system calls.

`dir.c` implements all the functions related to directory management like `opendir`, `readdir`, `closedir` which require the `getdirent` system call. `exec.c` include all the exec library functions but `execve` which is the sole system call of the set, used by all the others.

The most interesting module of Pure_Libc is `stdio.h`. The implementation is based on the `fopencookie` function provided by glibc. `fopencookie` allow the creation of virtual files, the return value of `fopencookie` is a `FILE *` discriptor like the one returned by `fopen`, but `fopencookie` has a struct argument to define *hook* functions to be called to read, write, seek and close the file. `fopencookie` defined files call these functions instead of the system calls.

Unfortunately the implementation of `fileno` requires to return the kernel integer descriptor of the file from the C library `FILE *` descriptor and glibc does not provide a function to access the private data set by `fopencookie` from its `FILE *` decriptor. Pure_Libc uses a hash table to keep the mapping between the two sets of descriptors.

Another tricky part of the code is the management of standard files (stdin, stdout, and stderr) and the `freopen` call. The standard files are already open when the library starts, thus they are not managed by Pure_Libc implementation of stdio based on `fopencookie`.

The code of the `_pure_start` function (file `syscall.c`) includes statement to reopen the standard file if requested. The code to redefine stdin is the following (stdout and stderr are the same but the opening mode of `fdopen` which is `w` for stdout and `a` for stderr)

```
if (flags & PUREFLAG_STDIN) {
  fdtmp=dup(fileno(stdin));
  dup2(fdtmp,STDIN_FILENO);
  stdin=fdopen(STDIN_FILENO,"r");
  if (isatty(STDIN_FILENO))
    setlinebuf(stdin);
  close(fdtmp);
}
```

`fdtmp` is defined as a dup copy of the current stdin fileno and then it is reassigned to STDIN_FILENO (closing the previous file). At this point STDIN_FILENO is a just opened file that can be opened at the C library layer using Pure_Libc's `fdopen`. The result is reassigned to stdin. When the file is a tty the line buffering mode must be forced. `fdtmp` can be closed, it was a temporary descriptor.

The `freopen` function is typically used for redirection of stdin/stdout/stderr to other files. Pure_Libc's implementation of `freopen` (file `stdio.h`) is not completely consistent with ISO standard for its return value. In fact the return value (the new stream) should be allocated at the same address of the old one, i.e. the return value should coincide with the latter argument (or NULL in case of errors).

The structure of `freopen` code is similar to the std file reopening above. The new `FILE *` decriptor returned by the fdopen call is generally different from the descriptor just closed. This does not create any trouble when redefining stdin/stdout/stderr as Pure_Libc redefines the respective variables. Pure_Libc's implementation works also for other files if programmers have reassigned the return value of `freopen` to the original descriptor in this way:

```
myfile=freopen(newpath, "rw", myfile);
if (myfile==NULL)....
```

But the following code may not work as it relies on the unchanged value of myfile pointer.

```
if (freopen(newpath, "rw", myfile) == NULL)
  ...error...
fprintf(myfile,....)
```

We are seeking for a more general solution. In any case, the use of `freopen` for other files but stdin/stdout/stderr are quite rare.

8.5 ▼Pure_Libc in education

Pure_libc tracks the system call of the calling process. It can be used for some interesting exercises.

- Write a program to provide simple process-level virtualizations: file substitution, directory hiding etc.
- Write a program similar to strace. This program intercepts and prints all the system calls called by a process.
- Write a program to track all the accesses of a program to a specific file or device. (something like a wireshark for devices).
- Design and implement a library able to *wrap* an existing library. The goal of the exercise is to use the given library for some different purpose. For example a library designed to access files or devices, like a file system implementation or a sound playing library, can be wrapped by a library which detours the requests to sockets or memory buffers.

Chapter 9

*MView

UMView and KMView are View-OS implementations as System Call Virtual Machines, more precisely they are Partial Virtual Machines. A partial virtual machine (ParVM) is a System Call Virtual Machine that provides the same set of system calls of the hosting kernel. A ParVM allows to:

- combine several ParVMs together applying one VM on the top of the other;
- define a transparent ParVM that simply forwards each system call to the kernel (or to another ParVM in the lower layer);
- add modules inside the VM: the module can redefine some of the system calls under certain conditions, while the other system calls can be forwarded to the lower layer unchanged.

UMView and KMView share the same principles and work in the same way: their common base will be detailed further in this section and they will be referred jointly with the name *MView [10, 7]. On the other hand, the main differences between them concern the implementation of the system call capturing:

- KMView depends on a Linux kernel module, while UMView is implemented entirely in user space, consequently, no changes are required to the hosting OS for UMView;
- KMView relies on the *utrace()* process tracing mechanism, while UMView uses *ptrace()*;
- KMView is faster than UMView, because the latter is implemented exclusively with user-mode code. On the other hand, there is a kernel patch that makes UMView faster, but this way administrator acces is required anyway;

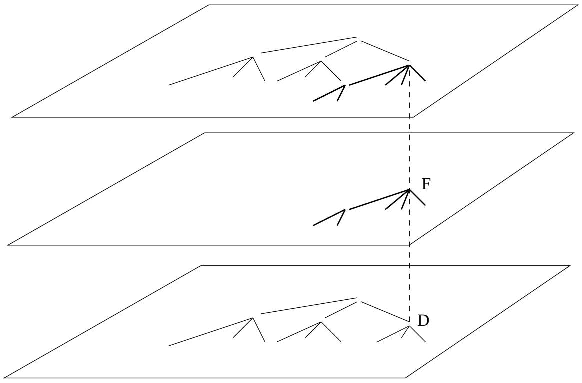

Figure 9.1: The mount metaphore

- UMView has some minor limitations on signal handling and VM nesting that KMView hasn't.

9.1 *MView file system management

UNIX is a file system centric Operating System, therefore the pathnames in the global (unique) file system are used as a global naming scheme. Special files are typical examples of pathnames used as naming entities for device drivers: Unix sockets, named pipes, and virtual files of the /proc subtree have pathnames that refer to structures in the kernel.

This file system based convention is convenient in several ways: the same set of system calls used for file I/O can be used to access devices or other kernel variables and naming schemes for virtual services can be easily created. If a process has a virtual perception of the file system structure, its view of the environment can change not only for file access but also for devices and services in general.

*MView uses the `mount` system call to add virtual subtrees to the file system, thus to the naming scheme. In fact, the *MView `mount` command has the same semantics as the standard `mount` command (Fig. 9.1): when a file system F is mounted on a directory D the root directory of F hierarchy *becomes* D, and the contents of F override the contents of D. If D is not empty, then D and all its subtree of the file system become inaccessible, hidden by F.

The *MView `mount` operation can be seen as an overlay operator: F is superimposed over D. *MView applies and extends this idea: the virtual `mount` system call is used by *MView to define a new view for processes: when a module inside *MView manages a `mount` call it creates an overlay. Usually the choice of the module depends of the *filesystemtype* type parameter.

*MView `mount` has two fundamental differences with the *kernel* `mount` system call:

- the effects of the *MView `mount` system call is limited to the partial SCVM which performed the call. After *MView has run a `mount` operation on the target directory `/mnt`, the command `ls /mnt`, if run from a shell *outside* the VM, will continue to show the previous contents of `/mnt`;

- The standard kernel `mount` system call has usage restrictions: it is generally limited for system administration use only, while the *MView `mount` can be invoked by any process.

*MView extends the idea of mount as follows:

- Each `mount` can redefine completely the process' view of the file system. This includes the redefinition of the mountpoint or the hiding of a mounted image by a further mount. Therefore, in *MView `mount` is a view-transformation function. Defining v_0 to be the view[1] provided by the operating system, then the view after the first mount m_1 is $v_1 = m_1(v_0)$.

  A second mount operation m_2 generates the view $v_2 = m_2(v_1) = m_2(m_1(v_0)) = m_2 \circ m_1(v_0)$.

  Obviously `umount` is a legal operation only when no further mounts *depend* on some pathname provided by it.

- The *MView `mount` allows to mount *files*, while the POSIX specifications limit the use to a directory as mountpoint).

- *MView allows recursive istances. From a *MView user's point of view, when a *MView machine is started from inside another *MView machine, the current view is shared by both VMs. Each one can further modify the view independently. Using the same mathematical notation introduced above: if v_n is the view of the *MView machine M, when a M starts a second machine M', the view seen by this latter machine is $v'_n = v_n$. All further mount operations will be applied independently to the two virtual machines. M's view will be $v_{n+k} = m_{n+k} \circ m_{n+k-1} \ldots \circ m_{n+1}(v_n)$ while the view of M' will be $v'_{n+h} = m'_{n+h} \circ m'_{n+h-1} \ldots \circ m'_{n+1}(v'_n)$ where v_n and v'_n coincide.

The *MView implementation is able to manage efficiently mount nesting as well as *MView nested executions. Each `mount` operation is theoretically a Partial SCVM that performs the view transformation: *mount nesting* is the composition of the correspondig Partial SCVMs (Fig. 9.2).

Finally, *MView does not spawn other processes to manage multiple mounts or recursive executions[2]. Using *purelibc*, *MView captures the system calls to run with the efficiency of a function call.

Unfortunately, the file system is not the naming mean for everything; in fact, the networking support has a separate naming space. Interfaces, stacks, protocol families have not file system entities to name them. The entire networking support has been designed as global and shared by all the processes. Without a manageable naming space the application program interface (API) provided by the libraries does not give the abstractions needed to virtualize

[1] The view can be considered as a function: it maps all the pathnames to the corresponding entities, or to an error

[2] This is not entirely true for KMView. In this case, recursive execution can also correspond to real process nesting (kmview child of another kmview, it is up to the used to decide which kind of nesting to use).

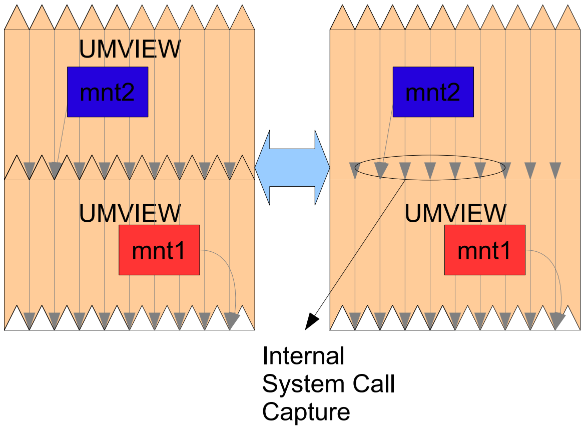

Figure 9.2: Composition of Partial Virtual Machines

networking entities (stacks, interfaces etc) or to access several stacks at the same time.

*MView supports the *msockets* extension of the Berkeley sockets interface, thus it is possible to map several stacks on the file system.

9.2 *MView modularity

*MView is divided in two parts: the core and the modules. The core takes care of the system calls interception (with *ptrace()* for UMView and *utrace()* for KMView) and, in case, deliver them to the appropriate module. Each module implements a subset of system calls, based on the virtualization service offered.

The modules are implemented in the form of dynamic libraries, loaded by the user at execution time. Each module defines a particular *choice function*, used to determine whether the module has to be used or not. This way, after the interception of a system call the core must decide whether the call has to be managed by *MView or not, and in the positive by *which* module.

Currently, several modules have been implemented by the V^2 team, that will be detailed later in this section.

9.3 ★*MView basic usage

As already noted in this chapter, UMView and KMView share the same principles, and so they share the same command syntax. The only difference is that in order to invoke UMView, the command `umview` is used, while for KMView, the command `kmview` is used instead. Options, flags and parameters are the same[3]. In the following examples `umview` will be used.

[3]Only one exception: UMView may use a kernel optimization flag that KMView doesn't need.

umview/kmview

It is used to start a new command within the partial virtual machine: the command is passed as an argument to `umview/kmview`, with its additional arguments:

```
$ umview bash
```

Once the VM is started, if there is not any module loaded, the command behaves as if it was executed outside umview: therefore, when starting a non-interactive shell within umview it may be useful to preload modules (with the `-p` option):

```
$ umview -p umfuse -p umnet bash
```

It is possible to define a system wide initialization file (`/etc/viewosrc`) and a personal initialization file ($\tilde{7}$`.viewosrc`, or the file specified by the `-rc` option). `umview/kmview` run the commands in the initialization files prior to start the execution of the virtualized command (bash in the examples above).

Usually `umview/kmview` work in *omnipotent mode*, i.e. access control is disabled: all the commands/system calls allowed by the real kernel are allowed also when executed inside the virtual machine. This eases the work for a user that need `umview/kmview` to exploit virtualizations on his/her resources. On the contrary this is useless when `umview/kmview` must guarantee security. The command flag `s` or `secure` set `umview/kmview` in secure mode, also known as *human mode*. In this state capabilities and permissions are enforced. `umview/kmview` set the uid/euid to 0 at starting, it means that the virtual machine starts the command providing it the privileges of a *virtual root*. When a process sets its uid to the one of an unprivileged user (e.g. via `viewsu` here below), it looses its capabilities: it can access/execute only those files that are accessible/executable in the virtual view for that specific user. Moreover privileged system services like mounting file systems or changing viewos modules are forbidden.

`umview/kmview` can be used as login shells, but we postpone the description of this feature the the end of the next chapter (10.18) to use viewos modules in the examples.

um_add_service

This command, that must be launched within the process run by umview, loads a specified service module to the umview chain of services. It is possible to specify the exact position of the module being loaded: this allowed to choose the order followed by the system call interception among service modules in former versions of umview. Now the service module is always determined by the overlay mounting paradigm (see 9.1). um_add_service has a option flag `-p`: when set the module becomes permanent.

```
$ um_add_service umfuse
```

um_ls_service

This command returns a list with the currently loaded umview service modules.

```
$ um_ls_service
umfuse: virtual file systems (user level FUSE)
umnet: virtual (multi-stack) networking
```

um_del_service

It removes the specified service module from the umview service chain: the module to be removed must be specified by its name as listed by um_ls_service (usually the name is consistent with the filename of the module).

```
$ um_ls_service
umfuse: virtual file systems (user level FUSE)
umnet: virtual (multi-stack) networking
$ um_del_service umfuse
```

umshutdown

This command is used to shut down the current umview/kmview virtual machine. When called without any argument, it sends a TERM signal to all the processes running in the current virtual machine, waits for 30 second and then sends a KILL signal to all the surviving processes. With an optional argument `time`, the number of seconds between the TERM and KILL signals can be specified.

```
$ umview bash
$ umshutdown
```

viewname

This command sets and shows the view name of the current *mview machine.

```
$ viewname

$ viewname myview
$ viewname
myview
```

The `-p` option is useful to create a prompt for shells. Instead of returning an empty string when the name is not set, it returns some information about the view: the hostname of the hosting operating system, the process number of the *mview hypervisor and the number of view.

```
$ viewname -p
v2host[23784:0]
$ viewname myview
$ viewname
myview
```

The `-q` option does not print any error when executing the command outside a *mview virtual machine.

It is possible to set the bash prompt to include the id of the current view.

For example a user can have in her `.bashrc` file the following command list:

```
if [ -x /usr/local/bin/viewname ]; then
  viewname=$(/usr/local/bin/viewname -pq)
fi
if [ "$viewname" == "" ]; then
  viewname=$(/bin/hostname)
fi
PS1='\u@$viewname:\w\$ '
```

In this way the bash prompt for a host named `v2host` will be:

- `alice@v2host:/$` when the user is working outside *mview;
- `alice@v2host[23784:0]:/$` when the user is working in a *mview session;
- `alice@myview:/$` when the user is working in a *mview session named `myview`.

Please note that `PS1` must be recomputed to change the prompt, in this way:

```
alice@v2host[23784:0]:/tmp$ viewname myview
alice@v2host[23784:0]:/tmp$ source ~/.bashrc
alice@myview:/tmp$
```

vuname

`vuname` is the extension of `uname(1)` for view-os. `vuname` can be used instead of uname, it has the same options of the standard unix command. `vuname -a` shows three more fields when executed inside a *mview virtual machine.

```
$ uname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 2007 i686 GNU/Linux
$ vuname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 2007 i686 GNU/Linux/View-OS 24410 0
$ viewname myview
$ vuname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 2007 i686 GNU/Linux/View-OS 24410 0 myview
$ umview bash
UMView: nested invocation

$ vuname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 2007 i686 GNU/Linux/View-OS 24410 1 myview
$ exit
$ vuname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 2007 i686 GNU/Linux/View-OS 24410 0 myview
```

The operating system name is changed from “GNU/Linux” to “GNU/Linux/View-OS”, then there is the server-id, i.e. the pid of the *mview server process, the view number, and the view name.

There are new options for the `vuname` new parameters:

- `-U, --serverid`: print the server id;
- `-V, --viewid`: print the view id;
- `-N, --viewname`: print the view name.

viewsu

`viewsu` is the view-os counterpart of the standard `su` utility. This utility starts a shell running as a different user specified as a parameter. The new user is root when started with no parameters.

```
$ viewsu bin
$ whoami
bin
$ exit
$ viewsu
# whoami
root
# exit
```

fsalias

`fsalias` defines aliases for file system types.

Example:

```
$ um_fsalias ext2 umfuseext2
$ um_fsalias ext3 umfuseext2
$ um_fsalias ext4 umfuseext2
```

After these commands it is possible to use:

```
mount -t ext2 -o ro /tmp/diskimage /mnt
```

which is simpler and more natural then using `-t umfuseext2`.

9.4 ▲*MView modules

This section provides a guide for writing *MView modules.

A module is a shared library. It must define a permanent variable of type `struct module`. Normally it is a global (maybe static) variable, but it could also be a static variable of the constructor.

The source code of the simplest (useless) *MView modules is in Figure 9.3. Each module must define a global variable (of type *struct service*) named `viewos_service`. Alternatively it is possible to give the variable a different name (`s` in the example) and redefine it through the macro `VIEWOS_SERVICE`. The constructor defines some variables of the service structure.

`printk` is equivalent to `fprintf(stderr,...)`, but it cannot be further processed by other modules so the error output is faster and cannot be hidden by erroneous managements.

Each system call generated by a process running in a *MView machine must be associated to a module for its managemnt, or sent to the kernel.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "module.h"

static struct service s;
VIEWOS_SERVICE(s)

static void
__attribute__ ((constructor))
init (void)
{
  printk("simplemodule init (hello world)\n");
  s.name="hello";
  s.description="Hello world module";
  s.syscall=(sysfun *)calloc(scmap_scmapsize,sizeof(sysfun));
  s.socket=(sysfun *)calloc(scmap_sockmapsize,sizeof(sysfun));
}

static void
__attribute__ ((destructor))
fini (void)
{
  printk("simplemodule fini\n");
}
```

Figure 9.3: A simple "hello world" *MView module

The umview or kmview core system (or hypervisor, or virtual machine monitor VMM) works as a dispatcher for system calls. Each module must *register* its *working domain*, i.e. the subtree of the file system, the system calls, address families it defines. Modules use `ht_tab_add` and `ht_tab_pathadd` to add generic objects or pathnames respectively.

There is no such definition for file descriptors. In fact the *MView hypervisor assigns all the system calls based on file descriptors (like `read` or `fchmod`) to the same module which generated the fd (by an `open`, `socket` or `msocket` call, for example).

Type can have the following values:

CHECKPATH arg is a pathname, this check is for all the system calls with a path like `open(2)` or `access(2)`;

CHECKSOCKET for `socket(2)` arg is an address family;

CHECKSC the arg is a system call number for system calls like `getuid`, that have neither pathname nor other means of partialization.

CHECKBINFMT for `execve(2)` the arg is the pointer to a sctructure including the pathname (input), the interpreter, one extra arg for the interpreter, and a flag field (out).

CHECKIOCTLPARMS this is not called to decide if this is the module for `ioctl(2)` but just to define the type and length of the argument for an ioctl call (see over).

There are two macros in `module.h` to define implementation functions for system calls.

