ICT-257422

CHANGE

CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions

Specific Targeted Research Project

FP7 ICT Objective 1.1 The Network of the Future

D3.3: Flow Processing Platform: Main Implementation

Due date of deliverable: 30 September 2012

Actual submission date: October 26th, 2012

Start date of project	1 October 2010
Duration	36 months
Lead contractor for this deliverable	NEC
Version	Final, December 8, 2012
Confidentiality status	Public

Abstract

This deliverable describes the main implementation of the CHANGE flow processing platform and the source code distribution that accompanies this document, which is restricted. We begin by covering the platform requirements and the resulting platform design. We continue by describing the platform's API, that is, the set of functions that are externally visible. This allows CHANGE platforms to vary significantly from each other in terms of hardware while still being able to inter-operate. In addition, we provide a detailed description of the internal platform controller architecture. Next, we give a detailed discussion of several processing module technologies developed in the project: netmap, VALE, Ministack, FlowOS and ClickOS. Finally, this document gives a brief description of how to install and run a CHANGE platform.

Target Audience

Network experts and researchers. This document is public, but the accompanying source code is restricted to the Commission Services and groups specified by the consortium.

Impressum

Full project title	CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions
Title of the workpackage	Work Package 3. Flow Processing Platform Design and Development
Editor	Felipe Huici, NEC
Project Co-ordinator	Adam Kapovits, Eurescom
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2012 Participants in project CHANGE

List of Authors

Authors	Felipe Huici (NEC), Costin Raiciu (PUB), Luigi Rizzo (UNIPI), Laurent Mathy (ULANC), Mehdi Bezahaf (ULANC), Michio Honda (NEC), Mohamed Ahmed(NEC)
Participants	NEC (Editor), PUB
Work-package	WP3: Flow Processing Platform Design and Development
Confidentiality	Public (PU)
Nature	Report (R)
Version	1.0
Total number of pages	66

Contents

List of Figures

List of Tables

1 Introduction

The Internet has grown over the last twenty years to the point where it plays a crucial role in todays society and business. By almost every measure, the Internet is a great success. It interconnects over a billion people, running a wide range of applications, with new ones appearing regularly that take the world by storm. Yet despite this success the Internet comes with important shortcomings. The limitations are well-known: the Internet does not provide predictable quality of service, and does not provide a sufficiently robust and secure infrastructure for critical applications. Worse, making changes to the basic Internet infrastructure is costly, time-consuming and often unfeasible: operators are paid to run stable, always available networks which is anathema to deploying new mechanisms.

To overcome these problems CHANGE introduces a new evolutionary (i.e., incrementally deployable) architecture based around the notion of *flow processing platforms*, or *Flowstream platforms* for short, located throughout the network. The idea is that for specific flows that need special processing, flow owners discover platforms along the path and request flow processing from them. What do we mean by “flow processing”? Flow processing can encompass a large number of actions, from filtering, NAT, DPI to packet scrubbing and monitoring, among others. In fact, as we envision at least certain versions of these platforms to be based on general-purpose hardware, flow processing will be largely user-defined, and we expect the more interesting uses of platforms will be for things that are currently not in existence.

In this deliverable we present the design and main implementation of the Flowstream platform. We begin by describing the requirements that led us to its design, giving an overview, and discussing how the platform relates to the work in other work packages of the CHANGE project.

We then continue by depicting, in detail, the platform’s external API. This API is used by any entities interested in inter-acting with a Flowstream platform (e.g., other platforms or CHANGE architecture components) in order to request flow processing on it. Next, we proceed to describe the design of the platform controller, the software that controls every function of a platform.

Beyond this control plane functionality, the main task of a platform is to process flows. This is the job of the processing modules, and in this deliverables we give details into a number of different systems developed in CHANGE that can perform this task: ClickOS, FlowOS, and Ministack; we further discuss netmap and VALE, other software developed in the project that accelerates packet I/O, improving the performance of the three processing module technologies previously mentioned.

The deliverable ends with a description of the current platform instantiation, using ClickOS as the processing module technology and deployed across four partner sites in three different countries.

1.1 Platform Requirements

The goals outlined above dictate a number of requirements for the Flowstream platforms:

- **Flexibility**: platforms should be able to perform a wide range of flow processing.

- **Dynamic Installation**: platforms should be able to install new types of processing on demand.
- **Dynamic Scalability**: platforms should be able to scale their capabilities to meet dynamic demand.
- **Isolation**: platforms should be able to concurrently host different kinds of processing for different users without one user's processing affecting another's. This includes taking measures so that untrusted code is not able to adversely affect other users or the basic functionality of the platform.
- **Flow Statistics**: platforms should be able to provide at least basic on-demand per-flow statistics to requesting users.
- **High Performance**: the platform should perform flow processing with good performance.

1.2 Flow Processing Platform Overview

As mentioned, Flowstream platforms have to have a fair amount of flexibility, both in terms of processing and in the ability to dynamically install processing. While the CHANGE architecture does not dictate any particular implementation of the platforms as long as they conform to a set of APIs, recent studies have shown that x86 servers not only have the processing flexibility needed but can also yield high performance [10][8][15]. These, coupled with programmable commodity hardware switches such as OpenFlow switches and virtualization technologies such as XEN [3], form the basis of one type of Flowstream platform (see figure 1.1).

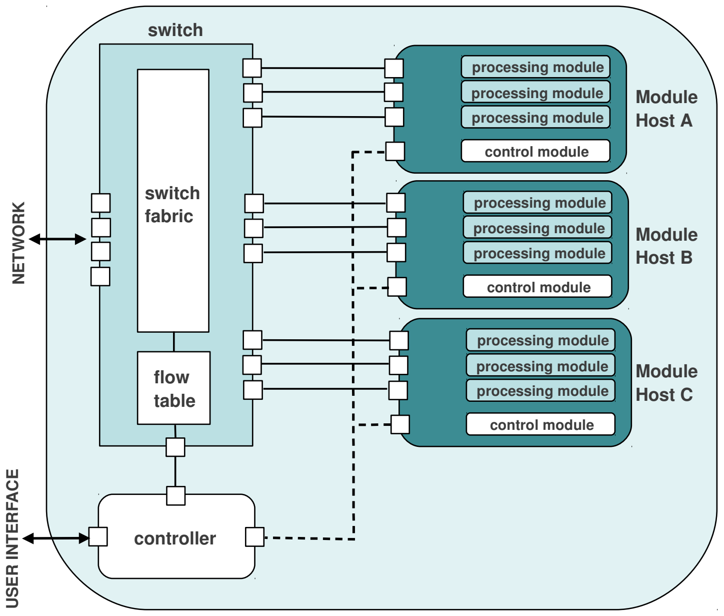

Figure 1.1: Flowstream Platform Overview.

In a Flowstream platform, a programmable switch such as an OpenFlow switch is used to distribute incoming flows to various *module hosts* (essentially x86 servers). The platform's capabilities can thus dynamically be scaled up by adding servers and down by shutting them down. Generally we assume module hosts to be computers containing general-purpose CPUs (e.g., x86 or x86_64 architectures). What system they actually run can vary. For instance, a module host could run XEN, another one Linux and yet another one FreeBSD.

In addition, the module hosts contain a number of entities called *processing modules* (PMs) where the actual network processing takes place. Here again, there are choices regarding a PM's implementation. These can range from a process in a Linux system (e.g., a process running Bro or Click [18]), a minimalistic OS or a full virtual machine running on XEN[1].

It is also worth pointing out that if needed, it is entirely possible to include a specialized hardware device (e.g., a DPI box) in a Flowstream platform. After the flows are processed by the processing modules, the switch is once again used to send them back out of the platform. Naturally, it might not always be necessary to send flows out; for instance, if Flowstream is being used to monitor mirrored flows the processing modules will act as sinks.

Flowstream also contains a controller which manages the entire platform and we will dedicate the rest of this document to its description.

1.3 Relation to Other Work Packages

This deliverable forms part of WP3, whose work it is to derive requirements for, design, and implement the CHANGE platforms and the flow processing modules that run in them; this work is described extensively in this deliverable.

The vision of the project is larger than a single platform however. How platforms can be coordinated to attain the ultimate goal of creating a more evolvable Internet is the task of WP2, concerned with the design of the CHANGE architecture. For instance, D2.4, the architecture's final specification, deals with issues such as flow naming, how to secure flow processing (e.g., who is allowed to instantiate which kind of processing on platforms), and how to reason about what happens to the network when different types of processing are instantiated (e.g., have I created networking loops?).

WP4 then is in charge of the practicalities of implementing the CHANGE architecture: for instance, how do potential clients discover platforms, how are flows attracted to Flowstream platforms, what sort of signalling should be used for platforms to communicate with each other and clients?

Finally, WP5 drives all this work by targeting and implementing specific applications to run on the CHANGE architecture, that is, a set of distributed Flowstream platforms. The deployment of a wide-area network CHANGE testbed, described at the end of this document, is a (large) step in this direction.

[1]The XEN case provides an easy way to provide isolation for untrusted code.

2 External Interface: Primitives

CHANGE platforms can vary widely in terms of the hardware they contain, from large deployments in data-centers consisting of commodity servers connected via switches (e.g., Openflow), to blade servers and even single-host platforms. In order for such vastly different platforms to be able to interact under a common CHANGE architecture we need to abstract a common API that summarizes the primitive functionality that is needed in order to instantiate flow processing.

We've split the API into two parts: a high-level API which should be the most commonly used one; and a low-level API for more advanced uses of CHANGE platforms. The software distribution that accompanies this document implements both, but currently only makes the high-level API externally visible.

2.0.1 High-Level API

Here's a list of the functions that a CHANGE platform must implement when conforming to its high-level API:

- `install_allocation`: Installs an allocation on a CHANGE platform. This includes instantiating all necessary processing modules and creating connections between them and traffic coming into and out of the platform. Returns an id for the allocation.

Parameters		
Name	**Type**	**Description**
`alloc`	`string`	The allocation request (see section below)

- `delete_allocation`: Deletes an allocation on a CHANGE platform. This includes removing processing modules and the connections between them. No return value.

Parameters		
Name	**Type**	**Description**
`alloc id`	`integer`	The id of the allocation to delete

- `deploy_pm`: Installs a new type of processing module on a CHANGE platform. This allows for remotely updating a platform. No return value.

Parameters		
Name	**Type**	**Description**
`pm_name`	`string`	The name of the new processing module
`pm_src`	`string`	The source code for the pm (see section 3).

- `get_pm_info`: Gets information about all processing modules currently supported by a platform. Returns a list of two-tuples. Each tuple has a string providing a human description of what the pm does and what parameter it takes, plus a dictionary describing the pm's primitives (e.g., "Read", see D2.4 section 4.4 for a full listing of these). Takes no parameters.

2.0.2 Low-Level API

Here's a list of the functions that a CHANGE platform must implement when conforming to its low-level API:

- `create_pm`: Instantiates a processing module on a CHANGE platform. No return value.

Parameters		
Name	**Type**	**Description**
`pm`	`PM`	The processing module to instantiate.

- `delete_pm`: Removes a processing module from a CHANGE platform. No return value.

Parameters		
Name	**Type**	**Description**
`pm`	`PM`	The processing module to remove.

- `create_connections`: Creates connections between processing modules on a platform. Currently this assumes the existence of an Openflow software switch. No return value.

Parameters		
Name	**Type**	**Description**
`switch_name`	`string`	The name of the Openflow switch.
`switch_table_id`	`integer`	The id of the table within the switch.
`alloc`	`string`	The allocation request describing the connections.

- `delete_connections`: Creates connections between processing modules on a platform. Currently this assumes the existence of an Openflow software switch. No return value.