```
SERVICESYSCALL(struct module s,syscall name, implementation function)
SERVICESOCKET(struct module s,syscall name, implementation function)
```

Ususally these macros are used in the module's contructor. All the implementation functions have the same syntax of the original system calls.

*Mview hypervisor reduces the number of syscall to implement by using equivalent one when possible. Table 9.1 is a summary of the subsitution rules used by *MView.

All the pathnames provided by the hypervisor are absolute (canonicalized). They always refer to the file or directory, target of the system call. It is possible in this way to unify the management of several system calls. For example the implementation of lchown in a module provides the support for handling all the changing ownership system calls: chown, lchown, fchown and fchownat. In fact fchownat does not use the current directory argument, as the pathname is absolute. chown and lchown behavior is different on symbolic links, chown changes the ownership of the linked file while lchown operates on the link itself. The hypervisor traverses the link if the process used chown while does not traverse the link if lchown was called. lchown implementation in the module will operate on the right file in both cases. fchown changes the ownership of an open file. The hypervisor tracks all the descriptors of open files and traslates fchown into lchown by putting the pathname instead of the descriptor number. In the same way all the system calls having l- f- prefixes and -at suffixes can be unified to one single l- implementation in the module.

This idea descreases the number of system call implementations provided by each module. The flip side of this simplification could be a worst performance to handle f- prefixed calls. In fact, the module could already have information on open files in local data structures thus retrieving the same information from the pathname is a waste of time. For this reason *mview calls the modules' implementations of lstat64, lchown, lchmod, getxattr, setxattr, listxattr, removexattr with an extra trailing integer argument which is the file descriptor for the f- prefixed calls, -1 otherwise. Each module can decide to use this feature or not. Following the example of the previous paragraph, the implementation function of the module for lchown can be defined in the standard way:

```
int lchown(const char *path, uid_t owner, gid_t group);
```

and the module uses the pathname to change the ownership of the file. If lchown implementation is defined:

```
int lchown(const char *path, uid_t owner, gid_t group, int fd);
```

the implementation may use the fd passed by the user process in its fchown call.

Please note that also the implementation functions for `recv`, `revfrom`, `send`, `sendto` are getting deprecated in favour of `recvmsg` and `sendmsg`. In a future version modules will be required to implement only `recvmsg` and `sendmsg`.

substituted	substitute	comments
creat	open	creat is open with `O_CREAT|O_WRONLY|O_TRUNC`
readv writev preadv pwritev	read write pread pwrite	A temporary buffer is used anyway
time	gettimeofday	time can be implemented by gettimeofday
setpgrp getpgrp	setpgid getpgid	pgid is more complete
umount	umount2	umount(f) is umount2(f,0)
stat fstat chown fchown xxx fxxx	lstat lstat lchown lchown lxxx lxxx	the hypervisor manages the link traversal for non-l calls and provide the path for the f- calls
openat mkdirat mknodat fchownat futimesat unlinkat renameat linkat ...at statat64	open mkdir mknode lchown utimes unlink rename link ... lstat64	modules always receive absolute pathnames
lstat getdents truncate statfs	lstat64 getdents64 truncate64 statfs64	for 32 bits machines, modules should implement the 64 bits calls. If a process uses the 32 bits syscalls results will be trunced
getuid getgid ...id	getuid32 getgid32 ...id32	All the old calls with 16 bits uid/gid (where present) are supported through the 32 bit versions. In modules the syscall names are without the trailing 32, but all uids/gids are 32bits wide.
dup dup2 chdir fchdir chroot mmap munmap mremap		These system calls are processed by the hypervisor, never forwarded to modules
select poll pselect ppoll		These system calls are processed by the event_subscribe function (see text)
fcntl		*MView handes some tags (like F_DUPFD,F_{GET,SET}FD) internally, lock is unsupported. Can be redefined for other module defined tags
llseek	lseek	If llseek is not defined lseek is used instead
msocket socket	socket msocket	if msocket is defined it is used for all the system calls,otherwise socket is used. In this latter case socket is used for msocket with NULL path

Table 9.1: System call substitution rules for *MView modules

When a implementation function emulate a system call which returns a file descriptor (like `open` or `msocket`), the return value is simply an integer, it is not required to be a real file descriptor, nor that for the number to be unique across different modules. This number is also unrelated with the file descriptor as seen by the process. *MView keep track of the matching and uses the integer returned by these implementation function as the file identifier in all the successive calls involving file descriptors (e.g. `write`).

`select`, `poll`, `pselect` and `ppoll` system calls need a specific support. These system calls wait for events on several file descriptors that could be managed by different modules. *MView use an event subscribe function to manage these system calls:

```
static long module_event_subscribe(void (* cb)(), void *arg, int fd, int how)
...
static void __attribute__ ((constructor))
init (void)
{
  ....
  s.event_subscribe=module_event_subscribe;
}
```

When a process uses one of the event waiting system call, *MView calls the event_subscribe function. `how` is the desired event, encoded as explained in `poll(2)`. If some of the events the process is waiting for already occured, event_subscribe returns the bit mask of aoccured events (a subset of `how`). If `cb` is non null and no events occured, the module must keep note of the request and call back the function, using the argument `arg`. If `cb` is NULL, any pending request for having the same `arg` should be deleted.

If a module open files, it can use the same inteface to wait for events. Note that when a module opens file, these files could be implemented by other modules or even by the same module. For this feature the support interface for modules provide the following function:

```
int um_mod_event_subscribe(void (* cb)(), void *arg, int fd, int how);
```

There are several utility functions for modules:

```
extern int um_mod_getpid(void);
extern int um_mod_umoven(long addr, int len, void *_laddr);
extern int um_mod_umovestr(long addr, int len, void *_laddr);
extern int um_mod_ustoren(long addr, int len, void *_laddr);
extern int um_mod_ustorestr(long addr, int len, void *_laddr);
extern int um_mod_getsyscallno(void);
extern int um_mod_getumpid(void);
extern struct stat64 *um_mod_getpathstat(void);
extern int um_mod_getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
extern int um_mod_getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
extern int um_mod_setresuid(uid_t ruid, uid_t euid, uid_t suid);
extern int um_mod_setresgid(gid_t rgid, gid_t egid, gid_t sgid);
extern int um_mod_getfs_uid_gid(uid_t *fsuid, gid_t *fsgid);
extern int um_mod_setfs_uid_gid(uid_t fsuid, gid_t fsgid);
extern int um_mod_getsyscalltype(int escno);
extern int um_mod_nrsyscalls(void);
void *um_mod_get_private_data(void);
```

um_mod_getpid returns the pid of the calling process;

um_mod_umoven, um_mod_umovestr, um_mod_ustoren, um_mod_ustorestr are used to trasfer data with the caller's memory;

um_mod_getsyscallno returns the system call number as several system calls may share the same implementation funciton;

um_mod_getumpid returns the umpid of the process, i.e. the *MView internal pid. It is a small integer that can be used as an index for an array.

um_mod_getpathstat provides the stat of the file (the ViewOS monitor has already read the stat of the current file so it keeps it available for the modules, as an optimization).

um_mod_[gs]res[ug]id reads/sets the current virtual real/effective and saved user/group.

um_mod_nrsyscalls returns the syscall requested by the user (modules can read which was the syscall in case of unification).

um_mod_get_private_data returns the private data for the hash table element that mached this system call request.

The second example (see Figure 9.4) of module changes the nodename. The effect is the following:

```
$ uname -a
Linux v2host 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 2007 i686 GNU/Linux
$ um_add_service ./abitmore
$ uname -a
Linux mymodule 2.6.22-viewos #5 SMP Tue Jul 31 22:29:46 CEST 2007 i686 GNU/Linux
```

In the source tree of View-OS there are several examples of modules are included in the `um_testmodule` directory:

real: nothing seems to change, but the file system get accesses by KMview.

unreal: all the filesystem is visible also inside the `/unreal` directory. If means that `/unreal/etc/passwd` is `/etc/passwd`. This module support nested calls with itself (two levels of virtual calls). So, `/unreal/unreal/etc/passwd` is already `/etc/passwd` but `/unreal/unreal/unreal/...` does not exist.

sockettest: is a test similar to "real", applied to sockets instead of the file system.

sockip: is sockettest limited to AF_INET sockets.

Modules often need data structures to store private data about processes or about other modules or mounted partitions. Modules in this case need notifications when there are state changes that may affect also their data stuctures. Modules can subscribe to receive these notifications by setting the variable `ctlhs` variable included in the `struct service`. `ctlhs` is a bit mask. There are specific macros in `module.h` to define `ctlhs`.

```
#define MCH_SET(c, set)   *(set) |= (1 << c)
#define MCH_CLR(c, set)   *(set) &= ~(1 << c)
#define MCH_ISSET(c, set) (*(set) & (1 << c))
#define MCH_ZERO(set)   *(set) = 0;
```

MCH_ZERO initializes the value to zero. SET and CLR are used to set a bit to one or zero respectively, and ISSET can be used to test the value of a bit.

There are three classes of event currently defined by *MView:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/utsname.h>
#include <umview/module.h>

static struct service s;
VIEWOS_SERVICE(s);
struct ht_elem *htuname;

static int my_uname(struct utsname *buf) {
  if (uname(buf) >= 0) {
    strcpy(buf->nodename,"mymodule");
    return 0;
  } else
    return -1;
}
  static void
  __attribute__ ((constructor))
init (void)
{
  int nruname=__NR_uname;
  printk("Second module (uname) init\n");
  s.name="abitmore";
  s.description="Uname Module";
  s.syscall=(sysfun *)calloc(scmap_scmapsize,sizeof(sysfun));
  s.socket=(sysfun *)calloc(scmap_sockmapsize,sizeof(sysfun));
  SERVICESYSCALL(s, uname, my_uname);
  htuname=ht_tab_add(CHECKSC,&nruname,sizeof(int),&s,NULL,NULL);
}

  static void
  __attribute__ ((destructor))
fini (void)
{
  ht_tab_del(htuname);
  printk("Second module (uname) fini\n");
}
```

Figure 9.4: Source code of abitmore.c: a module that changes the nodename

MC_PROC: events related to processes, starting and termination of processes;

MC_MODULE: loading and unloading of modules;

MC_MOUNT: mount and umount of partition or files.

`ctlhs` must be set in the contructor prior to call `add_service`.

When a state change occur *MView use the function pointer `ctl` also part of `struct service`. This function has a variable number of arguments:

```
long (*ctl)(int, va_list);
```

the first argument is a tag, a bit mask composed by the class and the type of the event. The predefined types are: `MC_ADD` for a new process, module or mounted item, `MC_REM` for the termination of a process, removal of a module or an umount. The `va_list` contains the following fields:

MC_MODULE | MC_ADD or MC_MODULE | MC_REM: `int servicecode`
  `servicecode` is the code of the loaded/unloaded module (s.code).

MC_PROC | MC_ADD: `int umpid, int pumpid, int numprocs`
  numprocs is the current max number of processes: service implementation

can use it to realloc their internal structures. UMPID is an internal id, *not* the pid! id is in the range $0, \ldots, numprocs - 1$ it is never reassigned during the life of a process, can be used as an index for internal data pumpid is the similar id for the parent process, -1 if it does not exist

MC_PROC | MC_REM: int umpid
: is the garbage collection function for the data that addproc may have created

MC_MOUNT | MC_ADD or * MC_MOUNT | MC_REM:
: these events are defined but not generated yet.

An example of use of `ctl` function for event notification is the source code `unreal.c` of the unreal module, the relevant code has been copied in Figure 9.5.

The event notification method can also be used for inter module communication. The following function:

```
#define MC_USERCTL(sercode, ctl) /* */
#define MC_USERCTL_SERCODE(x) /* */
#define MC_USERCTL_CTL(x) /* */

void service_userctl(unsigned long type, service_t sender, service_t recipient, ...);
```

can be called by module to notify events to other modules. Modules should agree on tags and arguments of notifications. Recipient can be the code of another module or UM_NONE when the notification is a broadcast for all the modules which suscribed for compatible tags.

When a new module calls `add_service`, new process and new module events are generated for all existing processes and module, so that the module can set up its data structures consistently.

9.5 ♦UMview internals

Umview implementation

The lowest layer of the architecture is contained in the source file `capture_um.c`. This layer captures all the system calls. Each system call is converted into an up-call to the upper layer when a management function has been registered for that kind of system call. The lowest layer also manages the virtual machine process table. A snapshot of the registers is made at each call, in order to have all the parameters ready. Please note that umview does not use shared memory with the processes inside the virtual machine. This approach is different from User-Mode Linux. umview uses ptrace calls (or access to process memory via `/proc/<pid>/mem` files to exchange data from/to the controlled process memories. This is the reason for the relative slowness of umview on unpatched kernels, where PTRACE_MULTI patch has not been applied. In fact, with standard kernels, umview needs to invoke a ptrace call for each memory word, which needs to be exchanged with the calling process. The PTRACE_MULTI is both the name of the patch and the name of the new tag of ptrace used to pack several data exchange operations into one single ptrace call. It is a similar idea to the readv, writev or recvmsg, sendmsg calls. These calls can do I-O

```
static long addproc(int id, int max) {
  fprintf(stderr, "add proc %d %d\n", id, max);
  return 0;
}

static long delproc(int id) {
  fprintf(stderr, "del proc %d\n", id);
  return 0;
}

static long addmodule(int code) {
  fprintf(stderr, "add module 0x%02x\n", code);
  return 0;
}

static long delmodule(int code) {
  fprintf(stderr, "del module 0x%02x\n", code);
  return 0;
}

static long ctl(int type, va_list ap)
{
  int id, ppid, max, code;
  switch(type)
  {
    case MC_PROC | MC_ADD:
      id = va_arg(ap, int);
      ppid = va_arg(ap, int);
      max = va_arg(ap, int);
      return addproc(id, max);
    case MC_PROC | MC_REM:
      id = va_arg(ap, int);
      return delproc(id);
    case MC_MODULE | MC_ADD:
      code = va_arg(ap, int);
      return addmodule(code);
    case MC_MODULE | MC_REM:
      code = va_arg(ap, int);
      return delmodule(code);
    default:
      return -1;
  }
}
`
static void __attribute__ ((constructor))
init (void)
{
  ...
  s.ctl = ctl;
  MCH_ZERO(&(s.ctlhs));
  MCH_SET(MC_PROC, &(s.ctlhs));
  MCH_SET(MC_MODULE, &(s.ctlhs));
  ...
}
```

Figure 9.5: Example of ctl event notification function.

using several buffers. UMview uses PTRACE_MULTI to limit the number of user/kernel mode switches with the hosting kernel. In this way it has a much better performance. It is worth noting that umview slows down the system call management, as all the remaining parts of the process run on the real processor without any kind of emulation or virtualization. For example all "cpu-bound" processes will receive almost no change in their performance. The file utils.c contains all the routines to copy memory areas from or to the process memories. These operations have been implemented as complex loops when there is no PTRACE_MULTI support from the kernel. One system call is needed for each

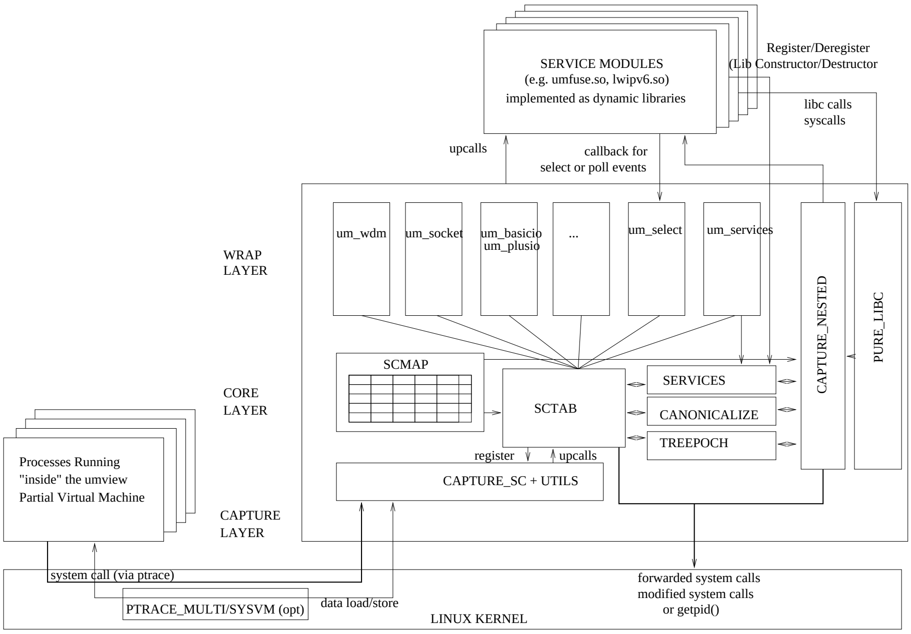

Figure 9.6: Design of UMview Implementation

memory word. Special code is needed to read/write the first or last voice when the field to be read is not aligned. On the other hand, these functions collapse to one single statement when running on a PTRACE_MULTI patched kernel. Umview code also tries to access the process memory by the `/proc/`*`pid`*`/mem` file. This can speed up reading, but current Linux kernels do not allow writing on that file.

All the tables maintained inside umview are auto-expanding: they change their size when the table is full and a new entry is needed. The process table initially consists of four elements, which then doubles its size as soon as the previous size is not sufficient for the number of processes running inside the virtual machine. Pointer arrays have been used as data handles in many parts of the code. This is specifically the case for the process table. The index within the array gets used as an identifier. Each pointer in the array contains the address of the corresponding item. This technique is useful, as there is no need to reallocate actual data structures when expanding the array. All the pointers to data do not change. Moreover, in this way it is faster to expand the array: the operation just involves four bytes per process, instead of the entire data structures.

All the system calls to create processes (like fork, vfork, clone) get converted into clone, i.e. whichever call the process called the real kernel runs a clone. Clone, in fact, has a more complete interface and permits implementation of other calls. Clone also has the CLONE_PTRACE flag: processes or threads created in this way inherit the ptrace mode, so no calls can be lost.

ptrace sends a SIGCHLD signal for each relevant event, as a termination, signal or system call. The SIGCHLD signal handler defined by umview forwards the signal through a pipe to the main event loop.

`capture_um` has a table `scdtab` where the upper layer registers at startup time the upcalls for each system call (function `scdtab_init`, file `sctab.c`). For those architecture where all the socket API is implemented with one system call (`__NR_socketcall`) there is a second table named `sockcdtab`: `capture_um` recognizes and decode socket calls. `capture_um` calls the functions stored in `scdtab` and `sockcdtab` twice, before and after the kernel syscall (IN phase and OUT phase). These upcall functions in the IN phase return the *behavior* of the call:

STD_BEHAVIOR: the kernel runs the system call in the standard way;

SC_FAKE: the kernel does not run any system call (it really skips the system call if PTRACE_VM optimization is installed, otherwise the call is converted into an effectless getpid);

SC_CALLONXIT: the kernel runs the system call but the result of the system call must be further processed: sometime the virtualization requires the kernel to run a different system call, or the same system call with different parameters.

The main event loop is at the end of umview.c source file. This loop manages the ptrace signals, as well as all the asynchronous signals coming from modules to notify some I-O relevant for unblocking processes waiting for select or poll system calls. Synchronization between the signal handler and the main loop is implemented by pselect on recent kernels, with a pipe when pselect is not supported.

The second layer of the architecture can be read in the sctab.c, scmap.c and services.c files. scmap is essentially a large table: each system call entry specifies what is the corresponding choice function, which are the wrapper functions to be invoked before and after the call takes place. These wrapper functions take their parameters from the registers and call the service module functions with the same syntax as the original system calls. Service modules in this way can be implemented in a natural way: the system call invocation by controlled process is translated into the call of an identical function to the service module. This idea is similar to remote procedure call. Remote procedure call provides services to invoke an identical function on another computer. Here the call is just translated to the service module. These wrappers consist of the third layer of the architecture. scmap also includes a separate table for the socket calls, because these are not individual system calls in Linux (for many, not all the architectures), but one single system call exists (with identifier __NR_socketcall), and the first parameter is the socket call identifier.

The file sctab.c implements the core pare of the second layer. After registering the system call to the capture layer it receives up-calls for each system call, or for any process creation or termination event. Functions named megawrap search the scmap tables to decide whether a a system call is managed by a service module or not, and call the corresponding wrapper. For the sake of precision, sctab contains one single megawrap meta-function, which is able to run for system calls or socket calls, depending on the parameters. This solution minimizes duplication of code. Each service module has a Boolean function to signal the virtual machine that wants to handle a specific pathname, filesystem type, address family, etc.

sctab extends the process descriptor with the fields needed by this second layer. First layer process descriptors include a handle for further layers to extend the data structure. All the pathnames get converted to absolute pathnames before any call. Relating to absolute path transformation is carried out in the canonicalize.c file (the code has the same goal of what is implemented in the homonymous file, canonicalize.c, of the C library. The code has been completely rewritten and optimized for the ViewOS monitor).

The sctab.c file uses the routines defined in services.c to manage all the data structures related to service modules.

The core structure used to dispatch the system calls to the modules is the global hash table (`hashtable.[ch]`).

This data structure stores all the services provided by the modules (and their sub-modules) and allow a fast and scalable way to dispatch all the system calls to the right module.

Several kinds of objects can be stored in the hash table: modules, pathnmames, address families, char/block devices, system calls, interpreters for executable.

Each object has its own hash sum which is a one word (long) integer, the hash key is the sum modulo the number of elements of the hash table.

Each object is stored in the hash table in a collision list corresponding to its hash key. The data structure associated to each object follows:

```
struct ht_elem {
  void *obj;
  char *mtabline;
  struct timestamp tst;
  unsigned char type;
  unsigned char trailingnumbers;
  unsigned char invalid;
  struct service *service;
  struct ht_elem *service_hte;
  void *private_data;
  int objlen;
  long hashsum;
  int count;
  confirmfun_t confirmfun;
  struct ht_elem *prev,*next,**pprevhash,*nexthash;
};
```

- obj is the object (whose length is objlen bytes)
- type is the tag of the object type
- hashsum is the hash sum, it allows a quick selection among the collision list, if the hash sum does not coincide, the object is not the one currently wanted one.
- tst is the timestamp as defined by the treepoch module.
- service and service_hte are quick link to the service (module) owning this element.

- private data is an opaque data where the module can store its information about this object.
- count is the number of instances currently used for garbage collection.
- confirmfun if the confirmation function to manage exceptions.
- mtabline is the mount tab line (the one shown by umproc in /proc/mounts)
- prev,next,pprevhash,nexthash, links for the collision list, and for the linear scan of all the elements of the same type.

Each kind of object has its own search policy. Sometimes there are more different policies for the same type of object.