Parameters		
Name	**Type**	**Description**
`switch_name`	`string`	The name of the Openflow switch.
`switch_table_id`	`integer`	The id of the table within the switch.
`alloc`	`string`	The allocation request describing the connections.

2.0.3 Return Values

All functions above return a common `ReturnValue` structure consisting of (1) a status code, (2) a message in the form of a string and (3) a function-specific return value. For the status code, the values supported are:

Code	Value	Explanation
CODE_NOT_READY	-1	The instruction is being carried out but is not finished
CODE_SUCCESS	0	The instruction was carried out successfully
CODE_FAILURE	1	The instruction encountered problems when executing

In the case of CODE_FAILURE, the message string that's part of `ReturnValue` should include an explanation of what the problem was. Note that functions in the API saying that they have no return value still return a `ReturnValue` object but with its third item set to void.

2.0.4 Allocation Request Format

At its most basic, a request to do network processing on a CHANGE platform (a *allocation request*) needs to specify three things: (1) a set of parametrized processing modules to process the actual packets; (2) how these processing modules should be inter-connected; and (3) where to receive packets from and where to send them to when finished.

To achieve this, CHANGE platforms use a very simple language to describe an allocation request[1]; here's an example:

```
BASESYS=xen.clickos


ft :: FromNet("tun0", "ip,nw_src=10.10.1.2,nw_dst=8.8.8.8")
tt :: ToNet("tun1")
mirror1 :: Mirror(")


ft -> [0]mirror1[0]
   -> tt
```

The first section of a config file defines key-value pairs. In the example, the configuration file that the processing module types should be instantiated as ClickOS virtual machines (a processing module such as a firewall may support different kinds of implementations, for instance as a ClickOS vm, a Linux vm, or perhaps an ipfw rule).

Next, the config defines the set of processing modules to be used, giving parameters for each of them where appropriate. Finally, the config specifies how the processing modules are connected. Each module can have a number of input and output ports, so users must explicitly specify which ports should be connected (the numbers in brackets in the example above).

Note that other related mechanisms such as authentication and access control are handled separately from the actual installation of an allocation request and are described in other deliverables.

[1]Note that in previous deliverables we were using an XML-based description language. In contrast, this new language is much simpler and accessible to users of the platform.

3 Main Implementation: the Controller

Having described what a CHANGE platform looks like to the outside world, we now move on to describing its internals, and in particular the architecture of the controller software that manages the platform.

3.1 Controller Components

While logically the controller is a single entity, in actuality it is implemented as a set of inter-communicating daemons (see figure 3.1; only one module host is shown for simplicity's sake). The separation of the controller into a set of daemons allows us to break down its functionality into more manageable pieces that use common daemon code. Further, this allows for different implementations of the daemons to exist as long as they comply with a common interface. For instance, the Openflow daemon could have two versions: one for a Linux-based Openflow switch talking to another daemon on the switch, and another one using NOX to program a hardware switch using the Openflow protocol.

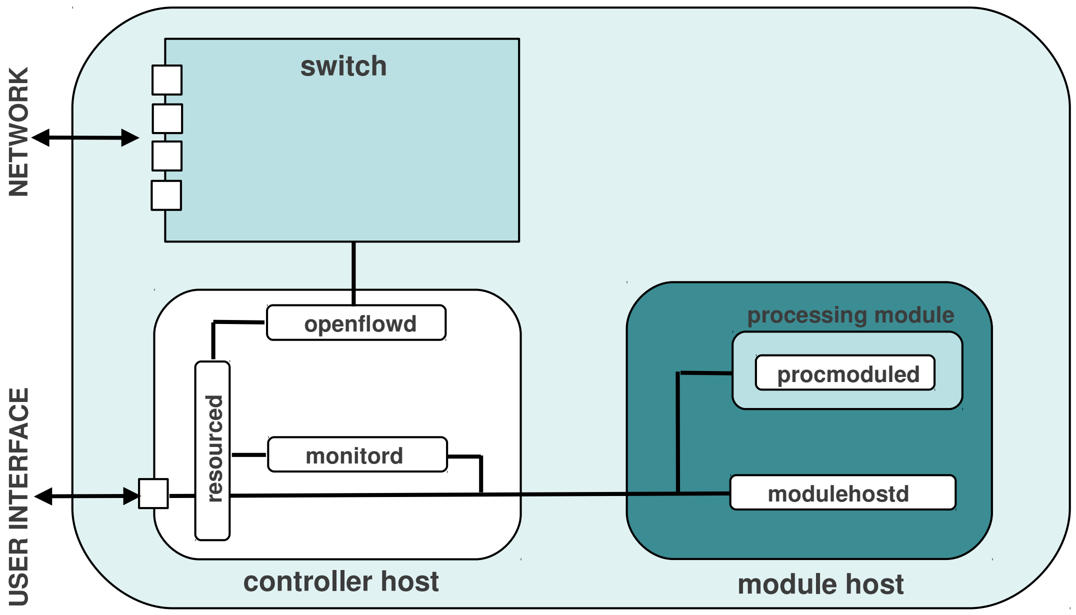

Figure 3.1: Platform controller architecture overview, only control-plane connections are shown.

The daemons comprising the Flowstream platform's controller are as follows:

- **Resource daemon**: This is the controller's main daemon, providing the interface to the outside world through which Flowstream users/clients submit *allocation requests* (as described in the previous chapter).

  This daemon takes care of receiving such requests and deciding how to allocate resources to them. It talks to the module host, processing module and Openflow daemons in order to install these allocations. In addition, it receives both the Openflow statistics from the Openflow daemon and performance statistics from the module hosts and processing modules via the monitoring daemon. Both of these are used as input to the resource allocation algorithms.

- **Openflow daemon**: This daemon is in charge of inserting and removing the Openflow entries that will direct traffic from the outward-facing ports (labeled "network" in the figure) to the necessary module hosts and processing modules and then back out. In addition, this daemon periodically retrieves Openflow statistics (e.g., per-flow byte and packet counts) and gives them to the resource daemon to use as input for its resource allocation algorithms. This daemon can make use of existing Openflow protocol implementations such as NOX [14] for communicating with the switch.

- **Module host daemon**: Runs on a module host and takes care, among other things, of instantiating and deleting processing modules, as well as setting up the necessary networking to get flows in and out of them. It also provides performance figures to the monitoring daemon about the current load of the module host.

- **Processing module daemon**: Runs on a processing module and handles modifications to the processing it is doing, as well as reporting performance statistics to the monitoring daemon. Note that depending on the implementation of the module host, it may or not be possible to run a daemon directly on the processing module. For instance, a processing module implemented as a minimalistic OS running on a Xen module host may not be able to run the processing daemon. This could be overcome by running the daemon on the module host (i.e., Xen's dom0) and using Xen's interface to modify the properties of the processing module.

- **Monitoring daemon**: This daemon periodically gathers performance statistics about the platform from the module host daemons, processing module daemons and the switch daemon, and gives these to the resource daemon, which uses the data as input to the resource allocation algorithms.

3.2 Daemon Architecture and Communications

At a high level, a daemon consists of a stand-alone process running an XML-RPC server. We chose XML-RPC since it is simple to use and widely available in different programming languages. A daemon takes care of executing two types of items: a *task* or a *command*. All daemons inherit from a common daemon class. The basic idea behind the architecture of this class is that most of a daemon's functionality resides in tasks and commands, while the daemon itself simply takes care of receiving requests and instantiating the right task or command for them. This simplifies the daemon itself, encourages code re-use (since commands or tasks can be shared by multiple daemons) and enables mechanisms such as hot-patching. The exception are functions that are common to many commands or tasks of the daemon; these functions typically reside in the class that implements the daemon, or alternatively could be put in a library file.

3.3 Tasks and Commands

The main difference between a task and a command is that a command will block while waiting for a result and a task will not. As a result, a command is generally meant to be used for carrying out a simple and quick operation, such as registering a module host with the resource daemon. A task, on the other hand, is asynchronous, and will immediately return after being called. Despite this, tasks, if so desired, can still return a value.

The final difference between tasks and commands is that tasks can be assigned normal or high priority, and also allow for compound tasks. A compound task is a task containing multiple sequential tasks. Tasks within a compound task are executed by the daemon in the same order as they were added to the compound tasks (FIFO). As a result, a compound task provides a convenient way of doing sequential execution of multiple tasks. Of course, a developer could instead choose to mimic the mechanism using multiple regular tasks, but the compound task interface is cleaner.

3.3.1 Structure of Tasks and Commands

The Flowstream controller's tasks and commands will follow a two-tier structure. Tasks and commands common to all daemons (for instance, a task to upgrade a daemon's set of commands and tasks) will reside in a common directory, while additional per-daemon directories will contain daemon-specific functionality.

In order to implement a new task (and similarly for a command), developers place a file in the relevant daemon's sub-directory under tasks, and in that file place a class that inherits from a task super-class. The subclass will have to implement a `run_task` method that receives a single parameter containing all of the method's parameters. Which data structure this single parameter is depends entirely on the command's implementer. If the method requires more than one parameter, these need to be packed into the single parameter, for instance by using a list. After the method finishes performing its function, it returns a common return type to signal the result of the operation back to the caller.

3.3.2 Task Priorities

Tasks will have at least two priority types, high and normal. As their names indicate, these are used to give priority to some tasks over others, with normal tasks only running when no high priority tasks exist at the daemon. For example, tasks that carry out background monitoring (for example, checking CPU loads on module hosts) would be invoked with normal priority, while installing a user configuration would use high priority.

3.3.3 Hot Patching

The controller's daemons include a hot patching mechanism in order to provide the ability to upgrade a daemon's functionality without having to restart it. The simplest way of achieving this is to have all tasks and commands dynamically loaded. In other words, when a request to execute a task arrives at a daemon, the daemon looks in the relevant directory to see if a task with the given name exists, and if it does, loads the task's code and runs it. With this in place, introducing new tasks or commands to a daemon simplifies to putting the relevant files in the daemon's directory. For tasks or commands that are already loaded, their respective modules are reloaded.

4 Main Implementation: Processing Module Technologies

4.1 Netmap and VALE

In a previous deliverable (D3.2) we reported the development and evaluation of Netmap, a novel framework for fast packet I/O. In this section we describe the continuation of this work called VALE. VALE takes the work further by creating a very fast software bridge that leverages the fast I/O provided by netmap. We dedicate this section to a detailed discussion of VALE.

4.1.1 Introduction

A large amount of computing services nowadays are migrating to virtualized environments, which offer significant advantages in terms of resource sharing and cost reduction. Virtual machines need to communicate and access peripherals, which for systems used as servers mostly means disks and network interfaces. The latter are extremely challenging to deal with even in non-virtualized environments, due to the high data and packet rates involved, and the fact that, unlike disks, traffic generation is initiated by external entities on which the receiver has no control. It is then not a surprise that virtual machines may have a tough time in operating network interfaces at wire speed in all possible conditions.

As it is often the case, hardware assistance comes handy to improve performance. As shown in section 4.1.2.2, some proposals rely on multiqueue network cards exporting resources to virtual machines through PCI passthrough, and/or on external switches to copy data between interfaces. However this solution is expensive and not necessarily scalable. On the other hand, software-only solutions proposed to date tend to have relatively low performance, especially for small packet sizes.

We then wondered if there was an inherent performance problem in doing network switching in software. The result we found is that high speed forwarding between virtual machines is achievable even without hardware support, and at a rate that exceeds that of physical 10 Gbit/s interfaces, even with minimum-size packets.