- CHECKPATH (pathnames): there is a tree traversal from the root to the leaf. Step by step each component of the pathname is added and the resulting partial path is searched in the hash table. The search process provides the most recent match among those found. To be more precise, the first scan provides the sequence of all the most recent matched that has a non-null confirmation function (so may have exceptions) plus eventually the first without exceptions. This sequence is named carrot, see over.
- CHECKPATHEXACT: for umount: only complete match is permitted.
- CHECKSOCKET, CHECK CHR/BLK DEVICE, CHECKSC: these objects are integers or sequence of integers. All the objects stored in the hash table having a common prefix (integer by integer) can match.
- CHECKMODULE: search a module by its module name (complete match).
- CHECKFSTYPE: the name of the file system (for mount) must have a module name as a prefix.
- CHECKFSALIAS: standard string match.

The first part of the search process generate the list of possible matches, i.e. the list of possible most recent matches (in terms of timestamp), those having a confirmation function plus the first with confirmfun==NULL. This list is named carrot. The idea of mount in View-OS can be thought as a layer that changes the view. A layer without exception is completely opaque while it is semi-transparent when exceptions may occur. A carrot is a probe resulting by digging all possibly tranparent layers to the first opaque. The search algorithm then calls all the confirmation function, returning the first confirmed match.

The object type is used in the hash sum and key computation, thus objects having the same value but different types are stored independently. Sometimes modules register null objects (zero-length). These objects (of the same type) obviously have the same hash sum and key. The collision list for zero-length is stored in a separate list (to prevent the collision with objects of different types).

When a user process requires a system call, view-os searches in the hash table which is the object which is responsible to handle it. The hash table element (often referred in the code as hte) is used by the whole virtual machine monitor (umview/kmview) and by modules as a key to find the virtualization which applies. View-OS modules does not need (any more) to implement their search methods or mount tables (as it happened in View-OS 0.6). The implementation of the system calls in the modules can access the private data of the virtualization for the current request using um_mod_get_private_data.

umproc.c manages all the other information about processes. In particular, it keeps the open file table synchronized. Each file, including those not handled by any service module, has an entry in this table. Absolute paths of the open file are stored together with management information.

The third layer, alias the wrapper layer, consists of functions that restructure all the data for the service module calls. These functions are in source files named with a `um_` prefix, e.g. `um_basicio`, `um_plusio` etc... Each the system calls have in this layer a `wrap_in` and a `wrap_out`. Sometimes wrap functions are shared for similar calls. A `wrap_in` function decides if the kernel system call must be executed or not. Usually it calls the module implementation of the system call and then return `STD_BEHAVIOR`, `CALL_ON_XIT` or `SC_FAKE`.

Most of them just deal with data fetching and storing, but some require original solutions to be implemented. We will limit the discussion to this latter case.

When a file descriptor is opened (e.g. by the open or creat system call, but also with socket and dup), there is a need to give the process a meaningful descriptor to represent the virtual descriptor. All these calls are then translated into an open call to the read side of a named pipe. This is useful for managing select or poll system calls. The process inside the ParVM will block onto the named pipe, and umview can signal the process by writing something on the pipe.

The first successive virtual I/O operation will empty the named pipe. The management of the current working directory is another critical point. Obviously, the real hosting kernel does not accept chdir operation for non-existent directories. It has been necessary to split the view of the working directory of the kernel from the view of processes. When a process has an existing directory (in the real file system) as its working directory, then both views coincide. On the other hand, when the process has a virtual directory as the working directory, the chdir for the real kernel moves to an umpty subdirectory in /tmp. The current directory view of the kernel does not appear to processes, as all the calls regarding working directory management are virtualized.

Another original solution has been implemented to select and poll calls. These calls could block umview. Non-blocking calls are used to check that the conditions requested by select or poll are already satisfied. If the process has to wait, umview registers a call back at the service module. When the unblocking event occurs, the module calls umview back to unblock the calling process.

um_services is the source code which manages the umview management system call. Through this call it is possible to add, remove, and lock service modules. It is also possible to modify the sequence of service module applications. The commands named um_add_service, um_del_service, um_ls_service, um_mov_service, um_lock_service use this system call. um_services decodes the user-level request and calls the corresponding function defined in services.c. Each command of the list above has a tag for um_services. LIST_SERVICE returns a tag list (one byte each) which is the sequence of service codes. NAME_SERVICE is the tag that retrieves the name of the module from its code. LOCK_SERVICE denies any further changes to the partial virtual machine.

If `pure_libc` is installed, the system calls generated by modules or by libraries gets captured by the `capture_nested` module of umview. This self virtualization feature allow the service nesting. The system call generated by the virtualization of an entity (e.g. a virtual file system) can be captured and further virtualized. `capture_nested` uses specific wrappers, not those in the `um_*.c` files. In fact, nested calls do not require to retrieve arguments from register and from the private memory of processes.

UMview supports also nested invocations of `umview` itself. In fact, if inside a UMview session a user can start another `umview` machine. Actually istead of starting another umview process and tracing the system calls twice (it would be a performance bottleneck and it is very hard for ptrace limitations) the running UMview process virtualizes the activation of the new machine.

Both virtualization nesting for modules and nested invocation of umview use a specific data structure for timestamping the events: `treepoch` (the name is a contraction of tree of epochs). Epoch is a 64bits counter, it is incremented each time there is a change in a view, e.g. a mount or umount operation succeded. The current epoch can be read using the function `get_epoch`.

The idea is that each mount operation has a timestamp. When a process executes a system call each module search if it can process the request as the result of a mount operation it accepted. If the system call is in the domain of several mount operation the *most recent* is chosen (for this reason the check functions of modules return a value of type `epoch_t`). In the same way if several modules return positive values, i.e. can manage the call, the *most recent* wins.

In this way if several file system, device, network has been mounted at the same location of the file system, the latest mount operation is the one seen by the process. It is the idea of layered mount shown in 9.1.

In the paragraph above, the words "*most recent*" are in italics because the problem is a bit harder. As a second approximation we can say that *most recent*" may mean, *the most recent before now*. It sounds a bit silly as at a first sight it seems impossible to have something happend after *now*.

But when a system call gets captured by a module, the implementation function of the module has the current time (epoch) moved back to the time when the correspondent mount operation M was executed. In this way all the system calls generated by the module system call implementation are captured by pure_libc and executed in the environment at the time of M. All the mount operations with timestamp *older* than M, while all the mount operations *more recent* than M are not considered.

As an example consider the follwing situation: initially the epoch is 12; the user mounts a filesystem $FS1$ on `/mnt` by the module $M1$, epoch goes to 13, `/mnt/image` (inside $FS1$) is the image of the file system $FS2$. The user mounts it again on `/mnt` my the module $M2$, epoch is now 14.

When a user's process reads `/mnt/myfile`, $M1$ and $M2$ find out that this file is inside the mountpoint subtree of both mount. The mount of $FS2$ is the most recent. Module $M2$'s implementation of read is called, but the epoch is moved back to 13. $M2$ need to read `/mnt/image` which is again in the subtree of both mountpoints. This time the second mount cannot be considered because is *too recent* for the current epoch, thus $M1$'s read is called.

Unfortunately there is also another source of complexity: umview can run several nested umview invocations at the same time. When `umview` is started as a command inside a UMview machine, the modifications due to mount operation executed before the `umview` command will be seen by both partial virtual machines, while the mount operations executed after that time will be local the PVM when `mount` was called.

The timestamp used by treepoch is logically composed by an epoch and a bit-string. When a new umview nested machine is started the caller machine adds a 0 to its bit string, while the new machine inherits the bitstring of the old one plus a trailing 1.

The *most recent* control becomes:

- mount epoch must be older than the current epoch, and,
- mount bitstring must a subset (proper or not proper) of the process bitstring, and,
- mount epoch must have the maximum value.

Figure 9.7 shows the timestamping for three nested invocations and three mount operations. The first umview machine (umview0) mounts A, starts umview1 and mounts B. umview1 starts umview2 which mounts C and then umview1 mounts D. At the point ▼ the timestamp for umview0 is (61,"0"), for umview1 is (61,"10") and for umview2 is (61,"11"), (termination of processes is explained in the following). umview1 sees B and A, umview1 sees D and A, umview2 sees C and A. Note that umview0, umview1 and umview2 are not umview processes. UMview does not fork to handle nested invocations. In the example the command `umview xterm` starts a umview process, that execute a virtual syscall `UM_SERVICE/RECURSIVE_VIEW`. If this system call succeeds it means that this is a recursive umview as the real kernel does not implement the call. When the existing UMview receives the system call it splits the view by assigning to the former view a new bit-string with a trailing 0 and one with a trailing 1 to the calling process.

The bitstring size could increase monotonically in this way as *Mview adds one bit for each nested `umview` invocation. Treepoch avoids this problem: when there are no processes left for a nested invocation the correspondent bit is deleted. The bitstring in the timestamp is a pointer to a node in a tree where the actual bitstring is stored. When a node gets deleted all the bitstrings in the subtree are recomputed.

Figure 9.8 illustrates the evolution of the tree for the sequence of actions of Figure 9.7. At state ■ the processes of umview0 run in the state *emptystring* which is the sole node of the tree. The timestamp of the mount A points to the same node. Each node of the tree has a creation epoch. For the root it is an epoch less than 55, say 1. When umwiew1 starts (state ♦), the root node is moved as the left son of a new root, thus all the processes running in umview change their bit string without any loop (complexity O(1)). The new root keeps the creation epoch 1, while the creation epoch of node 0 is 56. The mount A operation initially (and erroneously) keeps the pointer to the former root, now the node with bitstring 0. Treepoch use a lazy update method, each time a timestamp is read if it has an older creation epoch than the node the pointer is moved up towards the root. In our situation it is moved to point to the new root. Note that mount operations timestamps can point to any node of the tree while running processes always point to the leaves.

	umview0	umview1	umview2
	timestamp=(55,"")		
	mount A		
■	timestamp=(56,"")		
	umview xterm →	*starting*	
	timestamp=(57,"0")	timestamp=(57,"1")	
	mount B		
⧫	timestamp=(58,"0")		
		umview xterm →	*starting*
	timestamp=(59,"0")	timestamp=(59,"10")	timestamp=(59,"11")
			mount C
	timestamp=(60,"0")	timestamp=(60,"10")	timestamp=(60,"11")
		mount D	
▲	timestamp=(61,"0")	timestamp=(61,"10")	timestamp=(61,"11")
	terminate umview0		
▼		timestamp=(62,"0")	timestamp=(62,"1")

Figure 9.7: UMview nested invocation and (simplified) treepoch timestamping, symbols in the first column are reference to Figure 9.8

The state ▲ shows the evolution of the tree after the starting on umview2, and after C and D has been mounted. When umview0 terminated, i.e. when all the last process running in umview0 exits (state ▼), the former node 1 is moved to the root but it inherits the creation epoch of the former root. The former nodes are not deleted if there are mount operations pointing to them, they stay as zombie nodes (the former leaf keeps its creation epoch, while the former root is assigned the maximum 64bit number). With a lazy update method, the pointers to zombie nodes gets moved to their root node, (if the timestamp of the mount operation is older than the creation date of the zombie node, otherwise the is automatically unmounted).

*Mview design choices

There are in *Mview some design choices that were needed to workaround some lack of support of this new kind of virtualization. This subsection describes some of the solutions included in *Mview code.

*Mview: changing syscall parameters *Mview needs sometimes to change the parameters for system calls. It is the case, for example, of the `open` system call for virtual files. The pathname of the file to open is set to the path of a fifo used to communicate with the *Mview hypervisor for two reasons:

- to reserve a valid file descriptor for the virtual file;
- to unblock `select` or `poll` calls when events occur on the virtual file.

In the same way `execve` for a virtual file should use a temporary copy of the file in the file system instead. While scalar system call parameters stored in the registers are easy to change, pathnames or other complex

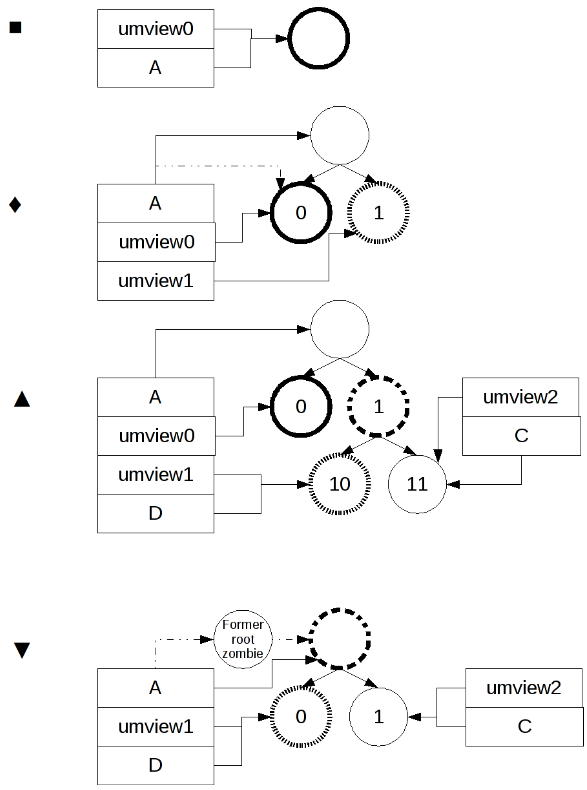

Figure 9.8: Treepoch timestamping: tree structure

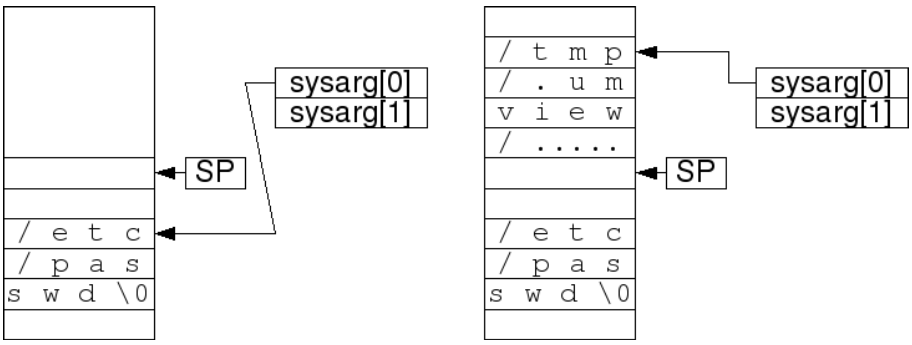

Figure 9.9: System Call management: syscall argument substitution

structures stored in the stack or in other parts of the memory require some more efforts.

*Mview uses area above the stack pointer of the process to store data like pathnames or complex memory structures needed by the kernel to run the system call as shown Figure 9.9.

This method does not generate conflicts because these values are used by the kernel and do not need to retain their values when the system call returns and because the kernel uses a different stack area to process the system call. The value of the stack pointer is not changed.

Canonicalize

The file canonicalize.c include a new implementation of the realpath function of the C library. `um_realpath` has several new features:

- It has a flag to decide if a leaf symbolic link should be traversed or not (to support l- and not l- system calls, like lstat vs. stat).
- It minimizes the number lstat/access/readlink calls.
- It supports a starting directory for relative pathames (for -at system calls, like openat).
- It uses a recursive function.

um_realpath starts the recursive rec_realpath canonicalize function. um_realpath, in fact, creates the non-canonicalized absolute path i.e.: the current root dir followed by the path specified by the user if the path is absolute, the starting directory followed by the path specified by the user for relative pathnames.

The recursive canonicalization function rec_realpaths takes the non-canonicalized absolute path and calls itself for each path component added to the canonicalized (resolved) pathname.

If there is a dot `.` in the pathname rec_realpath loops without calling itself, if there is a dotdot `..` it returns and the calling rec_realpath loops (it has the effect to delete a component from the canonicalized pathname).

When rec_realpath reaches a symbolic link it creates a new non-canonicalized path composed by the symbolic link target followed by the remaining part of the pathname then it loops and continue to resolve the pathname if the symbolic link is relative or it returns ROOT. In this latter case all the recursive calls of rec_realpath returns up to the one correspondent to the current root. (it has the effect to delete all the components of the canonicalized pathname following the current root).

*Mview path rewriting and chroot management. There is a path rewriting control in *Mview: when a process request is not managed by a module (i.e. *Mview redirects the request to the kernel) sometimes the canonicalized pathname computed by *Mview and the pathname used by a system call can be different.

It is the case, for example, of `chdir("..")` from the root of a virtually mounted file system. In this case the current working directory for the real kernel is a parking directory (the real kernel does not know anything about the virtual file system) and without a path rewriting control the command would change the working directory to the parent of the parking directory.

Virtual chroot support has been implemented by the path rewriting control. In fact, after the redefinition of a new root by chroot, *mview rewrites all the pathnames for the system calls redirected to the kernel.

chroot behavior has been implemented to be consistent with the Linux kernel's chroot: all the cwd pathnames referring to the chroot-ed subtree are relative to the new root, all the remaining paths are absolute.

*Mview management of mmap. Unfortunately mmap access to the file systems cannot be captured by ptrace (or by utrace). V^2 is studying patches and extensions to the utrace support for this purpose.

In the meanwhile V^2 has developed a way to give a limited support for mmap, expecially for the support of dynamic libraries. Only read only, `MAP_PRIVATE` calls are currently supported on virtual files. The trick is based on a hidden file opened by umview or kmview and inherited through all the fork and exec calls.

All the mmap support code is in the `um_mmap.c` source file.

If a process under *Mview control tries to execute a system call on that hidden file gets an error.

When a process executes a mmap on a virtual file, the file gets copied in a section of the hidden file. The support system take trace of the mapping between virtual files and sections of the file. Each section of the file is shared by all the process using the same file in mmap mode. This is quite common when dealing with dynamic libraries. The garbage collector delays the deallocation of sections using a LRU approximation. This avoids loading and unloading libraries and files when frequently used.

The data structure used is a list, the head is `mmap_sf_head` and each section (or chunk) information is stored in a `struct mmap_sf_entry` element.

LRU approximation uses the lastuse field, its MSB is set at each use and right shifted at any allocation. when more than `sizeof(long)` files get allocated and a chuck is still unused, it is unloaded and the space freed.

mmap call arguments get changed to use the hidden file in the right position.

*Mview management of select/poll/pselect/ppoll.

*Mview implements partial virtualization. It means that system calls are executed by the *Mview hypervisor (umview or kmview) when referring to virtualized entities, or by the kernel otherwise.

This is (quite) simple for calls referring to the file system (like open) or to file descriptors (read). The absolute pathname or the file descriptor is the key value to decide whether the call refers to a virtualized entity or not.

Unfortunately the select/poll set of system calls works on sets of file descriptors, maybe some referring to virtualized files and others to real files.

The metod used to manage these calls is an evolution of that used in the Alpine project and then in our former Ale4Net project. When a system call need to define a new file descriptor (open, creat, socket) for a virtualized file, it returns a (valid) file descriptor of a named pipe opened

```
#define VIRUMSERVICE 1
#define VIRSYS_MSOCKET 2

#define ADD_SERVICE 0
#define DEL_SERVICE 1
#define LIST_SERVICE 3
#define NAME_SERVICE 4
#define VIEWOS_GETINFO     0x101
#define VIEWOS_SETVIEWNAME 0x102

#define VIEWOS_KILLALL     0x103

#define VIEWOS_ATTACH      0x104
#define VIEWOS_FSALIAS     0x105

struct viewinfo {
  struct utsname uname;
  pid_t serverid;
  viewid_t viewid;
  char viewname[_UTSNAME_LENGTH];
};

int um_add_service(int position,char *path) {
  return virsyscall3(VIRUMSERVICE,ADD_SERVICE,position,path); }

int um_del_service(int code) {
  return virsyscall2(VIRUMSERVICE,DEL_SERVICE,code); }

int um_list_service(char *buf, int len) {
  return virsyscall3(VIRUMSERVICE,LIST_SERVICE,buf,len); }

int um_name_service(int code, char *buf, int len) {
  return virsyscall4(VIRUMSERVICE,NAME_SERVICE,code,buf,len); }

int um_view_getinfo(struct viewinfo *info) {
  return virsyscall2(VIRUMSERVICE,UMVIEW_GETINFO,info); }

int um_setviewname(char *name) {
  return virsyscall2(VIRUMSERVICE,UMVIEW_SETVIEWNAME,name); }

int um_killall(int signo) {
  return virsyscall2(VIRUMSERVICE,UMVIEW_KILLALL,signo); }

int um_attach(int pid) {
  return virsyscall2(VIRUMSERVICE,VIEWOS_ATTACH,pid); }

int um_fsalias(char *alias,char *filesystemname) {
  return virsyscall3(VIRUMSERVICE,VIEWOS_FSALIAS,alias,filesystemname); }

long msocket(char *path, int domain, int type, int protocol) {
  return virsyscall4(VIRSYS_MSOCKET,path,domain,type,protocol); }
```

Figure 9.10: *mview specific system calls

in read only mode. The other end is opened (twice) by the *mview hypervisor.

A select or poll system call get changed to wait from the real files and something to read from the virtual files. If data on real file unblock the syscall the hypervisor gets informed by ptrace/utrace. If the hypervisor needs to unblock the process (as the state of virtual files change), it sends a char on the named pipe. The process syscall unblocks, the hypervisor gets informed, it rewrites the arguments to merge the events notification on real and virtualized files and finally it reads the char from the named pipe. This latter action is necessary for the next select/poll call block on the virtual file.

Figure 9.10 shows all the tags defined for *umservice* with their arguments, and *mocket*.

*Mview management of exec for virtual files.

`execvp` needs a file in the file system. When the file is a virtual file there is no way to feed the file to `execvp`. The basic idea is to create a temporary copy of the executable file and then execvp it instead of the virtual file.

The support for exec is in `um_exec.c`.

There is also the binfmt support that needs to execute an interpreter for the executable instead of the executable, and the interpreter can be a virtual file, too.

Let us first consider the case of standard executables (without binfmt, see the `wrap_in_execve`, at the end, the else branch of `if (binfmtser != UM_NONE))`.

When the file is virtual (and the module does not need to give a specific implementation of execve) the executable is copied (function filecopy) and the executed. The virtual file temporary copy file will be deleted at the next system call executed by the same process (thus `execve` will be completed at that time).

When binfmt need to start an interpreter for the executable, *mview executes a wrapper named `umbinwrap` instead of the executable. `umbinwrap` encodes in `argv[0]` the path of the interpreter, the path of the executable and when binfmt needs it the former `argv[0]`.

The wrapper decodes `argv[0]` and starts the interpreter, working as a standard process under the control of *mview. umbinwrap uses `execvp` to start the interpreter thus it could be a virtual file or even a further interpreter, but `um_exec` can cope with both cases.

*Mview management of specific and module-added system calls

*Mview adds some system calls like *umservice* and *msocket*, which are not part of the standard set of Linux syscalls.

The umview private syscalls are piggybacked on `sysctl(2)` system calls. *mview use sysctl with NULL `name`. This configuration is impossible for a real sysctl call. The virtual syscall number is stored in the field `nlen`, the array of arguments in `newval`, the number of arguments in `newlen`.

The `um_lib.h` include file defines some macro `virsyscall0` to `virsyscall6`. These macro execute a *mview system call, the suffix digit is the number of parameters.

The system call *umservice* is used to configure the *mview hypervisor. The first argument is a tag for the request.

`um_add_service`, `um_del_service` have the same meaning of their correspondent commands. `um_list_service` provide a list of the codes of loaded modules. `um_name_service` maps each code of a module to its name. `um_view_getinfo` is used by vuname, it provides an externded version of `struct utsname` including all the info about the *mview environment.

*MView process control table The process control table element of *MView contains fields related to different layers of the internal architecture of tha application. There are fields for the capture layer, for sctab, etc. Even some wrap modules like select and mmap need specific fields. Instead of a single complex (maybe unreadable) include file with all the stuff related to the process control block nmanagement there are several sections of pcb.h named pcb.00.capture.h, pcb.00.mainpoll.h etc.

Each section has several sections defined by `#ifdef`

```
#ifdef _PCB_DEFINITIONS
/* data type definitions, inclusion of header file */
#endif

#ifdef _PCB_COMMON_FIELDS
/* fields both for process pcb and nested calls pcb */
#endif

#ifdef _PCB_ONLY_FIELDS
/* fields only for pcb of processes */
#endif

#ifdef _NPCB_ONLY_FIELDS
/* fields only for pcb of nested calls */
#endif

#ifdef _PCB_CONSTRUCTOR
/*pcb contructor for this section */
#endif

#ifdef _PCB_DESTRUCTOR
/*pcb destructor for this section */
#endif
}
```

All the sections gets concatenated by the makefile into pcb-all.h and pcb.h generates the data structure including pcb-all.h several times with different constants defined as shown in Figure 9.11.

9.6 ♦UMview patches for the Linux Kernel

PTRACE_VM

This patch simplify and extends the management of `PTRACE_SYSCALL`, `PTRACE_SINGLESTEP`, `PTRACE_SYSEMU`, `PTRACE_SYSEMU_SINGLESTEP`, `PTRACE_BLOCKSTEP` etc.

The idea is to use tags in the "addr" parameter of existing `PTRACE_SYSCALL` `PTRACE_SINGLESTEP` `PTRACE_CONT`,`PTRACE_BLOCKSTEP` calls to skip the current call (`PTRACE_VM_SKIPCALL`) or skip the second upcall to the VM/debugger after the syscall execution (`PTRACE_VM_SKIPEXIT`).

```
#ifndef _PCB_H
#define _PCB_H
#include <sys/stat.h>
#include <sys/ptrace.h>
#include <asm/ptrace.h>
#include "treepoch.h"

/* ... */

#define _PCB_DEFINITIONS
#include "pcb-all.h"
#undef _PCB_DEFINITIONS

struct pcb {
#define _PCB_COMMON_FIELDS
#include "pcb-all.h"
#undef _PCB_COMMON_FIELDS
#define _PCB_ONLY_FIELDS
#include "pcb-all.h"
#undef _PCB_ONLY_FIELDS
};

struct npcb {
#define _PCB_COMMON_FIELDS
#include "pcb-all.h"
#undef _PCB_COMMON_FIELDS
#define _NPCB_ONLY_FIELDS
#include "pcb-all.h"
#undef _NPCB_ONLY_FIELDS
};

void pcb_constructor(struct pcb *pcb,int flags,int npcbflag);
void pcb_destructor(struct pcb *pcb,int flags,int npcbflag);
void pcb_inits(int flags);
void pcb_finis(int flags);

#endif
```

Figure 9.11: Structure of pcb.h file

The ptrace tag `PTRACE_SYSEMU` is a feature mainly used for User-Mode Linux, or at most for other virtual machines aiming to virtualize *all* the syscalls (total virtual machines).

In fact:

```
ptrace(PTRACE_SYSEMU, pid, 0, 0)
```

means that the *next* system call will not be executed. `PTRACE_SYSEMU` has been implemented only for the `x86_32` architecture.

This patch extends the features of the standard ptrace tags as follows:

```
ptrace(PTRACE_SYSCALL, pid, XXX, 0)
```

This call:

- is the same as `PTRACE_SYSCALL` when XXX==0,
- skips the call (and stops before entering the next syscall) when `PTRACE_VM_SKIPCALL | PTRACE_VM_SKIPEXIT`
- skips the ptrace call after the system call if `PTRACE_VM_SKIPEXIT`.

This patch has been implemented for x86_32, powerpc_32, um+x86_32. (x86_64 and ppc64 exist too, but are less tested).

The main difference between SYSEMU and the new support is that with PTRACE_VM it is possible to decide if *this* system call should be executed or not (instead of the next one). PTRACE_VM can be used also for partial virtual machines (some syscall gets virtualized and some others do not), like our umview.

PTRACE_VM above can be used instead of `PTRACE_SYSEMU` in user-mode linux and in all the others total virtual machines. In fact, provided user-mode linux skips *all* the syscalls it does not matter if the upcall happens just after (SYSEMU) or just before (PTRACE_VM) having skipped the syscall.

The patch is backward compatible with existing applications: the addr field is defined as unused in the former ptrace specifications. All the code examined (user-mode linux, strace, umview...) use 0 or 1 for addr (being defined unused). Defining `PTRACE_VM_SKIPCALL=4` and `PTRACE_VM_SKIPEXIT=2` (i.e. by ignoring the LSB) everything previously coded using `PTRACE_SYSCALL` should continue to work. In the same way `PTRACE_SINGLESTEP`, `PTRACE_CONT` and `PTRACE_BLOCKSTEP` can use the same tags restarting after a SYSCALL.

This patch would eventually simplify both the kernel code (reducing tags and exceptions) and even user-mode linux and umview.

The skip-exit feature can be implemented in a arch-independent manner, while for skip call some simple changes are needed (the entry assembly code should process the return value of the syscall tracing function call, like in `arch/x86/kernel/Entry_32.S`).

V^2 is proposing this patch to enter the Linux mainstream for the reasons listed in the following list:

1. (eventually) Reduce the number of PTRACE tags. The proposed patch does not add any tag. On the contrary after a period of deprecation SYSEMU* tags can be eliminated.

2. it is backward compatible with existing software (existing UML kernels, strace already tested). Only software using strange "addr" values (currently ignored) could have portability problems.

3. (eventually) simplify kernel code. SYSEMU support is a bit messy and x86/32 only. These new `PTRACE_VM` tags for the addr parameter will allow to get rid of SYSEMU code.

4. It is simple to be ported across the architecture. This patch already support `PTRACE_VM_SKIPEXIT` for all architectures and `PTRACE_VM_SKIPCALL` for x86_32/64 (incl. x86_64 emu32), powerpc32/64, UML.

5. It is more powerful than `PTRACE_SYSEMU`. It provides an optimized support for partial virtualization (some syscalls gets virtualized some other do not) while keeping support for total virtualization à la UML.

6. Software currently using `PTRACE_SYSEMU` can be easily ported to this new support. The porting for UML (client side) is already in the patch. All the calls like:

   ```
   ptrace(PTRACE_SYSEMU, pid, 0, 0)
   ```

   can be converted into

```
ptrace(PTRACE_SYSCALL, pid, PTRACE_VM_SKIPCALL, 0)
```

(but the first `PTRACE_SYSCALL`, the one which starts up the emulation. In practice it is possible to set `PTRACE_VM_SKIPCALL` for the first call, too. The "addr" tag is ignored being no syscalls pending).

PTRACE_MULTI

ptrace's PTRACE_MULTI tag sends multiple ptrace requests with a single system call. In fact, a process that uses ptrace() often needs to send several ptrace requests in a row, for example PTRACE_PEEKDATA for getting/setting some useful, even small pieces of data, or several registers or other ptrace commands.

You can see this fact using the following commands:

```
strace -o /tmp/trace strace ls
```

or

```
strace -o /tmp/trace linux ubd0=linux.img
```

(where linux is a UML kernel).

Looking into /tmp/trace you'll see runs of ptrace syscalls like: (strace example) ...

```
ptrace(PTRACE_PEEKUSER, 27177, 4*ORIG_EAX, [0xb]) = 0
ptrace(PTRACE_PEEKUSER, 27177, 4*EAX, [0xffffffda]) = 0
ptrace(PTRACE_PEEKUSER, 27177, 4*EBX, [0xbfe2d4b0]) = 0
ptrace(PTRACE_PEEKUSER, 27177, 4*ECX, [0xbfe2e698]) = 0
ptrace(PTRACE_PEEKUSER, 27177, 4*EDX, [0xbfe2e6a0]) = 0
ptrace(PTRACE_PEEKDATA, 27177, 0xbfe2d4b0, [0x6e69622f]) = 0
ptrace(PTRACE_PEEKDATA, 27177, 0xbfe2d4b4, [0x736c2f]) = 0
ptrace(PTRACE_PEEKDATA, 27177, 0xbfe2e698, [0xbfe2f992]) = 0
ptrace(PTRACE_PEEKDATA, 27177, 0xbfe2f990, [0x736c0065]) = 0
ptrace(PTRACE_PEEKDATA, 27177, 0xbfe2f994, [0x45485300]) = 0
ptrace(PTRACE_PEEKDATA, 27177, 0xbfe2e69c, [0]) = 0
ptrace(PTRACE_PEEKDATA, 27177, 0xbfe2e6a0, [0xbfe2f995]) = 0
...
```

(uml example)

```
ptrace(PTRACE_SETREGS, 27086, 0, 0x82f16bc) = 0
ptrace(PTRACE_CONT, 27086, 0, SIG_0)    = 0
...
```

It is useful for these programs to run several ptrace operations while limiting the number of context switches. For Virtual Machines limiting the number of context switches is a must, while a speed up for debuggers is not so crucial but it helps. Having a faster debugger should not be a problem, expecially when you've to fix large complex programs... For User Mode Linux the number of context switches due to ptrace should be reduced 33(look at the trace above, all the sequences PTRACE_SETREGS followed by PTRACE_CONT or PTRACE_SYSCALL or PTRACE_SYSEMU collapse in a single PTRACE_MULTI call).

Ptrace-multi gets a "struct ptrace_multi" array parameter (together with its number of elements). It is a similar concept/syntax to the management of buffers for readv or writev.

```
struct ptrace_multi {
  long request;
  long addr;
  void *localaddr;
  long length;
};
```

Each `struct ptrace_multi` specifies a single standard ptrace request. So you can join several requests into one request array that will be passed through the "void* addr" parameter (the third) of ptrace(). request, addr and localaddr have the same meaning of ptrace's request, addr and data field for a single request. Here is an example of PTRACE_MULTI call:

```
struct ptrace_multi req[] = {
  {PTRACE_SETREGS, 0, regs, 0},
  {PTRACE_SYSCALL, 0, 0, 0}};
if (ptrace(PTRACE_MULTI,pid,req,2))
  /*ERROR*/
```

The last field in the struct (length) specifies the numbers of requests to be accomplished by ptrace on a sequence of words/bytes. - PTRACE_PEEKTEXT, PTRACE_PEEKDATA, PTRACE_PEEKUSR, PTRACE_POKETEXT, PTRACE_POKEDATA, PTRACE_POKEUSR requests load/store chunks of registers, data, text code. "length" is the number of memory words to exchange. field==0 has the same meaning as field=1.

While normal ptrace requests can get a word at a time, I have added some other request for simplify the interface between kernel and applications that use trace(); these requests can get from user space more than one word at a time: - PTRACE_PEEKCHARDATA and PTRACE_POKECHARDATA is used for transferring general ata, like structure, buffer, and so on... lenth is in bytes. - PTRACE_PEEKSTRINGDATA get strings from user space (using the new mm function: `access_process_vm_user`) stopping the transfer if the `'\0'` string termination occur. length is in bytes.

Debuggers and virtual machines (like User Mode Linux, or Virtual Square's umview) and many other applications that are based on ptrace can get performance improvements by PTRACE_MULTI: the number of system calls (and context switches) decreases significantly.

This patch is architecture independent. This patch is logically independent with PTRACE_VM: applying this patch after PTRACE_VM (and viceversa) generates just some warnings about line offsets.

Probing PTRACE features

Programs using ptrace must be able to probe which ptrace features are provided by the kernel, and adapt their behavior consequently. The structure of code of the test in `ptrace_multi_test.c` is in Figure 9.12 (the same code has been ported to the patch for User-Mode Linux based on `PTRACE_VM`). The tags

```
    static int child(void *arg)
    {
      int *featurep=arg;
      int p[2]={-1,-1};
      if(ptrace(PTRACE_TRACEME, 0, 0, 0) < 0)
        perror("ptrace test_ptracemulti");
C1    kill(getpid(), SIGSTOP);
C2    getpid();
      *featurep=PTRACE_SYSCALL_SKIPEXIT;
C3    pipe(p);
      if (p[0] < 0)
        *featurep=PTRACE_SYSCALL_SKIPCALL;
C4    getpid();
      return 0;
    }
    /* kernel feature test:
     * exit value =1 means that there is ptrace multi support
     * vm_mask is the mask of PTRACE_SYSVM supported features */
    unsigned int test_ptracemulti(unsigned int *vm_mask) {
      int pid, status, rv;
      static char stack[1024];
      *vm_mask=0;
P0    if((pid = clone(child, &stack[1020], SIGCHLD | CLONE_VM, vm_mask)) < 0){
        perror("clone"); return 0; }
P1    if((pid = r_waitpid(pid, &status, WUNTRACED)) < 0){
        perror("Waiting for stop"); return 0; }
P2    rv=ptrace(PTRACE_SYSCALL, pid, 0, 0);
P3    if(waitpid(pid, &status, WUNTRACED) < 0) goto out;
P4    rv=ptrace(PTRACE_SYSCALL, pid, PTRACE_SYSCALL_SKIPEXIT, 0);
      if (rv < 0) goto out;
P5    if(waitpid(pid, &status, WUNTRACED) < 0) goto out;
      if (*vm_mask<PTRACE_SYSCALL_SKIPEXIT) goto out;
P6    rv=ptrace(PTRACE_SYSCALL, pid, PTRACE_SYSCALL_SKIPCALL, 0);
P7    if(waitpid(pid, &status, WUNTRACED) < 0)
        return 0;
    out:
P8    if (ptrace(PTRACE_MULTI, pid, stack, 0) < 0)
        rv=0;
      else
        rv=1;
P9    ptrace(PTRACE_KILL,pid,0,0);
      if((pid = r_waitpid(pid, &status, WUNTRACED)) < 0){
        perror("Waiting for stop");
        return 0;
      }
      return rv;
    }
```

Figure 9.12: The code to probe PTRACE features supported by the current kernel

$C1, ..., C4$ and $P0, ..., P9$ have been added to track the syncronization between the function `test_ptracemulti` and the concurrent thread `child`.

The test of PTRACE_MULTI is simple (P8 in the code), the request for a sequence of zero request returns -1 or 0 depending on whether the feature is provided or not. The table 9.13 shows three executions: a kernel providing both SKIPEXIT and SKIPCALL, one providing SKIPEXIT, and the third is an unpatched kernel without any VM feature.

The probing thread `child` updates a shared variable (arg in `child` which is vmmask in `test_ptracemulti`. The two threads start a *dialogue*, doing one step, awakening the other, and waiting to be awaked again. The `child` thread tries two system calls, a getpid and a pipe. Getpid is used to test if SKIPEXIT works, in fact if it does not, the waitpid of line P5 reports the exit stop for PTRACE_SYSCALL after the system call execution. The `pipe` call

1. SKIPEXIT, SKIPCALL:

test_ptracemulti	probe thread
P0 start the probe thread P1 waitpid → P2 ptrace SYSCALL P3 → P3 ptrace SKIPEXIT P5 → P6 ptrace SKIPCALL P7 → P8 test multi P9 kill the probe thread	C1 initial stop C1 → C2 getpid vm_mask=SKIPEXIT → C3 pipe vm_mask=SKIPCALL (pipe was skipped) → C4

2. SKIPEXIT, NO SKIPCALL:

test_ptracemulti	probe thread
P0 start the probe thread P1 waitpid → P2 ptrace SYSCALL P3 → P3 ptrace SKIPEXIT P5 → P6 ptrace SKIPCALL P7 → P8 test multi P9 kill the probe thread	C1 initial stop C1 → C2 getpid vm_mask=SKIPEXIT → C3 pipe (pipe was not skipped p[0] is a valid fd) → C4

3. no extra features:

test_ptracemulti	probe thread
P0 Start the probe thread P1 waitpid → P2 ptrace SYSCALL P3 → P3 ptrace SKIPEXIT P5 → goto out P8 test multi P9 kill the probe thread	C1 initial stop C1 → C2 getpid C2 getpid (syscall exit was not skipped)

Figure 9.13: Some executions of test_ptracemulti

tests if SKIPCALL works leaving the file descriptors untouched (i.e. -1) or really creates a pipe.

9.7 ♦KMview internals

KMview shares the main part of the source code with UMview. The lower layer (system call capturing) is different. There is a kernel module (based on utrace extensions) and a the user-mode kmview application uses the `capture_km.c` source file instead of the `capture_um.c`.

`kmview_module` interface

`kmview_module` has been designed as a kernel support for view-os on linux but it is effectively an efficient support for any virtualization based on system call interception and transformation.

`kmview_module` could be effectively used also to security tools based on system call interposition.

There are two main entities in kmview: tracer and traced processes. A tracer process cannot trace itself but it can be a traced process of another tracer.

All the traced processes are in the offspring of their tracer, when a process is traced there is no way to exit from the control of the tracer.

A tracer process first open a read only connection to `/dev/kmview` (major=10,minor=233, officially assigned)

```
fd=open("/dev/kmview",O_RDONLY);
```

Before starting its first traced process, the tracer can set some flags to set some extra features in this way:

```
ioctl(fd, KMVIEW_SET_FLAGS, flags);
```

This ioctl must be called when there are no traced processes otherwise it returns `EACCES` (to prevent inconsistencies).

A "root" traced process is started in this way:

```
if (fork() == 0) {
  ioctl(fd, KMVIEW_ATTACH);
  close(fd);
  ..... code of the traced process, e.g. exec of some program
}
```

The root traced process must register itself as a traced process and close the tracing file. If the traced process forks (or clones) other processes they will be traced, too. No further direct interaction will take place between the traced process and their tracer.

If a tracer dies (or it closes the fd) all the traced processes will be killed (`SIGKILLi`).

A tracer receive all events related to its traced processes using a "read" or by the magicpoll technique (see over). The received data follows the struct `kmview_event` specification:

```
struct kmview_event {
  unsigned long tag;
  union {
    .... data for specific events ...
  }
}
```

There are four basic events identified by the following tags:

- `KMVIEW_EVENT_NEWTHREAD`: a new traced thread/process has just started
- `KMVIEW_EVENT_TERMTHREAD`: a new traced thread/process terminated
- `KMVIEW_EVENT_SYSCALL_ENTRY`: a traced process started a syscall
- `KMVIEW_EVENT_SYSCALL_EXIT`: a syscall for a traced process completed its execution.

In order to provide a fast interaction between kernel, module and tracer each layer keeps its own id for processes. The kernel identifies each process by its pid, the module has its own identifier named `kmpid` and the tracer can use its own identified, the `umpid`. (km stands for kernel-mode, um stands for user-mode). In this way each layer can use its identifier as an index within an array: there is not any waste of time to scan into tables or waste of code to keep hash tables. Technically speaking the whole system scales as O(1) (no extra costs related to the number of processes).

All the events reported by the module to the tracer carry the `umpid`, (except `KMVIEW_EVENT_NEWTHREAD`). All the requests sent (through ioctl) from the tracer to the module carry the `kmpid`. If a tracer tries to send an ioctl for a process handled by another tracer it gets an error (`EPERMi`). (a process handled by several nested tracers has a different `kmpid` for each tracer).