Our contribution: The main result we present in this paper is a system called VALE, which implements a Virtual Local Ethernet that can be used to interconnect virtual machines, or as a generic high speed bus for communicating processes. VALE is accessed through the netmap API, an extremely efficient communication mechanism that we recently introduced [28]. The same API can be trivially used to connect VALE to hardware devices, thus also providing communications with external systems at line rate [27].

VALE is 10..20 times faster than other software solutions based on general purpose OSes (such as in-kernel bridging using TAP devices or various kinds of sockets). It also outperforms NIC-assisted bridging, being capable to deliver well over 17 Mpps with short frames, and over 6 Mpps with 1514-byte frames (corresponding to more than 70 Gbit/s).

The use of the netmap API is an enabling factor for such performance, but the packet rates reported in this

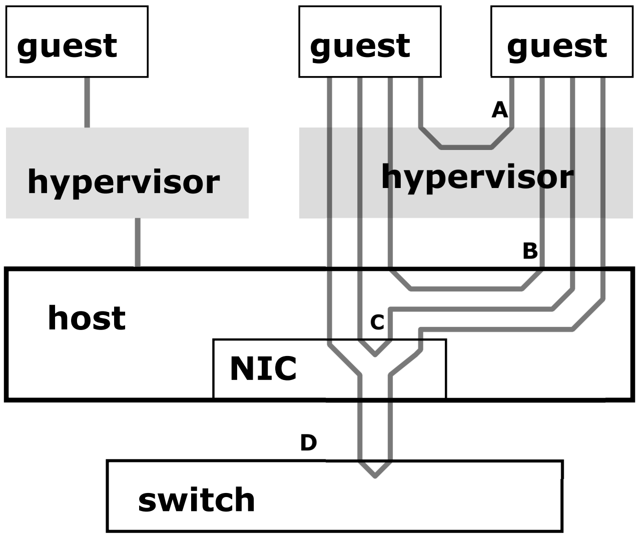

Figure 4.1: Possible paths in the communication between guests: A) through the hypervisor; B) through the host, e.g. via native bridging; C) through virtual queues the NIC; D) through an external network switch.

paper could have not been achieved without engineering the forwarding code in a way that exploits batched processing. section 4.1.3.3 discussed the solutions we use.

To prove that VALE's performance can be exploited by virtual machines, we then added VALE support to QEMU [5] and KVM [16], and measured speedups between 2 and 6 times for applications running on the guest OS[1], reaching over 2.5 Mpps with short frames, and about 2 Mpps with 1514-byte frames, corresponding to 25 Gbit/s. We are confident that we will be able to reach this level of performance also with other hypervisors and guest NIC drivers.

The paper is structured as follows. In Section 4.1.2 we define the problem we are addressing in this paper, and present some related work that is also relevant to describe the solutions we adopted. Section 4.1.3 details the architecture of our Virtual Local Ethernet, discussing and motivating our design choices. Section 4.1.4 comments on some implementation details, including the hypervisor modifications needed to make use of the VALE infrastructure, and device emulation issues. We then move to a detailed performance measurement of our system, comparing it with alternatives proposed in the literature or implemented in other products. We first look at the raw switch performance in Section 4.1.5, and then study the interaction with hypervisors and guest in Section 4.1.6. Finally, Section 4.1.7 discusses how our work can be used by virtualization solutions to achieve large speedups in the communication between virtual machines, and indicates some directions for future work.

4.1.2 Problem definition

The problem we address in this paper is how to implement a high speed Virtual Local Ethernet that can be used as a generic communication mechanism between virtual machines, OS processes, and physical network interfaces.

Solutions to this problem may involve several components. Virtual machine instances in fact use the services

[1]especially when we can overcome the limitations of the emulated device driver, see Section 4.1.6

supplied by a *hypervisor* (also called Virtual Machine Monitor, VMM) to access any system resource (from CPU to memory to communication devices), and consequently to communicate with each other. Depending on the architecture of the system, resource access from the guest can be mediated by the hypervisor, or physical resources may be partially or completely allocated to the guest, which then uses them with no interference from the hypervisor (except when triggering protection mechanisms).

Figure 4.1 illustrates implementations of the above concepts in the case of network communication between virtual machines. In case A, the hypervisor does a full emulation of the network interfaces (NICs), and intercepts outgoing traffic so that any communication between the virtual machine instances goes through it; in QEMU, this is implemented with the `-net user` option. In case B, the hypervisor still does NIC emulation, but traffic forwarding is implemented by the host, e.g. through an in-kernel bridge (`-net-tap`) or a module such as the one we present in this paper.

Other solutions give the virtual machine direct access to the NIC (or some of its queues). In this case, the NIC itself can implement packet forwarding between different guests (see the path labeled C), or traffic is forwarded to an external switch which in turn can bounce it back to the appropriate destination (path D).

NIC emulation, as used in A and B, gives the hypervisor a lot of control on the operations done by the guest, and works as a nice adaptation layer to run the guest on top of hardware that the guest OS would not know how to control.

Common wisdom suggests that NIC emulation is slow, and direct NIC access (as in C and D) is generally necessary for good virtual network performance, even though it requires some form of hardware support to make sure that the guest does not interfere with other critical system resources.

However, the belief that NIC emulation is inherently slow is wrong, and mostly the result of errors in the emulation code [32] or missing emulation of features (such as interrupt moderation, see [31]) that are fundamental for high rate packet processing. While direct hardware access may help in communication between the guest and external nodes, virtio [34], proprietary virtual device emulators [37, 36], and as we show in this paper, even e1000 emulation, done properly, provide excellent performance, comparable or even exceeding that of real hardware in VM-to-VM communication.

4.1.2.1 Organization of the work

In summary, the overall network performance in a virtual machine depends on three components:

- the guest/hypervisor communication mechanism;
- the hypervisor/host communication mechanism;
- the host infrastructure that exposes multiple physical or virtual NIC ports to its clients.