Basic Events

`KMVIEW_EVENT_NEWTHREAD`:

```
struct kmview_event_newthread{
  pid_t kmpid;
  pid_t pid;
  pid_t umppid;
  unsigned long flags;
} newthread;
```

A new thread has just started. The tracer must store its `kmpid`. umppid is the `umpid` of the parent (forking/cloning) process: the tracer can use this field to keep trace of the hierarchy in its data structures. flags are the cloning flags (as described in clone(2)). Before reading other events the tracer must send the `umpid` of this new thread to the module in this way:

```
struct kmview_ioctl_umpid {
  pid_t kmpid;
  pid_t umpid;
};
ioctl(fd, KMVIEW_UMPID, & {struct kmview_ioctl_umpid var} );
```

If a tracer wants to use pid or `kmpid` instead of having its own identifiers it should copy pid or `kmpid` respectively to the `umpid` field.

`KMVIEW_EVENT_TERMTHREAD:`

```
struct kmview_event_termthread{
    pid_t umpid;
      unsigned long remaining;
} termthread;
```

The process/thread identified by `umpid` terminated. No further event will be reported for that process/thread. The field remaining contains the overall number of processes handled by this tracer. Many tracers shut down when remaining==0;

`KMVIEW_EVENT_SYSCALL_ENTRY`:

```
struct kmview_event_ioctl_syscall{
  union {
    pid_t umpid;
    pid_t kmpid;
    unsigned long just_for_64bit_alignment;
  } x;
  unsigned long scno;
  unsigned long args[6];
  unsigned long pc;
  unsigned long sp;
} syscall;
```

(the `just_for_64bit_alignment` field, it is not used, it is a filler just for 64bit processors alignment.) `x.umpid` is the `umpid` identifier, `scno` is the syscall number, args are the arguments, `pci` is the program counter and `sp` the stack pointer. When the tracer receive the event the traced process is quiescent (as defined in utrace): it is waiting in a state very close to the user state. While a kmview traced process is quiescent the process can be restarted by its tracer or killed.

There are three different ways to restart a quiescent process for a `KMVIEW_EVENT_SYSCALL_ENTRY` event:

1. `KMVIEW_SYSRESUME`:

   ```
   ioctl(fd,KMVIEW_SYSRESUME,kmpid)
   ```

   the syscall gets retarted as is. The tracer will not receive any `KMVIEW_EVENT_SYSCALL_EXIT` event for this call.

2. `KMVIEW_SYSVIRTUALIZED`:

   ```
   ioctl(fd,KMVIEW_SYSVIRTUALIZED,
       &{struct kmview_event_ioctl_sysreturn var})

   struct kmview_event_ioctl_sysreturn{
     union {
       pid_t umpid;
       pid_t kmpid;
       unsigned long just_for_64bit_alignment;
     } x;
     long retval;
     long errno;
   } sysreturn;
   ```

   the call has been virtualized. This system call will not be executed by the linux kernel. `kmpid` must be set and the return value, errno will be those specified here. The tracer will not receive any `KMVIEW_EVENT_SYSCALL_EXIT` event for this call.

3. `KMVIEW_SYSMODIFIED`::

   ```
   ioctl(fd,KMVIEW_SYSMODIFIED,
       &{struct kmview_event_ioctl_syscall var})
   ```

The call may have been modified. The kernel will execute the syscall (maybe a different one) as stated by the registers. Registers, scno, will be changed. This call cause a `KMVIEW_EVENT_SYSCALL_EXIT` event after the syscall execution (to restore the original values of registers if needed).

`KMVIEW_EVENT_SYSCALL_EXIT`:

```
struct kmview_event_ioctl_sysreturn syscall;
```

With this event the tracer can get the result (return value or error) of the syscall executed by the linux kernel. The traced process is quiescent when the tracer receives the event. To restart the process the tracer can use `KMVIEW_SYSRESUME` or with the same syntax described above for `KMVIEW_EVENT_SYSCALL_ENTRY` or `KMVIEW_SYSRETURN` as follows:

```
ioctl(fd,KMVIEW_SYSRETURN,
    & {struct kmview_event_ioctl_sysreturn var})
```

by this latter call, return value and errno can be changed.

`KMVIEW_READDATA, KMVIEW_READSTRINGDATA, KMVIEW_WRITEDATA`: These tags are used to exchange data between the tracer and the memory of traced processes. The argument of this ioctl is a `struct kmview_ioctl_data` which has the following fields:

```
struct kmview_ioctl_data {
  pid_t kmpid;
  long addr;
  int len;
  void *localaddr;
};
```

`addr` and `len` are the address and len of the data in the memory of the traced process, while `localaddr` is the address in the tracer memory. `KMVIEW_READDATA, KMVIEW_READSTRINGDATA` copy data from the traced process, the latter stops the copy as soon as a NULL byte is copies. `KMVIEW_WRITEDATA` store data from the tracer to the traced memory.

Figure 9.14 is th source code of a minimal tracer using kmview:

Magicpoll.

When a tracer is a virtual machine monitor (an hypervisor, leaning the word from xen), often it does not keep waiting on a read as there are many source of events, file descriptors or signals.

If the hypervisor uses a ppoll it can wake up as soon as something happens. Unfortunately this means that for the standard virtualization cycle it needs two context switches to get the system call (or other event) from the kmview module.

```
#include <kmview.h>

void dowait(int signal)
{
  int w;
  wait(&w);
}

main(int argc, char *argv[])
{
  int fd;
  struct kmview_event event;
  fd=open("/dev/kmview",O_RDONLY);
  signal(SIGCHLD,dowait);
  if (fork()) {
    while (1) {
      read(fd,&event,sizeof(event));
      switch (event.tag) {
        case KMVIEW_EVENT_NEWTHREAD:
          {
            struct kmview_ioctl_umpid ump;
            printf("new process %d\n",event.x.newthread.pid);
            ump.kmpid=event.x.newthread.kmpid;
            /* we use umpid == kmpid */
            ump.umpid=event.x.newthread.kmpid;
            ioctl(fd, KMVIEW_UMPID, &ump);
            break;
          }
        case KMVIEW_EVENT_TERMTHREAD:
          printf("Terminated proc %d (%d left)\n",
              event.x.termthread.umpid,
              event.x.termthread.remaining);
          if (event.x.termthread.remaining == 0)
            exit (0);
          break;
        case KMVIEW_EVENT_SYSCALL_ENTRY:
          printf("Syscall %d->%d\n",
              event.x.syscall.x.umpid,
              event.x.syscall.scno);
          ioctl(fd, KMVIEW_SYSRESUME, event.x.syscall.x.umpid);
          break;
      }
    }
  } else { /* traced root process*/
    ioctl(fd, KMVIEW_ATTACH);
    close(fd);
    argv++;
    execvp(argv[0],argv);
  }
}
```

Figure 9.14: A minimal tracer based on the KMview kernel module

It the tracer sends the address of a buffer by the `KMVIEW_MAGICPOLL` ioctl, any select/poll like call will direcly tranfer the event to the buffer, thus when the return value of poll/select returns the availability of data for reading (e.g. POLLIN for poll), the data is already in the buffer. This reduces the number of context switches as there is no need to call read.

In order to further decrease the number of context switches per system call, it is possible to use an array of `struct kmview_event` as a magicpoll buffer. If there are several pending events (at most one per traced process), the kmview module fills in several elements of the array.

The magic poll ioctl is the following one:

```
struct kmview_magicpoll {
```

```
  long magicpoll_addr;
  long magicpoll_cnt;
};

ioctl(fd,KMVIEW_MAGICPOLL,& {struct kmview_magicpoll var} );
```

`magicpoll_addr` is the address of the buffer, `magicpoll_cnt` is the number of elements in the array. When poll returns either the array is full of pending events or the array element after the last significant pending event is tagged as `KMVIEW_EVENT_NONE` (i.e. the value zero).

KMVIEW_FLAG_SOCKETCALL.

Linux supports the Berkeley socket interface, but in many architectures instead of defining several different system calls (e.g. one for socket(2), one for connect, listen, accept etc.) is has just one system call (`__NR_socketcall`) with two parameters: the number of the call (as defined in `/usr/include/linux/net.h`) and a pointer to the array of parameters. It is the case of several widely used architectures like i386 or powerpc. Other architectures like x86_64, ia64 or alpha, has several system calls one for each Berkeley socket call.

To speed up the virtualization on architectures with `__NR_socketcall`, kmview provides the `KMVIEW_FLAG_SOCKETCALL` option.

When `KMVIEW_FLAG_SOCKETCALL` flag is set by `KMVIEW_SET_FLAGS`, the tracer receives a `KMVIEW_EVENT_SOCKETCALL_ENTRY` instead of the event `KMVIEW_EVENT_SYSCALL_ENTRY` when the system call `__NR_socketcall` is starting.

The `kmview_event_socketcall` structure is the following one:

```
struct kmview_event_socketcall{
  union {
    pid_t umpid;
    unsigned long just_for_64bit_alignment;
  } x;
  unsigned long scno;
  unsigned long args[6];
  unsigned long pc;
  unsigned long sp;
  unsigned long addr;
} socketcall;
```

`scno` is the number of the socket call (the number of the call listed in `/usr/include/linux/net.h`), `argsi` are the socket call args, `pc` and `sp` as usual, the final `addr` is the address of the argument array.

This prevents the hypervisor from spending one more context switch to grab the parameters.

The system call can be restarted by `KMVIEW_SYSRESUME`, `KMVIEW_SYSVIRTUALIZED`. Currently the module does not provide any `KMVIEW_FLAG_SOCKETMODIFIED` call: it is possible to use `KMVIEW_FLAG_SYSMODIFIED` to start another system call instead of a socket call, parameters of the same socket call can be changed by rewriting them using addr. Changing a socket call with another is rare, and must be done by hand carefully as the buffer pointed by addr can have not enough space to fit the new arguments.

KMVIEW_FLAG_FDSET.

There are several system calls that have a file descriptor in the first argument. It is for example the case of `read`, `write`, `fstat` etc. When a virtual machine monitor virtualize just some files it is useless to notify the tracer/monitor for fd relatied to non virtualized files.

When `KMVIEW_FLAG_FDSET` is set by `KMVIEW_SET_FLAGS ioctl`, all fd related calls are not notified to the tracer by default.

The tracer can add and delete file descriptors to the set of traced fd by using the following calls:

```
struct kmview_fd {
  pid_t kmpid;
  int fd;
};

ioctl(fd,KMVIEW_ADDFD,& {struct kmview_fd var});
ioctl(fd,KMVIEW_DELFD,& {struct kmview_fd var});
```

The tracer receives system call (and socket call) events only when the first argument file descriptor belongs to the set of traced fd.

The set of traced fd is automatically inherited during a clone/fork of a traced process.

There are two flags that can be specified together with `KMVIEW_FLAG_FDSET`: `KMVIEW_FLAG_EXCEPT_CLOSE` and `KMVIEW_FLAG_EXCEPT_FCHDIR`. These are notable exception that the tracer may specify. When `KMVIEW_FLAG_EXCEPT_CLOSE`, the close system call (as well as shutdown) always cause an event to the tracer. `KMVIEW_FLAG_EXCEPT_FCHDIR` has the same effect for fchdir. These two calls cause a change of the system state: some virtual machine

KMVIEW_FLAG_PATH_SYSCALL_SKIP.

When this flag is set (without any further configuration), all the system calls involving pathnames (like open, lstat or link) are not forwarded to the virtual machine monitor (tracer). This call minimizes the amount of messages to the virtual machine monitor when there are no active virtualizations of the file system.

It is possible to define exceptions for processes or for subtrees (prefixes for absolute pathnames)

When the tracer needs to receive the notification of the path system calls requested by a process, it runs the followin ioctl request:

```
ioctl(fd,KMVIEW_SET_CHROOT,kmpid)
```

This exception is inherited during a clone/fork of a traced process and can be undefined by:

```
ioctl(fd,KMVIEW_CLR_CHROOT,kmpid)
```

Kmview manages up to 64 prefixes for ghost mountpoints. A ghost mount is a very fast virtualization for file systems. Ghost mounted filesystems can be reached only by absolute pathnames. In fact, relative pathnames and symbolic links cannot be resolved by the kmview kernel module.

The profixes must be stored in struct ghosthash64:

```
#define GH_SIZE 64
#define GH_TERMINATE 255
#define GH_DUMMY 254

struct ghosthash64 {
unsigned char deltalen[GH_SIZE];
unsigned int hash[GH_SIZE];
};
```

all ghost mount pathnames get converted into hash signatures and stored in the hash array sorted by increasing pathname length. Each element of deltalen contains the difference betweeen the current element and the previous one. When a ghosthash64 contains less than 64 ghost mount hash values, `GH_TERMINATE` is stored in the deltalen element after the last (longest) element as a termination tag. The value of deltalen elements cannot exceed `GH_DUMMY`, fake dummy elements must be added when needed.

Kmview uses the following hash signature:

```
signature = 0
for each char in the path:
signature =  signature ^ ((signature << 5) + (signature >> 2) + char)
```

The following ioctl loads a new array of hash values in the kenrnel module:

```
ioctl(fd.KMVIEW_GHOSTMOUNTS,&gh);
```

where gh is a struct ghosthash64 variable.

e.g.:

```
struct ghosthash64 mygh={{2,2,6,GH_TERMINATE},{0x633,0x193694,0x5f710af7}};
ioctl(fd.KMVIEW_GHOSTMOUNTS,&mygh);
```

force the kmview module to forward to the tracer all the syscall using absolute path beginning by '/1','/1/2' or '/3/4567890'. In fact, the first path is 2 characters long, the second 4 (two more than the previous one), the third is 10 (+6). 0x633,0x193694,0x5f710af7 are the three hash values computed by the function above.

The file ghosthash.example.c in the source tree is an example of library for the management of ghosthash64 data structures.

Please note that this technique may generate false positives (although it happens very rarely: $1/2^{32}$), but it is very fast to select which system calls must be forwarded to the tracer.

KMVIEW_SYSCALLBITMAP (ioctl).

This ioctl selects the system calls for the tracer.

```
int bitmap[INT_PER_MAXSYSCALL];
ioctl(fd.KMVIEW_SYSCALLBITMAP,bitmap);
```

Each bit of bitmap corresponds to a specific system call, when a bit is set in the bitmap, the system call is *not* forwarded to the tracer. In other word this bitmap encodes the system calls which are *not* useful for the tracer.

Kmview.h file includes some inline functions to handle system call bitmaps:

Initialize bitmaps (all ones or all zeros)

```
static inline void scbitmap_fill(unsigned int *bitmap);
static inline void scbitmap_zero(unsigned int *bitmap);
```

Add/delete a bit in a bitmap

```
static inline void scbitmap_set(unsigned int *bitmap,int scno);
static inline void scbitmap_clr(unsigned int *bitmap,int scno);
```

Test if a bit in a bitmap is set:

```
static inline unsigned int scbitmap_isset(unsigned int *bitmap,int scno);
```

9.8 ▼*MView in education

- Write a *Mview module "hw" that adds a virtual file /hw containing "hello world" in the root directory. The file "/hw" must appear in the root directory.
- Write a module "hn" (hello net), which implements the address family 100. This family supports only a datagram service and is essentially a buffer, it stores all the messages sent that can be read by a recv later.
- Define some system calls and write test programs to use them (requires changed in *MVview code). For example implement:

  ```
  int rewrite(int fd, char *buf, int len);
  ```

  which reads from the file `len` bytes from the current position and writes the content of the buffer in the file, rewriting the read data. At the end the buffer must contain the data read from the file before overwriting them.

CHAPTER 10

*MView modules

10.1 ★umnet: virtual multi stack support

umnet is the View-OS module for multi-networking. In fact it supports the multi stack extension to the Berkeley Socket API named msockets.

Inside the *MView machine is possible to run standard networking programs like browsers, email readers, ssh, etc. These services will use virtual networks instead of the one provided by the kernel. Networking has here the broaden meaning of *any Berkeley socket supported service*, thus any protocol family or interprocess communication based on the Berkeley socket API can be virtualized while keeping the compatibility with existing applications.

umnet allows to test networking programs, to create a personal VPN limited to certain processes, and to have different VPNs run concurrently in different windows (processes).

From the user's point of view, umnet can be loaded in this way

```
$ um_add_service umnet
```

`umnet` can have trailing parameters to allow or deny the access to the networking stack provided by the OS. The default configuration permits the access to the existing socket/networking services.

For example:

```
$ um_add_service umnet,-all
```

or simply:

```
$ um_add_service umnet,-
```

denies the access to the pre-existing socket services for all the families of protocols. It is possible to select which protocol families must be permitted and which denied.

The following command allows all the protocols but AF_UNIX and AF_BLUETOOTH.

```
$ um_add_service umnet,-unix,bluetooth
```

When the first argument is a '-' the default is 'permit all' while if it is '+' the default is 'deny all'. So, it is possible to allow only ipv4 traffic with the following command:

```
$ um_add_service umnet,+ipv4
```

The first argument has the structure + or `-` plus a tag for a protocol or for a class of protocols. In the following arguments + or `-` is optional, when omitted the same permit or deny request of the previous argument is applied. The module recognizes the following tags for protocol families:

- `all` or nothing: all the protocols
- `u` or `unix` for AF_UNIX
- `4` or `ipv4` for AF_INET (ipv4)
- `6` or `ipv6` for AF_INET6 (ipv6)
- `n` or `netlink` for AF_NETLINK
- `p` or `packet` for AF_NETLINK
- `b` or `bluetooth` for AF_BLUETOOTH
- `i` or `irda` for AF_IRDA
- `ip` for all TCP-IP related protocols AF_INET, AF_INET6, AF_NETLINK and AF_PACKET
- `-#`n for the family number n.

Like many other View-OS modules umnet enables the mount operation for sub-modules prefixed by umnet

```
$ mount -t umnetnull none /dev/net/null
$ mount -t umnetcurrent none /dev/net/current
```

The former example define /dev/net/null to be a null network (socket/msocket calls fail, no networking is possible using /dev/net/null). The latter is a gateway to the default stack of the current process system. All the (m)sockets opened on /dev/net/current will use the networking stack provided for the current process in this moment.

umnet provides also the msocket backward compatibility tool named mstack.

```
$ mstack /dev/net/current ip link
```

gives the same output of ip link (there is a subtle but important difference: the mstack command causes the View-OS hypervisor – i.e. umview or kmview – to give the answer, using ip addr the answer comes directly from the kernel to the process bypassing View-OS).

mstack is a backward compatibility tool, not a protection tool. When several stacks are available it is possible to use mstack to switch from one stack to another.

mstack can have parameters:

- `h` : prints the mstack usage;

- `v` : sets the verbose mode on;

`o` *list* : defines the list of protocols. Without a -o option, mstack rede fines the default stack for all protocols families. The list of protocols may include the a comma separated sequence of the follow ing items: all, unix (or simply u), ipv4 (4), ipv6 (6), netlink (n), packet (p), bluetooth (b), irda (i), ip (which include all ip related protocols ipv4, ipv6, netlink and packet), #*n* where n is the number of protocol. Each item can be prefixed by + or - to specify whether the protocol/group of protocols must be added or removed from the set.

For example:

```
mstack -o ip /dev/net/lwip bash
```

starts a new bash which uses the stack /dev/net/lwip for ipv4 and ipv6 but not for the other protocols.

```
mstack -o -unix /dev/net/lwip bash
```

starts a new bash which uses the stack /dev/net/lwip for all protocols but AF_UNIX.

```
mstack -o +ip,-ipv6 /dev/net/lwip bash
```

starts a new bash which uses the stack /dev/net/lwip for ipv4, netlink, packet but not ipv6.

If a stack gets mounted on /dev/net/default, View-OS uses this stack as default.

```
$ mount -t umnetcurrent none /dev/net/default
```

defines the current network as the default network. The effect of this command is subtle: all the programs seem to access the network in the same way after this command as they did before it.

- Before the command the processes use the kernel stack directly: default networking has not been virtualized.

- After the command the networking calls get virtualized and View-OS (umview or kmview) uses the kernel stack to execute the calls.

This call

```
$ mount -t umnetnull -o perm none /dev/net/default
```

disables networking. The perm option denies the umount operation (the mount-point will always be busy), thus the operation is not undoable.

umnet submodules

umnetnull This module implements the null network: msocket and socket calls fail returning -1, errno EAFNOSUPPORT for all the protocols families. No networking is possible using umnetnull thus this submodule is used to deny networking.

```
$ mount -t umnetnull none /dev/net/null
$ mstack /dev/net/null ip link
Cannot open netlink socket: Address family not supported by protocol
$ mstack /dev/net/null telnet my.host.somedomain.it
Trying 10.20.30.40
telnet: Unable to connect to remote host: Address family not supported by proto
$ mstack /dev/net/null nc -u -l
Can't get socket : Address family not supported by protocol
```

umnetcurrent

The umnetcurrent network submodule provides a stack special file to access the stack currently used by the calling process.

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:1e:8c:b1:88:6f brd ff:ff:ff:ff:ff:ff
$ mount -t umnetcurrent none /dev/net/current
$ mstack /dev/net/current ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:1e:8c:b1:88:6f brd ff:ff:ff:ff:ff:ff
```

umnetlwipv6

Mounting a umnetlwipv6 stack means to start a lwipv6 stack and to associate the stack to a specific stack special file.

```
$ mount -t umnetlwipv6 none /dev/net/myip
$ mstack /dev/net/myip ip link
1: lo0: <LOOPBACK,UP> mtu 0
    link/loopback
2: vd0: <BROADCAST> mtu 1500
    link/ether 02:02:20:63:ef:06 brd ff:ff:ff:ff:ff:ff
```

Without any option, umnetlwip sets up one vde interface (provided there is a vde_switch running on the standard socket).

It is possible to start a lwipv6 stack with several interfaces. lwipv6 supports vde, tun or tap interfaces. The number, kind, and parameters for interfaces can be set by mount options (-o).

vd$n$ is the number of vde interfaces to be activated: *vd3* would be three vde interfaces;

vd$n$**=pathname** is the name of the vde switch that must be connected to the interface vd$n$. Interfaces are numbered from 0: defining vd$n$ means that all the interface vd0, vd$n-1$ will be automatically defined;

tp$n$ is the number of tap interfaces;

tp$n$**=interface_name** defines a tap interface with given name;

tn$n$ is the number of tun interfaces;

tn$n$**=interface_name** defines a tun interface with given name.

tuntap interfaces require that the user has write access to `/dev/net/tun`, or preallocated by `tunctl` usually restricted to root. Please note that when defining an interface with ordinal n all the interfaces $0, \ldots, n-1$ gets also defined. (`tp4=mytap4` defines also `tp0,...,tp3`)

umlwipv6 supports standard configuration tools based on `PF_NETLINK`, like those provided by the iproute, and even DHCP autoconfiguration clients.

Here are some examples:

```
$ mount -t umnetlwipv6 -o "vd0=/tmp/myswitch[4]" none /dev/net/yourip
$ mstack /dev/net/yourip ip link
1: lo0: <LOOPBACK,UP> mtu 0
    link/loopback
2: vd0: <BROADCAST> mtu 1500
    link/ether 02:02:62:84:74:06 brd ff:ff:ff:ff:ff:
```

vd0 is connected to the port number 4 of the switch /tmp/myswitch

```
$ mount -t umnetlwipv6 -o tn0=mytun none /dev/net/yourip
$ mstack /dev/net/yourip ip link
1: lo0: <LOOPBACK,UP> mtu 0
    link/loopback
2: tn0: <> mtu 0
    link/generic
```

The umnetlwipv6 stack defined by the special file /dev/net/yourip has a tun interface connected to mytun. A user can open a tun interface only if prviously authorized by the command:

```
# tunctl -u renzo -t mytun
```

where renzo is an example of username and mytun is the name of the tun interface used in our example.

Several mount options separated by commas allow to define multiple interfaces:

```
$ mount -t umnetlwipv6 -o "tn0=mytun,vd0=/tmp/myswitch[4]" none /dev/net/yourip
$ mstack /dev/net/yourip ip link
1: lo0: <LOOPBACK,UP> mtu 0
    link/loopback
2: vd0: <BROADCAST> mtu 1500
    link/ether 02:02:0b:d3:b2:06 brd ff:ff:ff:ff:ff:ff
3: tn0: <> mtu 0
    link/generic
```

umnetlink

This umnet submodule can be used to rename modules. It is named umnetlink as it recalls the idea of symbolic links when applied to stacks instead of files.

```
$ mount -t umnetcurrent none /dev/net/current
$ mstack /dev/net/current ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:1e:8c:b1:88:6f brd ff:ff:ff:ff:ff:ff
$ mount -t umnetlink /dev/net/current /dev/net/kstack
$ mstack /dev/net/kstack ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:1e:8c:b1:88:6f brd ff:ff:ff:ff:ff:ff
```

In the example above /dev/net/kstack becomes a symbolic umnetlink of /dev/net/current.

It is possible to link some address families. When the target of a mount is already an already existing stack special file, it is possible to use the previous stack for the other families (not linked by the current mount) by the option o, override. This example loads an lwipv6 stack on /dev/net/ourip redefines the default network as a current network and links ourip to the default network just for protocol families IPV4, IPV6, netlink and packet. With the option o all the other families gets inherited by the previous default stack i.e. current.

```
$ um_add_service umnet
$ mount -t umnetlwipv6 none /dev/net/ourip
$ mount -t umnetcurrent none /dev/net/default
$ mount -t umnetlink -o 4,6,p,n,o /dev/net/ourip /dev/net/default
$ ip link
1: lo0: <LOOPBACK,UP> mtu 0
    link/loopback
2: vd0: <BROADCAST> mtu 1500
    link/ether 02:02:11:fb:67:06 brd ff:ff:ff:ff:ff:ff
```

umnetlink uses the same options of umnet (see 10.1, above), plus 'o' or 'override'.

```
$ mount -t umnetlink -o +ip,override /dev/net/ourip /dev/net/default
$ mount -t umnetlink -o 4,6,p,n,o /dev/net/ourip /dev/net/default
$ mount -t umnetlink -o +ipv4 -o +ipv6 -o +p -o +netlink -o override
                          /dev/net/ourip /dev/net/default
```

10.2 ▲How to write umnet submodules

Each submodule must define a structure of type `struct umnet_operations` named `umnet_ops`. The filesystem type used in the mount operation must coincide with the name of the dynamic library. `umnet` calls submodule's init function when a new stack gets mounted. The last argument (`struct umnet *nethandle`) is opaque for the submodule. The init function can set its private date using `umnet_setprivatedata`. All the following operations naming the same stack (`msocket`, `fini`, `ioctlparms`), receive the same opaque nethandle and the submodule can retrieve its private date using `umnet_getprivatedata`. The msocket function of a submodule usually returns the index of an array as the file descriptor. This is useful for all the following calls including file descriptors to retrieve the open socket data directly in the array.

The function `supported_domain` returns one if the family is supported, zero otherwise. `supported_domain` is used by `msocket` using `SOCK_DEFAULT` and `PF_UNSPEC` to define the default stack for all defined domains. If undefined, it is supposed that the submodule implements all the families.

The minimal umnet submodule is umnetnull (provided in the `umnet_modules` directory). The code is shown in Figure 10.1. This module denies the access to the network (e.g. by mounting it pemanently on /dev/net/default).

All the socket support for *MView is converging to use recvmsg/sendmsg instead of recv/send, recfrom/sendto, read/write. The interface will be updated soon.

10.3 ♦umnet internals

Umnet is based on the msockets api.

Umnet keeps track of the default stacks for processes. Each process can define its default stack for each protocol family. Umnet maintain an array of pointers (one for each process) using the event subscription and notification service provided by *MView. Each element of this array (`defnet`) points to a `struct umnetdefault`, shared by all processes having the same default set. In fact a `struct umnetdefault` has a counter and an array of pointer to `struct umnet`, one for each family. The counter is set to zero for the first process. The information about stack defaults is inherited during forks/clones, the counter increased consistently, and descreased when a process quits or changes its default stack set. When the counter of processes becomes negative the `struct umnetdefault` is freed.

The check function finds the mountpoints for msockets and for file operation on mountponts. If the user code calls `socket` the check function uses the CHECKSOCKET tag and through the `defnet` it is possible to find the

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <errno.h>

#include "umnet.h"

int umnetnull_msocket (int domain, int type, int protocol,
    struct umnet *nethandle){
  errno=EAFNOSUPPORT;
  return -1;
}

int umnetnull_init (char *source, char *mountpoint, unsigned long flags,
    char *args, struct umnet *nethandle) {
  return 0;
}

int umnetnull_fini (struct umnet *nethandle){
  return 0;
}

struct umnet_operations umnet_ops={
  .msocket=umnetnull_msocket,
  .init=umnetnull_init,
  .fini=umnetnull_fini,
};
```

Figure 10.1: umnetnull: The null network submodule for umnet

default stack for the requested family. The check function recognizes also all the filesystem types with the "umnet" prefix for the mount system call.

Filetab is the table of opened file/socket. It is an array of pointer to `struct fileinfo` elements. The index of this array is used as local file descriptor (service file descriptor, sfd *Mview terminology). This table maps each open file to the correspondent umnet node. Each submodule can define its local file descriptor stored as nfd in `struct fileinfo`.

The `stat` for the mountpoint uses a new defined file type `S_IFSTACK`. the mtime and ctime is set to the mount time while atime is the time of the latest successful `msocket` operation.

10.4 ★umfuse: virtual file systems support

umfuse is the filesystem virtualization module: the name means User-Mode FUSE [28]. The idea behind FUSE is the implementation of filesystem support in userspace. A filesystem implementation for FUSE is just a program that uses a specific interface provided by the FUSE library. When a filesystem implemented via FUSE is mounted, every action to that filesystem is captured by a kernel module (developed for FUSE) and forwarded to the user level filesystem implementation.

umfuse keeps the same interface of FUSE. Moreover, FUSE modules are source level compatible with *umfuse* ones, with the only difference that *umfuse* modules are dynamic libraries instead of a program which uses the FUSE library. Thus, it is possible to compile a shared object from the same source code and obtain both FUSE and *umfuse* modules with minor changes that enable *umfuse* modules to manage several mounted filesystem at a time.

umfuse's architecture is organized in submodules, one for each file system supported, and the effects of the umfuse mount are limited to the process running inside the virtual machine.

A great number of virtual file system implementation have been created for FUSE, and they can be used by *umfuse* after recompilation (the V^2 team recompiled cramfs, encfs, sshfs for umfuse), because of the source code compatibility: there is a full list of on the FUSE web site. Through FUSE several other services can be accessed also as a file system, like ftp, ssh, cvs, and packet management for GNU-Linux distributions. Aside the FUSE project modules, a certain number of *umfuse* modules have been developed or ported/adapted by the V^2 team (using the `fuse.h` interface):

umfuseext2: used to mount ext2/ext3/ext4 filesystems;

umfusefsfs: an encrypted filesystem more scalable than NFS;

umfuseiso9660: based on libcdio, used to mount iso9660 filesystems and compressed iso images;

umfusefat: used to mount FAT filesystems;

umfusentfs-3g: used to mount NTFS filesystems;

umfusearchive: used to mount tar/cpio archives, supporting compression and rw access;

umfuseramfile: virtualization of one file.

The following is an example of *umfuse* usage, within *MView:

```
$ um_add_service umfuse
$ ls /tmp/mnt
$ mount -t umfuseext2 -o rw+ ~/tests/linux.img /tmp/mnt
$ ls /tmp/mnt
bin  boot  etc  lib  lost+found  mnt  proc  sbin  tmp  usr
$ df /tmp/mnt
Filesystem           1K-blocks      Used Available Use% Mounted on
-                         8192      4494      3289  58% /tmp/mnt
$
```

In the same way users can access iso or fat file systems:

```
$ mount -t umfuseiso9660 image.iso /tmp/mnt1
$ ls /tmp/mnt1
[contents of the cdrom image]
$ mount -t umfusefat -o ro image.fat /tmp/mnt2
$ ls /tmp/mnt2
[contents of the fat image]
```

The submodule `umfuseext2` requires the option `-o rw+` to give write access as the code is already under development. The user must be aware of the dangers before some error can corrupt his/her valuable file system contents. Umfuse is source compatible with fuse file system implementations. Fuse source code can be compiled as a dynamic library and used as umfuse submodule. Unfortunately fuse file systems have specific command line argument syntax. By default fuse calls its submodule's main program using a standard UNIX syntax, like mount:

```
command -o options source mountpoint
```

In fact typing `-o rw+,showcall` in the mount command of the previous example instead of `-o rw+`, the command line arguments get printed in the *Mview console:

```
FUSE call:
argv 0 = umfuseext2
argv 1 = -o
argv 2 = rw+
argv 3 = /home/v2user/tests/linux.img
argv 4 = /tmp/mnt
```

It is possible to specify parameters to add command line arguments or change the command line syntax:

`nosource` No image file should be specified in the commandline.

`pre=`*string* The string contains parameters that must be put before "-o options"

`post=`*string* The string contains parameters that must be added at the end

`format=`*string* This is the most powerful rewriting rule. If the fuse program needs a completely different structure of the command line format can be used: the format string is similar to that used in printf. `%O, %S, %M` descriptors are substituted in the call as follows: `%O=-o options`, `%S=`source, `%M=`mountpoint.

A umfuse mount operation masks all the contents of the mountpoint directory and substitute it with the contents of the mounted file system. It is possible to add one or several `except="`*directory*`"` mount options. Each directory listed as exception is like a"hole" in the mount mask. The pathname of the directories listed as exceptions are relative to the mountpoint (or the root of mounted file system). The command

```
mount -t umfusexxx -o except="/a/b" source mountpoint
```

mounts a file system from the image `source` to the mountpoint but the directory `mountpoint/a/b` is not mounted: it the same existing at the path `mountpoint/a/b` before the mount command.

Figure 10.2 shows an abstract example of the option *except* usage. In Figure 10.3 there is a practical use of the same option: umfuse permits to mount a filesystem as root and to hide in this way all the pre-existiong file system. However some programs need to access to file inside the /proc tree and may fail for a complete substitution of the file system.

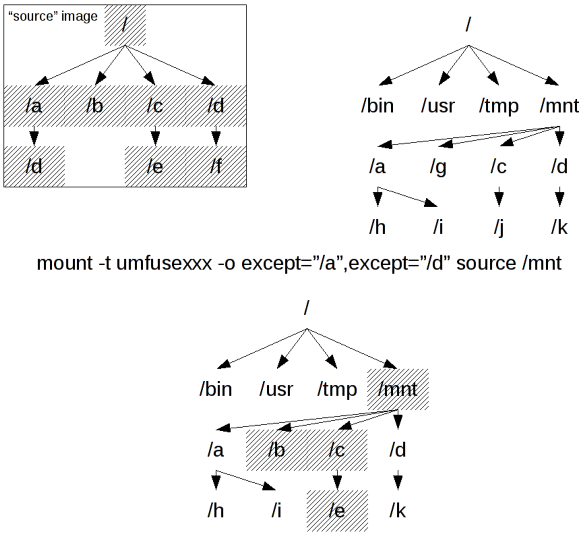

Figure 10.2: Use of the mount option *except* of umfuse

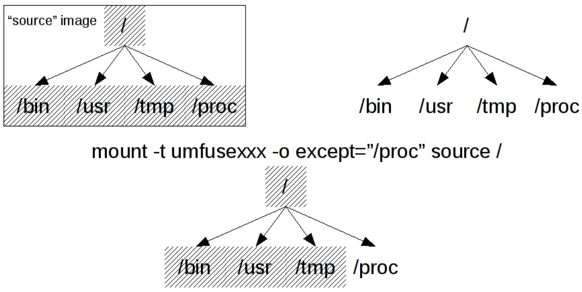

Figure 10.3: A common use of the option *except*

10.5 ★Some third parties umfuse submodules

While the support for ext2, iso, ntfs and fat has been designed by the V^2 project, we have tested other fuse modules like cramfs, sshfs and encfs. Actually the V^2 team in some cases cooperated with the original project to add new features: for example the auto endianess conversion for cramfs file system fuse module has been developed by V^2.

Encfs is a support for encrypted file systems. The following example shows the creation of an encrypted directory.

```
$ um_add_service umfuse
$ mkdir /tmp/clean /tmp/enc
$ mount -t umfuseencfs -o pre="" /tmp/enc /tmp/clear
[on the window where umview started from, there are some dialogues
about the password of the encrypted directory]
$ echo ciao > /tmp/clear/hello
$ ls /tmp/clear
hello
```

```
$ cat /tmp/clear/hello
ciao
$ ls /tmp/enc
qEV7deLtTqAZTo5uNu6tv0MN
$ cat /tmp/enc/qEV7deLtTqAZTo5uNu6tv0MN
0ew
$ umount /tmp/clear
$ ls /tmp/clear /tmp/enc
/tmp/clear:

/tmp/enc:
qEV7deLtTqAZTo5uNu6tv0MN
$
```

Sshfs uses a standard ssh connection to mount a remote file system. On the remote host runs a standard ssh daemon. It is possible to navigate the remote file system (with the permissions of the user who logs in using ssh), and copy files between systems using the cp. It is like a nfs mount. All the communication is encrypted as a ssh connection.

```
$ um_add_service umfuse
$ mount -t umfusessh remote_machine:/ /tmp/mnt
$ ls /tmp/mnt
[ls of remote_machine root dir]
$ umount /tmp/mnt
```

10.6 ▲How to write umfuse submodules

Umfuse submodules are source compatible with fuse modules. The source code must be compiled in two different ways for fuse or umfuse. Fuse file system implementations are executables. For example the `hello.c` test code included in the source tree of fuse is normally compiled as an executable:

```
gcc -o hello -D_FILE_OFFSET_BITS=64 hello.c -lfuse -ldl
```

The same source can be compiled for umfuse in this way:

```
gcc -D_FILE_OFFSET_BITS=64 -shared -nostartfiles -o umfusehello.so hello.o
```

If the file `umfusehello.so` is copied in the default directory for umfuse, e.g. `/usr/lib/umview/modules/` for debian packets, or in a directory included in `LD_LIBRARY_PATH`, it is now possible to use it.

```
$ um_add_service umfuse
$ mount -t umfusehello none /tmp/mnt2
$ ls /tmp/mnt2
hello
$ cat /tmp/mnt2/hello
Hello World!
$
```

```
static struct fuse_operations my_oper = {
  ...
  .init = my_init,
  ...
}

void *my_init(struct fuse_conn_info *conn)
{
  struct fuse_context *mycontext;
  mycontext=fuse_get_context();
  copy_of_my_user_data=mycontext->private_data;
  ....
  return new_private_data;
}

int main(int argc,char *argv[])
{
  ....
  /* set a variable "my_user_data" containing all the info for init */
  fuse_main(argc,argv,&my_oper,my_user_data);
  ....
}
```

Figure 10.4: user_data argument use for fuse/umfuse compatibility

Unfortunately most of the fuse modules support the mount of one partition of that type, they fail when somebody tries to mount the second partition. In fact, fuse modules have been designed to be executable so their designer often use global variables for data about the mounted partition. On the contrary umfuse use the same code as a library and the code is shared by all the mounted partitions having the same file system type.

The fuse/umfuse modules wrote by V^2 use private structures for each mounted partition thus these modules are fully compatible with fuse and umfuse and can mount several partitions at the same time.

Umfuse is compatible also with old versions of fuse (2.4 and 2.5), the development with newer versions (2.6 or later) is strongly suggested. In fact fuse project development team added in 2.6 a parameter for umfuse compatibility. Prior to version 2.6 there was no way to provide data to the init function but the use of global variables (e.g. some parameters coming from command line options). The solution was to keep a global variable just for the time from tha invocation of `fuse_main` to the starting of init. *MView is multithreading and this could lead to inconsistency for simultenous mount of the same file system types by different processes.

Fuse version 2.6 introduced an extra argument to `fuse_main` and `fuse_new` for this purpose. The last argument named `user_data` is passed in the `fuse_context` during the init phase. The source code shown in Figure 10.4 explains the correct use of this argument to create fuse/umfuse compatible modules.

All the data that the main need to pass to the init function must be reachable by a single pointer `my_user_data` in the example. This pointer is passed as `user_data` to the `fuse_main` function. The init function can read the

Main thread		**umfuse partition thread**
`umfuse_mount:`		
`...`		
`thread_start`	→	`startmain`
`cond_wait(startloop)`		*find the symbol "main"*
↓		*in the dynamic lib*
↓		↓
↓		`main:`
↓		*set up data structures*
↓		↓
↓		`fuse_main:`
↓		↓
↓		`fuse_loop:`
startloop signaled	←	`cond_signal(startloop)`
`return`		`cond_wait(endloop)`
		↓
call the `fuse_operations` *functions*		↓
		↓
`umfuse_umount2:`		↓
`cond_signal(endloop)`	→	*endloop signaled*
`thread_join`		`return`
↓		`return`
↓		*cleanup data stuctures*
↓		`return`
thread joins	←	`return` (*terminate thread*)

Figure 10.5: Umfuse: Program to dynamic library run-time conversion for fuse modules

data from main by getting the private data from the context, in the example `copy_of_my_user_data` points to the same address of `my_user_data`. When the `init` function exits, the private data is overwritten with the return value of the `init` function itself.

10.7 ♦umfuse internals

Umfuse is glue code between two different interfaces, *Mview module API and fuse.

The main difference is that fuse programs are designed as stand-alone programs, they have a `main` function while they are compiled as shared libraries in umfuse.

Figure 10.5 shows the technique (*trick*) used to run fuse programs as a library. Fuse main programs set up all the data for the file system to mount, and call `fuse_main` of `fuse_new`. These functions process the arguments in a different way, both call at the end `fuse_loop`, the main event loop of the program. `fuse_loop` waits for requests from the kernel, dispatches the request to the correspondent function defined in `fuse_operations` and returns the result. In case of unmount, `fuse_loop` terminates and the main program cleans up all the data structure, closes the external files preserving the consistency of the file system structure, and at last the fuse program terminates.

Umfuse use fuse structure *emulating* the call of the main function. It is another kind of virtualization. In fact, when mounting a umfuse partition, the dynamic library gets loaded. Umview creates a thread for each umfuse partition and wait for the main loop to start. The thread calls the main function. Everything in the thread happen like in the fuse program, thus the data structures gets set up properly before the main loop. `fuse_loop` for umview is not a loop at all. It just signals the main umview thread to restart and waits for a signal to terminate. The thread remains idle up to the umount of the partition. In the meanwhile all the `fuse_operations` can be called as standard library functions. When *MView receives the umount request for a umfuse partition, it sends the signal to the main loop and waits for the thread to terminate. The thread wakes up, returns to the main function of the module, clean up everything and terminates, waking up Kmview.