In this paper we first present an efficient architecture for the last component, showing how to design and implement an extremely fast Virtual Local Ethernet (VALE) that can be used by the hypervisors (and possibly exported to the guest OS), or even used directly by regular host processes. We then extend some popular hypervisors so that their communication with the host can be made efficient and exploit the speed of VALE. Finally, we discuss mechanisms that we implemented in previous work and that can be extremely effective to improve performance in a virtualized OS.

4.1.2.2 Related work

There are three main approaches to virtualized I/O, which rely on different choices in the communication between guest and hypervisor, and between hypervisor and the host/bridging infrastructure. We examine them in turn.

4.1.2.2.1 Full virtualization The simplest approach (in terms of requirements for the guest) is to expose a virtual interface of the type known to the guest operating system. This typically involves intercepting all accesses to critical resources (NIC registers, sometimes memory regions) and use them to update a state machine in the hypervisor that replicates the behaviour of the hardware. Historically, this is the first solution used by most emulators, starting from VMware to QEMU [5] and other recent systems.

With this solution the hypervisor can be a simple process that accesses the network using standard facilities offered by the host: TCP/UDP sockets (usually to build tunnels or emulate ethernet segments), or BPF/libpcap [19] or virtual interfaces (TAP) to inject raw packets into the network.

TAP is a software device with two endpoints: one is managed as a network interface (NIC) by the operating system, the other one is a file descriptor (FD) driven by a user process. Data blocks sent to the FD endpoint appear as ethernet packets on the NIC endpoint. Conversely, ethernet packets that the operating system sends to the NIC endpoint can be read by the user process from the FD endpoint. A hypervisor can then pass the guest's traffic to the FD endpoint of a TAP device, whose NIC endpoint can be connected to other NICs (TAPs or physical devices) using the software bridges available in the Linux and FreeBSD kernels, or other software bridges such as Open vSwitch [24].

There also exist solutions that run entirely in user space, such as VDE [12], providing configurable tunnels between virtual machines. In general, this kind of solution offers the greatest ease of use, at the expense of performance. Another possibility is offered by MACVTAPs [1], which are a special kind of TAP devices that can be put in a "bridge" mode, so that they can send packets to each other. Their main purpose is to simplify networking setup, by removing the need to configure a separate bridge.

4.1.2.2.2 Paravirtualization The second approach goes under the name of paravirtualization [4] and requires modifications in the guest. The guest becomes aware of the presence of the hypervisor and cooperates with it, instead of being intercepted by it. As far as I/O is concerned, the modifications in the guest generally come in the form of new drivers for special, "paravirtual" devices. VMware has always offered the possibility to install the VMware Tools in the guest to improve interoperability with the host and boost I/O performance. Their vSphere virtualization infrastructure also offers high performance vSwitches to interconnect virtual machines [36].

Xen offers paravirtualized I/O in the form a special *driver domain* and pairs of backend-frontend drivers.

Frontend drivers run in the guests and exchange data with the backend drivers running in the driver domain, where a standard Linux kernel finally completes the I/O operations. This architecture achieves fault isolation and driver reuse, but performance suffers [22]. Bridging among the guests is performed by a software bridge that connects the driver backends in the guest domain.

XenLoop [39] is a solution that improves throughput and latency among Xen guests running on the same host. It uses FIFO message queues in shared memory to bypass the driver domain. XenLoop is tightly integrated with the hypervisor, and in fact, its exclusive focus seems to provide a fast network channels to hypervisors. For this reason is not completely comparable with the VALE switch presented in this work, which also aims to implement be a generic communication mechanism useful also outside the virtualization world. In terms of performance, XenLoop seems to peak at around 200 Kpps (on 2008 hardware), which is significantly below the performance of VALE.

The KVM [42] hypervisor and the Linux kernel (both as a guest and a host) offer support for virtio [34] paravirtualization. Virtio is an I/O mechanism (including virtio-net for network and virtio-disk for disk support) based on queues of scatter-gather buffers. The guest and the hypervisor export shared buffers to the queues and notify each other when batches of buffers are consumed. Since notifications are expensive, each endpoint can disable them when they are not needed. The main idea is to reduce the number of context switches between the guest and the hypervisor.

Vhost-net [2] is an in-kernel data-path for virtio-net. Vhost-net is used by KVM, but it is not specifically tied to it. In KVM, virtio-net *notify* operations cause a hardware-assisted VM-exit to the KVM kernel module. Without vhost-net, the KVM kernel module then yields control to the KVM process in user space, which then accesses the virtio buffers of the guest, and writes packets to a TAP device using normal system calls. A similar, reversed path is followed for receive operations.

With vhost-net enabled, instead, the KVM kernel module completely bypasses the KVM process and triggers a kernel thread which directly writes the virtio buffers to the TAP device. The bridging solutions for this technique are the same as those for full virtualization using TAP devices, and the latter typically become the performance bottleneck.

4.1.2.2.3 Direct I/O access The third approach is to avoid guest/hypervisor communication altogether and allow the guest to directly access the hardware [41]. In the simplest scenario a NIC is dedicated to a guest which gains exclusive access to it, e.g., by PCI passthrough. This generally requires hardware support to be implemented safely. Moreover, DMA transfers between the guest physical memory and the peripheral benefit from the presence of an IOMMU [9]. More complex scenarios make use of multi-queue NICs to assign a separate queue to each guest [35], or of programmable network devices to implement device virtualization in the device itself [25]. These solutions are generally able to achieve near native performance in the guest, but at the cost of requiring specialized hardware. Bridging can be performed either in the NIC itself, as in[35], or by connecting real external switches.

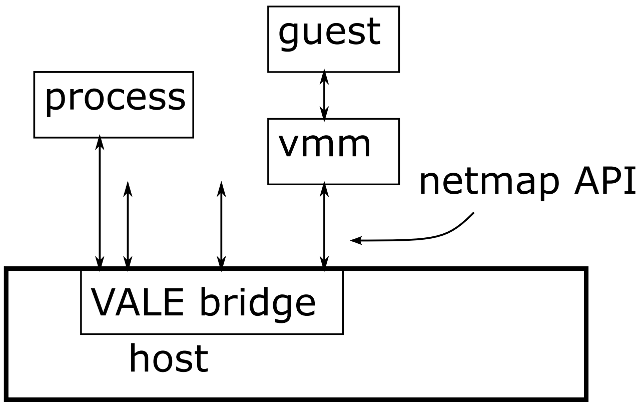

Figure 4.2: A VALE local ethernet exposes multiple independent ports to hypervisors and processes, using the netmap API as a communication mechanism.

4.1.3 VALE, a virtual local ethernet

The first goal of this work is to show how we built a software, high performance Virtual Local Ethernet (which we call VALE), that can provide access ports to multiple clients, be them hypervisors or generic host processes, as shown in Figure 4.2. The target throughput we are looking for is in the millions of packets per second (Mpps) range, comparable or exceeding that of 10 Gbit/s interfaces. Given that the hypervisor might be running the guest as a userspace process, it is fundamental that the virtual ethernet is accessible from user space with low overhead.

As mentioned, network I/O is challenging even for systems running on real hardware, for the reasons described in [28]: expensive system calls and memory allocations are incurred on each packet, while packet rates of millions of packets per second exceed the speed at which system calls can be issued.

In netmap [27] we solved these challenges through a series of simple but very effective design choices, aimed at amortizing or removing certain costly operations from the critical execution paths. Given the similarity to the problem we are addressing in VALE, we use the netmap API as the communication mechanism between the host and the hypervisor. A brief description of the netmap API follows.

4.1.3.1 The netmap API

The netmap framework was designed to implement a high performance communication channel between network hardware and applications in need of performing raw packet I/O. The core of the framework is based on a shared memory region (Figure 4.3), which hosts packet buffers and their descriptors, and is accessible by the kernel and by userspace processes. A process can gain access to a NIC, and tell the OS to operate it in netmap mode, by opening the special file `/dev/netmap`. An `ioctl()` is then used to select a specific device, followed by an `mmap()` to make the netmap data structures accessible.

The content of the memory mapped region is shown in Figure 4.3. For each NIC, it contains preallocated buffers for transmit and receive packets, and one or more[2] pairs of circular arrays (*netmap rings*) that store metadata for the transmit and receive buffers. Besides the OS, buffers are also accessible to the NIC through

[2]for NICs with multiple transmit and receive queues

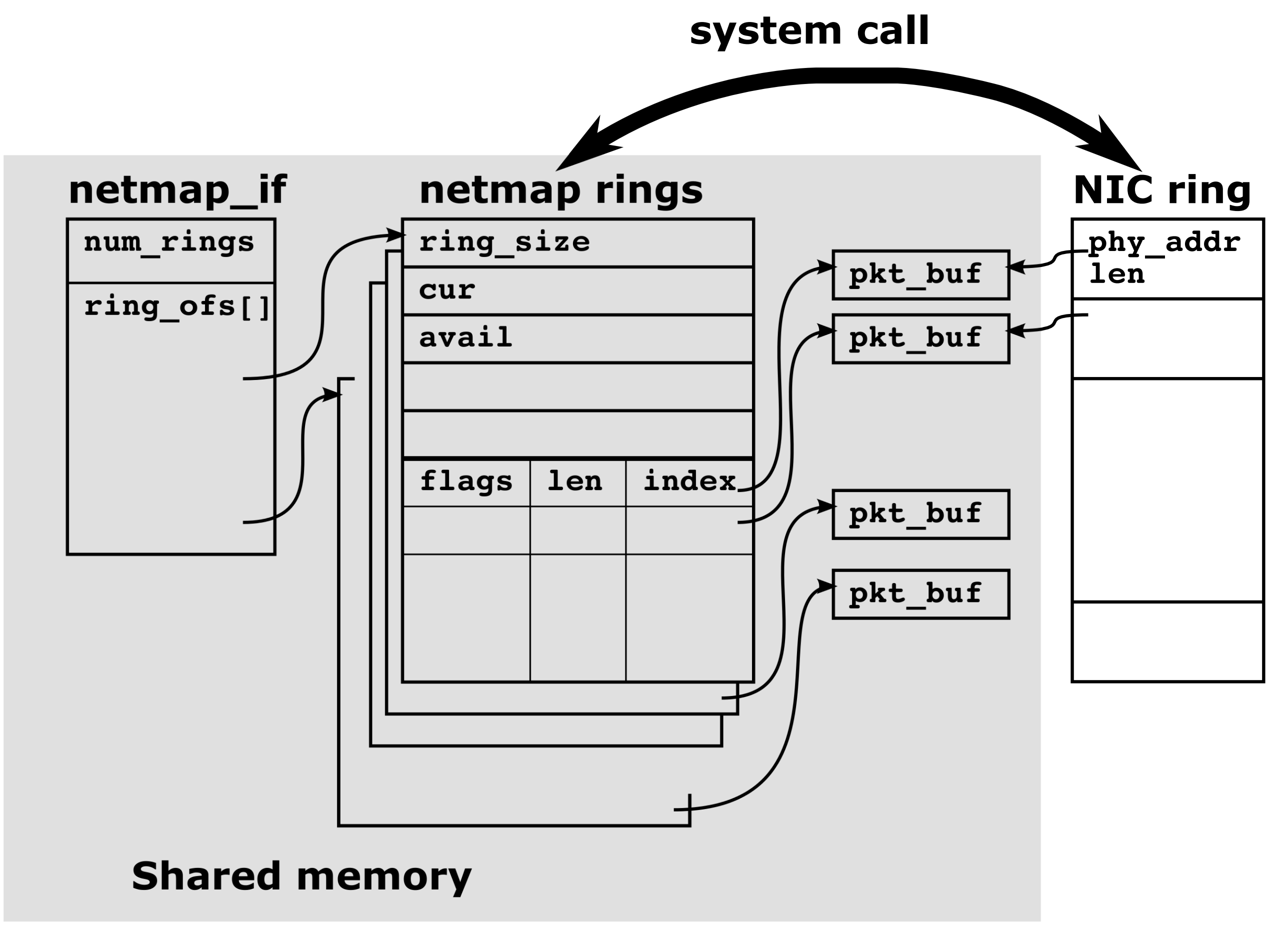

Figure 4.3: The memory areas shared between the operating system and processes, and manipulated using the netmap API.

its own *NIC rings* – circular arrays of buffer descriptors used by the NIC's hardware to store incoming packets or read outgoing ones.