10.8 ♦Some notes on umfuse modules internals

fuseext2 and fuseiso are extremely compact modules. The implementation of the file systems has been provided by standard libraries: libext2fs for ext2, libiso9660 for iso, and libz for compression. The fuse/umfuse code implement an interface between the fuse API and the libraries.

fusefat, on the contrary, has been developed completely in the V^2 project. The code for fusefat include a library named libfat to decode and access the fat format and the fuse module. The library, hence the fuse module, supports FAT12, FAT16 and FAT32 formats.

10.9 ★umdev: virtual devices support

umdev is the module used to create virtual devices: processes running inside *MView can access virtual devices with the same semantic as if they were real. *umdev* implements special files, able to process specific `ioctl` control calls. Like umfuse, *umdev* is based on a submodules architecture, where each submodule provides support for each kind of virtual device.

The mount operation for umdev submodules supports the following options:

`debug`: activate debug logging on the console;

`char`: mountpoint is a char special file;

`block`: mountpoint is a block special file;

`major=`*major number*: specify the major number;

`minor=`*minor number*: specify the minor number;

`mode=`*mode*: permissions on the special file;

`uid=`*uid*: owner of the special file;

`gid=`*gid*: group ownership of the special file;

`nsubdev=`*num* number of supported subdevices.

A umdev mount operation defines a file and a device address: the open operation of the mountpoint give access to the device as well as opening any other special file in the file system with matching major/minor number of the mounted device. If a device gets mounted on an existing device it gets the same major/minor number of the virtualized special file.

The virtual devices currently supported are:

umdevmbr: manages the Master Boot Record partition table, allowing disk partitioning and access to each partition. If the disk image is mounted on /dev/hda, *umdevmbr* will also define /dev/hda1, /dev/hda2, and so on, with the same naming convention used by the kernel. It is also possible to run any command on the mounted partitions (mkfs, fsck, ...), or mount other filesystems using umfuse, like in the following example:

```
$ um_add_service umdev.so
$ hexdump -C test.img
0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
1388000
$ mount -t umdevmbr test.img test_hd
$ fdisk -l test_hd
Disk test_hd: 2 cylinders, 255 heads, 63 sectors/track
Units = cylinders of 8225280 bytes, blocks of 1024
bytes, counting from 0

Device Boot Start   End  #cyls  #blocks   Id  System
test_hd1         0+    1      2-   16033+  83  Linux
test_hd2         0     -      0        0    0  Empty
test_hd3         0     -      0        0    0  Empty
test_hd4         0     -      0        0    0  Empty

$ mkfs.ext3 test_hd1
mke2fs 1.40-WIP (14-Nov-2006)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
4016 inodes, 16032 blocks
801 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=16515072
2 block groups
8192 blocks per group, 8192 fragments per group
2008 inodes per group
Superblock backups stored on blocks:
       8193
Writing inode tables: done
Creating journal (1400 blocks): done
Writing superblocks and filesystem accounting
information: done
```

```
$ um_add_service umfuse
$ mount -t umfuseext2 test_hd1 /mnt
$ umount /mnt
$ umount test_hd
```

umdevramdisk: this submodule is used to create virtual ramdisk devices. As linux ramdisks do, this submodule takes a segment of the active system memory and makes it available as a virtual block device. Unlike linux ramdisks, inside *MView it is possible to create virtual ramdisks at user level. Here is an example:

```
$ um_add_service umdev
$ mount -t umdevramdisk -o size=100M,mbr none test_rd
$ /sbin/mkfs.ext2 test_rd
[...]
$ um_add_service umfuse
$ mount -t umfuseext2 -o rw+ test_rd /tmp/mnt
$ ls /tmp/mnt
lost+found
```

umdevtrivhd: is a simplified example of a ramdisk implemented as a character device. Once mounted, it is possible to make standard I/O operations as if it were a real character device, up to 64k;

umdevtap: this submodule provides tun/tap device interface.

```
$ um_add_service umdev
$ mount -t umdevtap /var/run/vde.ctl /dev/net/tun
$ ... start you favourite virtual machine or tunnelling tool using tap
```

umdevtap currently connects all the interfaces to the same switch. The syntax to support connections to several switches, to define ports, and permissions on ports has not decided yet.

umdevvd: This module uses the VBoxDD library to access disk images for virtual machines. It supports VDI, VHD and VDMH disks, disk formats used by VirtualBox, VirtualPC and VMware, respectively.

```
$ um_add_service umdev
$ mount -t umdevvd .VirtualBox/HardDisks/test.vdi /dev/hdx
$ mount -t umdevmbr /dev/hdx /dev/hdy
$ um_add_service umfuse
$ mount -t umfuseext2 -o ro /dev/hdy1 /mnt
```

In the exaple above, a disk created by VirtualBox is mounted on /dev/hdx, then /dev/hdx is mounted on /dev/hdy as a mbr disk and finally the first partition is mounted on /mnt.

umdevnull: this is a test submodule, that defines a virtual null device (similar to /dev/null). A debug message is emitted for each I/O request for the virtual device.

10.10 ▲How to write umdev submodules

The interface for umdev submodules is defined is the include file `umdev.h`.

Each submodule must define a global variable named `umdev_ops` of type `struct umdev_operations`. When a device is mounted umdev calls the `init` function of the correspondent submodule, `fini` for the unmount. Processes can open, use and close communication sessions to the device using them as they were files. This is a fundamental idea of UNIX. Mounted virtual devices appear as special files, and processes can access virtual devices as the pseudo-files of `/dev`.

All the `struct umdev_operations` functions referring to the entire device has an opaque parameter `struct umdev *devhandle`. It is possible to set up and access private data of the device implementation using the functions:

```
void umdev_setprivatedata(struct umdev *devhandle, void *privatedata);
void *umdev_getprivatedata(struct umdev *devhandle);
```

Functions of `struct umdev_operations` referring to files opened by processes (communication sessions) have a parameter of type `struct dev_info *`. This parameter contains the `devhandle` field plus some flags and a 64 bit integer that the implementation can use to access the data of the file.

The source code of `umdevnull` is listed in Figure 10.6. It is a virtual `/dev/null`, the only difference is that it prints on the *MView console a log of all the open/read/write/close operations. All the `struct umdev_operations` functions return non-negative results or negative errors. `lseek` use a 64 bit offset as its return value thus seek is limited to 8 exabyte.

It is possible for a device to define subdevices.

```
void umdev_setnsubdev(struct umdev *devhandle, int nsubdev);
int umdev_getnsubdev(struct umdev *devhandle);

dev_t umdev_getbasedev(struct umdev *devhandle);
```

If a device implementation sets the number of its subdevices, it will receive the requests for devices with the same major number and minor numbers in the range from its minor number and its minor number plus `nsubdev`. Several devices have subdevices, like partitions for disks, or tty connections or different registration parameters for tapes or floppies. `umdev_getbasedev`provide the base device for subdevices and can be used to retrieve the actual subdevice number:

```
functionxxx(...., dev_t device, ..., struct dev_info *di)
{
  int subdevice=minor(device)-minor(umdev_getbasedev(di->devhandle));
}
```

10.11 ♦umdev internals

umdev is just an interface between *Mview module API and device submodules. The check function has a set of rules to check for special files, and capture access to virtualized devices under other pathnames but the mountpoint. The

```
#include <stdio.h>
#include "umdev.h"
#include <config.h>

static int null_open(char type, dev_t device, struct dev_info *di)
{
  printf("null_open %c %d %d flag %x\n",type,major(device),minor(device),di->flags);
  return 0;
}

static int null_read(char type, dev_t device, char *buf, size_t len, loff_t pos,struct dev_info *di)
{
  printf("null_read %c %d %d len %d\n",type,major(device),minor(device),len);
  return 0;
}

static int null_write(char type, dev_t device, const char *buf, size_t len, loff_t pos, struct dev_info *
{
  printf("null_write %c %d %d len %d\n",type,major(device),minor(device),len);
  return len;
}

static int null_release(char type, dev_t device, struct dev_info *di)
{
  printf("null_release %c %d %d flag %x\n",type,major(device),minor(device),di->flags);
  return 0;
}

struct umdev_operations umdev_ops={
  .open=null_open,
  .read=null_read,
  .write=null_write,
  .release=null_release,
};
```

Figure 10.6: umdevnull.c: the simplest virtual device

addressing of devices for `umdev_operations` functions is composed by the type of device (**c** or **b**) and a `dev_t` field (inlcuding the major and minor number).

10.12 ★umbinfmt: interpreters and foreign executables support

This module implements at user level the Linux kernel feature named *binfmt_misc* [18], that associates interpreters with executables and scripts, depending on some properties like file extensions, magic numbers or pattern matching. Using *umbinfmt* is possible to run binary executables compiled for machine A on an incompatible computer architecture B. In order to achieve this, the user must associate which interpreter has to be invoked with which binary.

The module works by overlaying a virtual `/proc` interface is compatible the real *binfmt_misc* `/proc` interface resides. The umbinfmt module must be loaded into the *MView chain of services, then *umbinfmt* virtual filesystem is usually mounted in `/proc/sys/fs/binfmt_misc`, to preserve compatibility with existing applications.

```
$ um_add_service umbinfmt.so
umbinfmt init

$ mount -t umbinfmt none /proc/sys/fs/binfmt_misc

$ ls -l /proc/sys/fs/binfmt_misc
--w------- 1 root root 0 Jan  1  1970 register
-rw-r--r-- 1 root root 0 Jan  1  1970 status
```

Once the interface is mounted, the new executable file formats are registered through the file `/proc/sys/fs/binfmt_misc/register`, where the user has to specify the interpreter in the form `:name:type:offset:magic:mask:interpreter:` .

The following example shows how it is possible to recognize foreign executables and run the right interpreter. Suppose the qemu-ppc emulator is installed and there is a ppc executable in your system.

```
$ file /tmp/busybox-ppc
/tmp/busybox-ppc: ELF 32-bit MSB executable, PowerPC or cisco 4500...
$ /tmp/busybox-ppc uname -a
bash: /tmp/busybox-ppc: cannot execute binary file
$ qemu-ppc /tmp/busybox-ppc pwd
/tmp
```

In a umview machine it is possible to configure um_binfmt to call qemu-ppc automatically for all ppc executables.

```
$ um_add_service umbinfmt.so
umbinfmt init
$ mount -t umbinfmt none /proc/sys/fs/binfmt_misc
echo   ':ppc:M::\x7fELF\x01\x02\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02
\x00\x14:\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x
fe\xff\xff:/usr/bin/qemu-ppc:' > /proc/sys/fs/binfmt_misc/register
$ qemu-ppc /tmp/busybox-ppc pwd
/tmp
```

Please note that the echo command must be typed on a single line. For a complete description of the binfmt_misc syntax the reader may refer to the file `Documentation/binfmt_misc.txt` included in any recent Linux kernel source hierarchy.

10.13 ★ummisc: virtualization of time, system name, etc.

This is a generic module for *miscellaneous* virtualization: like *umfuse*, it has a submodules architecture and each module can be loaded via the `mount` system call. Currently, two submodules are available for *ummisc*:

ummiscuname can be used to modify the hostname and domainname for the virtualized processes: it is useful in the case of multiple *MViews running on the same machine, because the easier distinguishing mean between them is the *hostname*. An example of *ummiscuname* use:

```
$ um_add_service ummisc
$ mount -t ummiscuname none /tmp/uname
$ ls /tmp/uname
domainname  machine  nodename  release  sysname  version
$ uname -n
v2host
$ # the following commands are interchangeable
$ echo "virtuous" > /tmp/uname/nodename
$ hostname virtuous
$ uname -n
virtuous
```

As shown in the example, after loading the module in the services chain, a `uname` filesystem structure is loaded: this structure is composed by several files, as defined in the `struct utsname` of **uname (2)**. Writing to these files can change the correspondent field.

ummisctime allows to change the frequency and offset of the system clock in the view: the *offset* is the difference (in seconds) between the time perceived by the processes running in the view and the system time, while the *frequency* is the rate of the clock: 1 means that the frequency of the virtual clock is 1hz, 2 means 2hz (the virtual clock runs faster) and 0.5 means 0.5hz (it runs slower). Offset and frequency can be changed by hand, editing the files. When the frequency is changed, the offset is automatically adjusted. The following example shows a negative frequency (the clock goes back in time!):

```
$ mount -t ummisctime none /tmp/time
$ ls /tmp/time
frequency  offset
$ echo -1 > frequency
$ date
Wed Nov 28 12:28:53 CET 2007
$ date
Wed Nov 28 12:28:52 CET 2007
$ date
Wed Nov 28 12:28:51 CET 2007
$ echo -1 > frequency
$ date
Wed Nov 28 12:28:51 CET 2007
$ date
Wed Nov 28 12:28:52 CET 2007
```

When the user changes again the frequency the time flows with the new pace from the time value of the change instant. It is a nice experiment to run several xclocks

```
$ xclock -update 1 &
```

in a umview machine and another outside umview and see that the arm of seconds of the clocks turn at different speeds.

10.14 ▲How to write ummisc submodules

Ummisc is different from the other modules as it is has been built to define a virtual file system for the virtualization parameters (it is similar to `/proc`). The interface defined in `ummisc.h` is minimal.

```
#define UMMISC_GET 1
#define UMMISC_PUT 0
struct fsentry {
  char *name;
  struct fsentry *subdir;
  loff_t (*getputfun)(int op,char *value,int size,struct ummisc *mh,int tag,char *pat
  int tag;
};

struct ummisc_operations {
  struct fsentry root;
  void (*init) (char *path, unsigned long flags, char *args,struct ummisc *mh);
  void (*fini) (struct ummisc *mh);
};
```

As usual the module must define a global variable named `ummisc_ops` of type `struct ummisc_operations`. This structure has only three fields: the root of the file system structure, a constructor of a desctructor for mount and umount operations respectively.

The constructor has four parameters: the mountpoint (path), the mount flags and args and an opaque pointer `mh`.

Like for other submodules, the opaque handler can be used to keep the private data for the submodule using two specific functions:

```
void ummisc_setprivatedata(struct ummisc *mischandle,void *privatedata);
void *ummisc_getprivatedata(struct ummisc *mischandle);
```

The virtual file system structure is built with arrays of `struct fsentry` each array is a directory of the virtual file system. Each elemet is virtual file, with a name, if it is the name of a subdirectory a pointer to the `struct fsentry` array of the subdirectory or the pointer of a function to read/write data (`getputfun`), and a tag used to share the same function for several virtual files.

`getputfun` has several parameters:

`op` the operation `UMMISC_GET` it the process reads data, `UMMISC_PUT` to write data.

`value` is a string of `MISCFILESIZE` (4096) butes of maximum length. `getputfun` works must read/write data on this strings.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <string.h>
#include "ummisc.h"
char buf[MISCFILESIZE];
int val[2];
static loff_t gp_file(int op,char *value,int size,struct ummisc *mh,int tag,char *path);
static loff_t gp_value(int op,char *value,int size,struct ummisc *mh,int tag,char *path);

struct fsentry valuesdir[] = {
  {"value1",NULL,gp_value,0},
  {"value2",NULL,gp_value,1},
  {NULL,NULL,NULL,0}};
struct fsentry fseroot[] = {
  {"asciifile",NULL,gp_file,0},
  {"values",valuesdir,NULL,0},
  {NULL,NULL,NULL,0}};
struct ummisc_operations ummisc_ops = {
  {"root",fseroot,NULL,0},
  NULL,
  NULL,
};

static loff_t gp_file(int op,char *value,int size,struct ummisc *mh,int tag,char *path) {
  switch(op) {
    case UMMISC_GET:
      strncpy(value,buf,size);
      return strlen(value);
    case UMMISC_PUT:
      value[size]=0;
      strncpy(buf,value,size);
      return size;
    default:
      return EINVAL:
  }
}

static loff_t gp_value(int op,char *value,int size,struct ummisc *mh,int tag,char *path) {
  switch(op) {
    case UMMISC_GET:
      snprintf(value,size,"%d\n",val[tag]);
      return strlen(value);
    case UMMISC_PUT:
      value[size]=0;
      val[tag]=atoi(value);
      return size;
    default:
      return EINVAL:
  }
}
```

Figure 10.7: miscdata.c: a misc submodule with a virtual file system for parameters

`size` is for `UMMISC_PUT` the length in bytes of the value.

`tag` and `path` are the tag and the path field of the `struct fsentry` element.

The value of the virtual file is computed once when the file is opened and stored when the file is closed. All read/write/lseek operation have access to the contents stored as a char array.

The source code of `miscdata.c` is listed in Figure 10.7. This submodule defines in the mounting point a directory containing a ascii file named `asciifile` and a subdirectory `values` which contains two numeric files `value1` and `value2`

The module can be compiled as a shared library:

```
gcc -shared -o ummiscdata.so miscdata.c
```

and tested in a *Mview machine:

```
$ um_add_service ummisc
$ mount -t ummiscdata none /tmp/mnt
$ cd /tmp/mnt
$ ls
asciifile  values
$ cat asciifile
$ echo "She sells sea shells" > asciifile
$ echo "on the sea shore" >> asciifile
$ cat asciifile
She sells sea shells
on the sea shore
$ cd values
$ ls
value1  value2
$ echo 10 > value1
$ echo 20 > value2
$ cat value*
10
20
$ echo ciao > value2
$ cat value*
10
0
$
```

Ummisc module can define handlers for the following system calls:

time related calls: gettimeofday, settimeofday, adjtimex, clock_gettime, clock_settime, clock_getres.

host identification calls: uname, gethostname (where definded), sethostname, getdomainname (where defined), setdomainname.

user mgmt calls: getuid, setuid, geteuid, setfsuid, setreuid, getresuid, setresuid, getgid, setgid, getegid, setfsgid, setregid, getresgid, setresgid.

priority related calls: nice (where defined), getpriority, setpriority.

process id related: getpid, getppid, getpgid, setpgid, getsid, setsid.

Some of the calls have a note saying “where defined”. Some system calls are defined only on some architectures, so ummisc uses those calls only where defined.

If a module wants to redefine a system call it must simply include a function with the name of the system call prefixed by “misc_”.

Figure 10.8 lists the source code of `fakeroot.c`. This very simple submodule changes the answer of getuid and geteuid. The module can be compiled with the following command:

```
#include <unistd.h>
#include <sys/types.h>
#include "ummisc.h"

uid_t misc_getuid(void){
  return 0;
}

uid_t misc_geteuid(void){
  return 0;
}

struct ummisc_operations ummisc_ops = {
  {"root",NULL,NULL,0},
  NULL,
  NULL,
};
```

Figure 10.8: fakeroot.c: a simple misc submodule

```
gcc -shared -o ummiscfakeroot.so fakeroot.c
```

And tested, in a *MView machine:

```
$ um_add_service ummisc
$ mount -t ummiscfakeroot none /tmp/mnt
$ bash
# echo $UID
0
#
```

10.15 ♦ummisc internals

ummisc has two peculiarities with respect to the other modules: the management of a virtual file system for parameters and the automatic join of system call implmentation functions (if the module has a function whose name is `misc_xxx` it is used as the implementation of system call `xxx`).

All system calls handled by ummisc are contextless: they are not related to pathnames, file descriptors, file system types, address families etc. They have been designed to change or read the execution environment (like time, system name, user id).

The management of the virtual file system for the I/O of module parameters is implemented with several functions:

- `searchentry`: it searches a pathname in the tree composed by fsentry arrays. It has been implemented as a recursive search by the fuction `recsearch`.

- `dirsize`: this function computes the size of the `struct dirent64` array needed to store a virtual directory.

- `dirpopulate`: this function gets called by the first invocation `ummisc_getdents64` on an open directory (open file of type directory). It generates a `struct dirent64` array containing the whole directory. All subsequent calls to `ummisc_getdents64` retrieves a section of the array. The array is freed when the directory is closed.

The automatic joining of the system call implementation function is in the source file `ummiscfun.c`. There is a table at the beginning of the file that maps each system call with the sorrespondent function name. The function `getfun` uses `dlsym` to locate the symbol in the submodule's code.

10.16 ★ViewFS

ViewFS is a module that virtualizes the file system structure, but is currently under development. Its main features are:

- possibility to hide files, directories, hierarchies;
- possibility to change mode of files and directories without affecting the underlying filesystem;
- permissions redefinition;
- copy-on-Write[1] access to files (and subtrees), to allow write access to read-only entities;
- merging of real and virtual directories.

ViewFS allows the user to give a new view of the filesystem to processes: this view is made of some kind of *patchwork* of files, taken from the existing filesystem. With this module, potentially dangerous modifications of the filesystem can be tested in a safe virtual environment, because the real filesystem remains intact.

The module `viewfs` is a basic implementation of viewfs.

```
$ um_add_module viewfs
```

Viewfs has no submodules. It supports four different modes selectable by options: move, merge, cow, mincow.

mode	source tree	existing tree at target dir
move	read-write	(inaccessible)
merge	read-only (EROFS)	read-write (for non-merged files)
cow	read-write	read-only (copied when written)
mincow	read-write	read-write (when permitted)

- `-o move` files or directories are simply moved across the file system. For example the following sequence hides all the home directries but one.

[1] Copy-on-write is an optimization strategy: if multiple callers ask for resources which are initially indistinguishable, pointers are given to the same resource. This fictional behaviour is maintained until a caller tries to modify its "copy" of the resource. At this point, a *true* private copy is created in order to prevent the changes becoming visible to everyone else. This is realized in a transparent way towards the callers. The main advantage is that a copy is not necessary unless the caller has to make modifications.

```
$ ls /home
otherusr1 otherusr2 v2user
$ mkdir /tmp/home
$ mkdir /tmp/home/v2user
$ um_add_service viewfs
$ mount -t viewfs -o move /home/v2user /tmp/home/v2user
$ mount -t viewfs -o move /tmp/home /home
$ ls /home
v2user
$
```

This is the default mode, thus `-o move` can be omitted.

The test module unreal (see 9.4) can be created by viewfs:

```
$ mount -t viewfs / /unreal
$ ls /unreal
bin   boot .....
$ ls /unreal/unreal
ls: cannot access /unreal/unreal: No such file or directory
$ mount -t viewfs / /unreal
$ ls /unreal/unreal
bin   boot .....
$ ls /unreal/unreal/unreal
ls: cannot access /unreal/unreal/unreal: No such file or directory
$
```

- `-o merge` viewfs unifies the file system tree of the source file or directory with the tree at the mount point. File and directories in one of the tree are visible in the merged view. When the same path is defined in both trees viewfs returns the file or directory defined in the source tree. In the following example two directories (`src` and `dest`) are merged together. In the example `2` is at the same time an empty directory in `src` and a file in `dest`. In the resulting merged file system the file of `dest` gets hidden by the directory in `src`.

```
$ ls -RF src dest
dest:
a/  b/  c/  f  g
dest/a:
a1  a2
dest/b:
b1  b2
dest/c:
src:
b/  d/  e/
src/b:
b2/  b3
src/b/b2:
src/d:
```

```
src/e:
$ um_add_service viewfs
$ mount -t viewfs -o merge src dest
$ ls -R dest
dest:
a/  b/  c/  d/  e/  f  g
dest/a:
a1  a2
dest/b:
b1  b2/  b3
dest/b/b2:
dest/c:
dest/d:
dest/e:
$ rmdir src/b/b2
$ ls -F dest/b
b1  b2  b3
$
```

At the end of the example there is the removal of the directory `src/b/b2`. When `src/b/b2` directory disappears, the pre-existing `dest/b/b2` file returns visible. This behavior may appear counter intuitive, a file continue to exist after it has been removed, but this is the result of a pure file system merge (in the overlay model of View-OS mount). Merge is commonly used to add files and directories in a read only way. In this way the consistency is maintained as removal actions are denied.

- `-o cow`. This is the copy on write mode. The file system structures get merged (in the same way seen for the option `-merge` above. Files and directories in the mount point tree are not modified, all the changes takes place in the src subtree. It is possible to remove files and directories. When a file or a directory gets deleted it disappears (if some file or directory do exist under the same file in the mount point subtree it is hidden).
- `-o mincow`. The minimal copy on write support is a transparent service for all permitted operations, it becomes a copy on write service only for unaccessible files and directories.

```
$ mkdir /tmp/newroot
$ um_add_service viewfs
$ mount -t viewfs -o mincow /tmp/newroot /
$ echo ciao >>/etc/passwd
$ tail /etc/passwd
.....
v2user:x:1000:1000::/home/v2user:/bin/bash
ciao
$ rm /etc/passwd
$ ls /etc/passwd
ls: cannot access /etc/passwd: No such file or directory
$
```

At most one of the options of the list above can be set, as the modes are mutually exclusive. The same source directory can be mounted later with a different option. For example it is possible to modify a filesystem using viewfs-mincow and then mount the same modification in merge mode. In this way it is possible to (virtually) install a set of programs or update a system. When the directory is mounted later as viewfs-merge the programs will be seen as installed or the system updated (but no further modification are possible on that source dir).

It is possible to add `-o except=....` option in the same way explained for unfuse (10.4).

`viewfs` supports also the `-o renew` mounting option. Renew is like a re-mount of the same already mounted file system, making visible new changes happened inside the source filesystem tree.

In the following example renew is needed otherwise tmp1 cannot be accessed through dest.

```
$ um_add_service .libs/viewfs
$ cd /tmp
$ mount -t viewfs /tmp/tst2 /tmp/src
$ mount -t viewfs -o merge /tmp/src /tmp/dest
$ ls dest
a  b  c  ciao  f  g  tst2
$ mount -t viewfs -o merge /tmp/tst1 /tmp/src
$ ls dest
a  b  c  ciao  f  g  tst2
$ mount -t viewfs,renew -o merge /tmp/src /tmp/dest
$ ls dest
a  b  c  ciao  f  g  tst1  tst2
$
```

ViewFS supports virtual ownership and permission when the mount option `-o vstat` is set.

Virtual installation of software by ViewFS

View-OS allows the virtual installation of software. The following examples show hot to (virtually) install some Debian packets by ViewFS. It is possible to install software as users, there is no need to log in as root or to execute sudo commands.

Let us suppose that a user wants to try the tinyirc application, which we suppose it is not installed in the system:

```
$ tinyirc
bash: tinyirc: command not found
```

In a view-os machine our user can install it:

```
$ um_add_service viewfs
$ mkdir /tmp/newroot
$ viewsu
# mount -t viewfs -o mincow,except=/tmp,vstat /tmp/newroot /
```

some files must be deleted, recreated to avoid warnings (it is not possible to virtually access really protected files, but we can delete them or change them to read-write mode)

```
# rm -rf /root/.aptitude
# mkdir /root/.aptitude
# touch /root/.aptitude/config
# touch /var/cache/debconf/passwords.dat
```

Now the packet can be installed in the standard way:

```
# aptitude install tinyirc
Reading package lists... Done
Building dependency tree
Reading state information... Done
...
Unpacking tinyirc (from .../tinyirc_1%3a1.1.dfsg.1-1_i386.deb) ...
Processing triggers for menu ...
Processing triggers for man-db ...
Setting up tinyirc (1:1.1.dfsg.1-1) ...
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
Writing extended state information... Done
Reading task descriptions... Done
# exit
$ tinyirc
TinyIRC 1.1 Copyright (C) 1991-1996 Nathan Laredo
This is free software with ABSOLUTELY NO WARRANTY.
....
```

it works.

It is possible to install more complex packets with all their libraries. Our user can install also gnome-games.

```
$ viewsu
# aptitude install gnome-games
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
Reading task descriptions... Done
The following NEW packages will be installed:
ggzcore-bin{a} gnome-games libggz2{a} libggzcore9{a} libggzmod4{a}
libgnomeprint2.2-0{a} libgnomeprint2.2-data{a} libgnomeprintui2.2-0{a}
libgnomeprintui2.2-common{a} libsdl-mixer1.2{a} libsmpeg0{a}
python-bugbuddy{a} python-gnomeprint{a}
0 packages upgraded, 13 newly installed, 0 to remove and 652 not upgraded.
```

```
Need to get 2603kB of archives. After unpacking 9933kB will be used.
Do you want to continue? [Y/n/?]
```

Several Debian packets need to be installed, and can be installed just by typing 'Y' exactly in the same way used by a sysadm to do the same operation.

```
Writing extended state information... Done
Get:1 http://www.debian.org unstable/main libggz2 0.0.14.1-1 [73.3kB]
...
# exit
$ gnomine
```

... our user can enjoy the most important application of the history of personal computers: the minekeeper board game.

10.17 ♦ViewFS internals

ViewFS store the modification of the target directory in the source directory. A virtual file that is accessible at a relative path P from the target directory is stored at the same path P from the suorce directory.

ViewFS tries to keep the maximum consistency between the source tree hierarchy and what appears when the source directory is moved to or merged with the destination.

There are however operation which may be forbidden: file or directory removal, change of ownership/permission, creation of special files.

ViewFS uses a hidden directory named `.-` to store the changes that cannot be applied on the visibile tree. When an empty file is stored inside the hidden directory, the file or directory having the same pathname in the target directory disappears. The idea can be catched by the name of "wipe-out files".

In this way it is possible to (virtually) remove files in cow and mincow mode: viewfs creates a wipe-out file. When a virtual file or directory gets deleted and a file or a directory exists in the same position exists in the target tree a wipe-out file is create to prevent the underlying file or directory to appear.

When permissions, ownership or device major/minor numbers cannot be stored in the source tree, a supplementary info file is created in the hidden directory at the same relative path adding a trailing escape char (ascii 255). The trailing char is needed to store information on directories. Info file contains the missing changes on permissions, user and group ownership and device specification is a endianess-independent representation: mode,owner,group and rdev field of the stat structure are stored as hexadecimal strings. When a field is left blank it means that the stat of the visibile file is correct and thus can be left unmodified.

10.18 ★Umview/kmview as login shells

`umview` and `kmview` can be used as login shells. The following chunk of a /etc/passwd file defines two users using kmview and umview respectively:

```
testkm:x:1003:1003:test KM,,,:/home/testvm:/usr/local/bin/kmview
testum:x:1004:1004:test UM,,,:/home/testum:/usr/local/bin/umview
```

This feature needs also a new configuration file named `/etc/viewospasswd`. This latter file has two fields per line separated by colons (:) as usual for many configuration files. The first field is the username, the second is the command kmview/umview must run.

```
testkm:/bin/bash --norc --noprofile /home/testkm/.startviewos
testum:/bin/bash --norc --noprofile /home/testum/.startviewos
```

In this example the startup scripts are in the users' home dir. This gives users the flexibility to redefine their view. If the feature is used to create security contraints the commands or the scripts should be stored elsewhere and protected from user changes.

The following example of startup script (.startviewos in the example above) gives the user his/her own ip address on vde:

```
#!/bin/bash --norc
/usr/local/bin/um_add_service umnet
/bin/mount -t umnetlwipv6 none /dev/net/lwip
/usr/local/bin/mstack /dev/net/lwip /bin/ip link set vd0 up
/usr/local/bin/mstack /dev/net/lwip /bin/ip addr add 192.168.10.1/24 dev vd0
exec /usr/local/bin/mstack /dev/net/lwip /bin/bash -l
```

(use the permanent option for mount to deny unmount of the stack, if required).

The following startup script creates a network-less environment for a user:

```
#!/bin/bash --norc
/usr/local/bin/um_add_service umnet,-ip
exec -l /bin/bash
```

It is also possible for a user to define an encrypted home directory, using encfs. The startup script follows:

```
#!/bin/bash --norc
/usr/local/bin/um_add_service umfuse
/usr/local/bin/um_add_service viewfs0
/bin/mount -t viewfs /home/testkm/crypt /tmp/testcrypt
/bin/mount -t umfuseencfs -o pre="" /tmp/testcrypt /home/testkm
exec -l /bin/bash
```

There are many other applications. All the virtualizations provided by umview/kmview can be defined and configured in the standard environment for a user.

The examples here above are just for user convenience, the security is not enforced. For example the network-less environment can be circunvented by removing the umnet module.

umview or kmview must start in *human mode* (see 9.3) to create sandboxes. For example the entries in `/etc/viewospasswd` should be changed as follows:

```
testkm:-s /bin/bash --norc --noprofile /home/testkm/.startviewos
testum:-s /bin/bash --norc --noprofile /home/testum/.startviewos
```

Remember that in *human mode* the command (bash in this case) starts as virtual uid 0. After the configuration the virtual uid must return to the uid of the user to enforce the protection. The network-less example script for the user testvm becomes:

```
#!/bin/bash --norc
/usr/local/bin/um_add_service umnet,-ip
exec viewsu testvm -l -s /bin/bash
```

In umview this protection denies the use of umview inside the environment, while kmview is able to provide nested virtualization. Note that the nested kmview sees the services provided by the outer kmview as its *real world*, thus it cannot circunvent the protections.

10.19 ▼*MVIEW modules in education

*MVIEW modules enables many kinds of exercises and projects.

- Design and implement an umdev device, for example RTC, disk, audio.
- Design and implement an umfuse file system implementation, for example Minix.
- Design and implement an umnet protocol stack, for example Appletalk or IRDA.
- Define a special user to test new software: after the login all the file system is writable (by viewfs) but all the modifications get lost at logout.
- Write a cache device driver. The driver has two threads, the first searches in the cache and enqueues a request in case of cache miss, the second refills the cache and applies a replacement algorithm to allocate space to load the new data.

Part IV

and they lived happily ever after...

CHAPTER 11

Conclusions and future developments

11.1 Conclusions

In the chapter introducing Virtual Square, the main goals of the project (*communication, unification and extension*) have been described, together with its design guidelines (*reuse of existing tools, modularity, no architectural constraints and openness*). In later chapters, several tools created by the Project Team have been detailed and studied in depth.

The tools achieved many of the project goals, and have been developeded strictly abiding the Virtual Square design guidelines. These tools are already available for use, and can be downloaded and tested, proving that the concepts behind the V^2 project are not only ambitious research ideas, but a concrete reality.

The Virtual Square research proved that it is possible to unify the communication interface accross a wide number of virtual machines (both System Virtual Machines and Process Virtual Machines), and it is possible to provide an interconnection mechanism between heterogeneous virtual machines.

The ViewOS concept has been proved feasible and useful, too: increasing the granularity of virtualization to processes via a modular implementation of a SCVM, can lead to specific virtualization for specific needs, thus avoiding waste of resources, and still mantaining a good isolation of potentially dangerous processes. Moreover, these possibilities are available entirely in user mode-user access.

In conclusion, Virtual Square can be considered as an innovative approach to virtuality, that breaks many of the barriers imposed by classic Operating Systems design, not solved by other Virtual Machines.

Future work

Although Virtual Square is already downloadable and in a good development stage, several improvements can be achieved, and new ideas can be exploited:

Millikernel. The research regarding KMView (the system call capture mode implemented at kernel level), together with a Linux kernel extensive support for partial virtualization, led to considerations about the kernel itself. Nowadays it is possible to imagine a minimal kernel providing basic functions, while all the other services can be provided by upper virtual layers. This vision could be a concrete attempt to mediate and overcome the old monolitic/microkernel debate. This is the reason behind the "millikernel" name: "milli" (10^{-3}) is the order of magnitude between 10^0 and "micro" (10^{-6}).

The use of partial virtual machines, instead of the servers of the classic microkernel, could be more flexible: a partial VM could be used either as a kernel module in the monolithic approach, and as a partial VM module for the microkernel approach. It could be up to the user to decide which module to use in which way.

Performance. Although every virtualization makes unavoidable some overhead, one of the aims of the V^2 project is to reach anyway the maximum level of optimization and performance possible. This step is very important in order to make the users able to effectively exploit the virtualized environment.

To achieve this, the project team is collaborating with the author of *utrace()* (a kernel patch used to capture system calls), in order to adopt it in the mainstream kernel: the use of *utrace()* is extimated to lead to a 20% loss of performance compared to a "real" environment.

VDE. Several improvements can be added to Virtual Distributed Ethernet:

- slirpvde currently supports only IPv4, so it may be interesting to develop a IPv6 support based on LWIPv6;
- support for GXemul and XEN;
- support for layer-3 switch for IPv6.

IPN. IPN is at a very early stage of development, thus a lot of effort on the progress of this tool is still required. Its implementation has to be completed, protocols can be added, other options for the network and the nodes need to be elaborated, and new applications exploiting IPN's potential must be created. These kind of applications include reimplementation of tuntap networking, VDE level 2 service, and generally every application that needs multicasting of data streams.

LWIPv6. This tool now supports can be used as a single stack, but it can be developed adding support for multi-stack use.

***MView.** Several improvements can be added to *MView: now processes within the virtual machine can only be started and closed, thus it may be interesting to ass support for freeze&restart of processes, so the user could interrupt a process, and restart it when needed; also a tool for the management of a mount table could be useful in order to control and use better the mounted files, partitions and services. Moreover, the ViewFS module needs to be improved, in order to achieve a complete and usable file system virtualization. Finally, new submodules for umfuse can be implemented to support other filesystem, like UDF.

FreeOSZoo. This project is developed by the V^2 team, and though it is not related to any of the previous tools, it offers an interesting virtualization service. FreeOSZoo provides ready-to-run images of QEMU virtual computers, pre-installed with a free operating system and a set of popular free software. The developers offers also a live version of FreeOSZoo in the official website: a place where anybody can test FreeOSZoo images without need to downloading them nor install QEMU. This project can be used in conjunction with VDE in order to provide virtual machines from the university to be used by users at home (with a vde_switch).

11.2 Acknowledges

The V^2 Team

Since this work is a review and recollection mainly of the Virtual Square Team's work, an acknowledgment is due to every member of the team. The list is quite long, but the project would not be the same without the large or small contribution of every single member:

Renzo Davoli (designer, main developer), associate professor of computer science, University of Bologna.

Michael Goldweber (virtual square in computer science education applications), full professor, Xavier University.

Ludovico Gardenghi (viewfs, optimization, logging interface), phd student, University of Bologna.

Daniele Lacamera (vde_cryptcab, debian packets) graduate student, University of Bologna.

Stefano Marinelli (OSZOO maintenance) former graduate student, University of Bologna.

Diego Billi (lwipv6 filtering, optimization and maintenance), former graduate student, University of Bologna.

Andrea Gasparini (nesting, kernel-patches) former graduate student in Computer Engineering, University of Bologna.

Mattia Gentilini (Live OSZOO) former graduate student, University of Bologna.

Guido Trotter (debian packet maintainer) former graduate student, University of Bologna, consultant.

Federica Cenacchi (documentation) research graduate student, University of Bologna.

and many others... Marco Dalla Via, Jacopo Mondi, Andrea Saraghiti, Marcello Stanisci, Andrea Forni, Mattia Belletti... people that contributed with minor but meaningful work.

Last but not least, very important contributions have come from some related projects, published under Free Software[1] Licences, that updated their code for Virtual Square compatibility: FUSE (Miklos Szeredi), QEMU (Fabrice Bellard), User-Mode Linux (Jeff Dike), utrace (Roland McGrath), Bochs.

[1] http://www.fsf.org

APPENDIX A

GNU Free Documentation License

Version 1.2, November 2002
Copyright © 2000,2001,2002 Free Software Foundation, Inc.

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Bibliography

[1] K. Andreev, B. M. Maggs, A. Meyerson, and R. Sitaraman. Designing overlay multicast networks for streaming. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 2003. San Diego, California.

[2] Y. Bard. An analytic model of cp-67 - vm/370. *In Proceedings of the workshop on virtual computer systems, pp. 170176*, 1973.

[3] F. Bellard. Qemu internals. 2005.

[4] J. Bowling. Zenoss and the art of enterprise monitoring. *Linux J.*, 2008(172):6, 2008.

[5] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security; Repelling the Wily Hacker.* Addison-Wesley, Reading, MA, second edition, 2003.

[6] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development, 25(5):483490*, 1981.

[7] R. Davoli. Virtual square project. http://www.virtualsquare.org, 2004.

[8] R. Davoli. Vde: virtual distributed ethernet. In *Proceedings of Tridentcom 2005*, Trento, 2005.

[9] R. Davoli. Virtual square. In *Proceedings of OSS2005. Open Source Software 2005*, Genova, 2005.

[10] R. Davoli, M. Goldweber, and L. Gardenghi. The virtual square framework. 2007.

[11] J. D. Dike. A user-mode port of the linux kernel. In *Proc. of 2000 Linux Showcase and Conference*, 2000.

[12] J. D. Dike. User-mode linux. In *Proc. of 2001 Ottawa Linux Symposium (OLS)*, Ottawa, 2001.

[13] A. Dunkels. Lwip web page. http://www.sics.se/ adam/lwip/.

[14] A. Dunkels. Minimal tcp/ip implementation with proxy support. Master's thesis, SICS - Swedish Institute of Computer Science, February 2001.

[15] A. Dunkels. Full tcp/ip for 8-bit architectures. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98, New York, NY, USA, 2003. ACM.

[16] A. Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proceedings of the First International Conference on Mobile Systems, Applications and Services (MobiSys)*, San Francisco, May 2003. USENIX.

[17] G. Galilei. *Il Saggiatore*. Accademia dei Lincei, 1623.

[18] R. Gunther. Kernel support for miscellaneous (your favourite) binary formats v1.1. Published inside the Linux source tree, Documentation/binfmt_misc.txt.

[19] V. Jacobson and R. T. Braden. Tcp extensions for long-delay paths, 1988.

[20] E. Kohlbrenner, D. Morris, and B. Morris. The history of virtual machines. 1999. http://www.cne.gmu.edu/modules/itcore/.

[21] J.-V. Loddo and L. Saiu. Status report: marionnet or "how to implement a virtual network laboratory in six months and be happy". In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 59–70, New York, NY, USA, 2007. ACM.

[22] G. F. Lyon. *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, USA, 2009.

[23] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. So-man, L. Youseff, and D. Zagorodnov. Eucalyptus open-source cloud-computing system. In *Proc. of In CCA08: Cloud Computing and Its Applications*, 2008.

[24] OpenVAS. Openvas home page. http://www.openvas.org.

[25] Quagga. Quagga routig suite home page. http://www.quagga.net.

[26] M. Rosenblum. The reincarnation of virtual machines. *ACM Queue, 2(5).*, 2004.

[27] J. E. Smith and R. Nair. The architecture of virtual machines. *IEEE COmputer*, 38(5):32–38, May 2005.

[28] M. Szeredi. Fuse: Filesystem in user space. http://fuse.sourceforge.net.