Using a single `ioctl()` or `poll()` system call, a process can notify the kernel to send multiple packets at once (as many as they fit in the ring). Similarly, receive notifications for an entire batch of packets are reported with a single system call. This way, the cost of system calls (up to 1 μs or more even on modern hardware) is amortized and their impact on individual packets can become negligible.

The other expensive operations – buffer allocations and data copying – are removed because buffers are preallocated and shared between the user process and (ultimately) the NIC itself. The role of the system calls, besides notifications, is to validate and convert metadata between the netmap and the NIC ring, and to perform safety-critical operations such as writing to the NIC's register. The implicit synchronization provided by the system call makes access to the netmap ring safe without the need of additional locking between the kernel and the user process.

Netmap is a kernel module made of two parts. Device-independent code implements the basic functions (`open(), close(), ioctl(), poll()/select()`), while device-specific *netmap-backends* extend device drivers and are in charge of transferring metadata between the netmap ring and the NIC ring (see

Figure 4.3). The backends are very compact and fast, allowing netmap to send or receive packets at line rate even with the smallest packets (14.88 Mpps on a 10 Gbit/s interface) and with a single core running at less than 900 MHz. True zero-copy between interfaces is also supported, achieving line-rate switching with minimum-sized packets at a fraction of the maximum CPU speed.

4.1.3.2 A netmap-based Virtual Local Ethernet

From a user's perspective, our virtual switch shown in Figure 4.2 offers each user an independent, virtual NICs connected to a switch and accessible with the netmap API. NIC names start with the prefix `vale` (to differentiate them from physical NICs); both virtual NICs and switches are created dynamically as users access them.

When the netmap API requests to access a NIC named `valeX:Y` (where X and Y are arbitrary strings), the system creates a new VALE switch instance called X (if not existing), and attaches to it a port named Y.

Within each switch instance, packets are transferred between ports as in a learning ethernet bridge: the source MAC address of each incoming packet is used to learn on which port the source is located, then the packet is forwarded to zero or more outputs depending on the type of destination address (unicast, multicast, broadcast) and whether the destination is known or not. The following pseudocode describes the forwarding algorithm. The set of destinations is represented by a bitmap, so up to 64 output ports can be easily supported. A packet is forwarded to port `j` if `dst & (1<<j)` is non zero.

```
void tx_handler(ring, src_if) {
    cur = ring->cur; avail = ring->avail;
    while (avail-- > 0) {
        pkt = ring->slot[cur].ptr;
        // learn and store the source MAC
        s = mac_hash(pkt->src_mac);
        table[s] = {pkt->src_mac, src_if};
        // locate the destination port(s)
        d = mac_hash(pkt->dst_mac);
        if (table[d].mac == pkt->dst_mac)
             dst = table[d].src;
        else
             dst = ALL_PORTS;
        dst &= ˜(1<<src_if); // avoid src_if
        // forward as needed
        for (j = 0; j < max_ports; j++) {
            if (dst & (1<<j)) {
                lock_queue(j);
                pkt_forward(pkt, ring->slot[cur].len, j);
                unlock_queue(j);
            }
        }
        cur = NEXT(cur);
    }
    ring->cur = cur; ring->avail = avail;
}
```

This code, operating on one packet at a time, is very similar to the implementation of most software switches found in common OSes. However its performance is poor (relatively speaking; we measured almost 5 Mpps as shown in Figure 4.8, which is still 5 times faster than existing in-kernel bridging code), for two main reasons: locking and data access latency.

The incoming queue on the destination port, in fact, must be protected against concurrent accesses, and in the above code the cost of locking (or equivalent mechanism) is paid on each packet, possibly exceeding the packet processing costs related to bridging. Secondly, the memory accesses to read the metadata and headers of each packet may have a significant latency if these data are not in cache.

4.1.3.3 Performance enhancements

To address these performance issues, we use a different sequence of operations, which permits processing packets in batches and supports data prefetching.

Batching is a well know technique to amortize the cost of some expensive operations over a large set of objects. The downside of batching is that, depending on how it is implemented, it can increase the latency of a system. Given that the netmap API used by VALE supports the transmission of potentially large sets of packets on each system call, we want to provide the system administrator with mechanisms to enforce the desired tradeoffs between performance and latency.

The key idea is to implement forwarding in multiple stages. In a first stage we collect a batch of packets of *bounded maximum batch size* from the input set of packets; a short batch is created if there are fewer packets available than the batch size. In this stage we copy metadata (packet sizes and indexes/buffer pointers), and also issue prefetch instructions for the payload of the packets in the batch, so that the CPU can start moving data towards caches before using them. Figure 4.4 shows how packets from the netmap ring are copied into a working array (`pool`) which also has room to store the set of destinations for each packet (these fields will be filled in the next stage). The pseudocode for this stage is the following:

```
void tx_handler(ring, src_if) {
    // stage 1, build batches and prefetch
    i = 0; cur = ring->cur; avail = ring->avail;
    for (; avail-- > 0; cur = NEXT(cur)) {
        slot = &ring->slot[cur];
        prefetch(slot->ptr);
        pool[i++] = {slot->ptr, slot->len, 0};
        if (i == netmap_batch_size) {
```

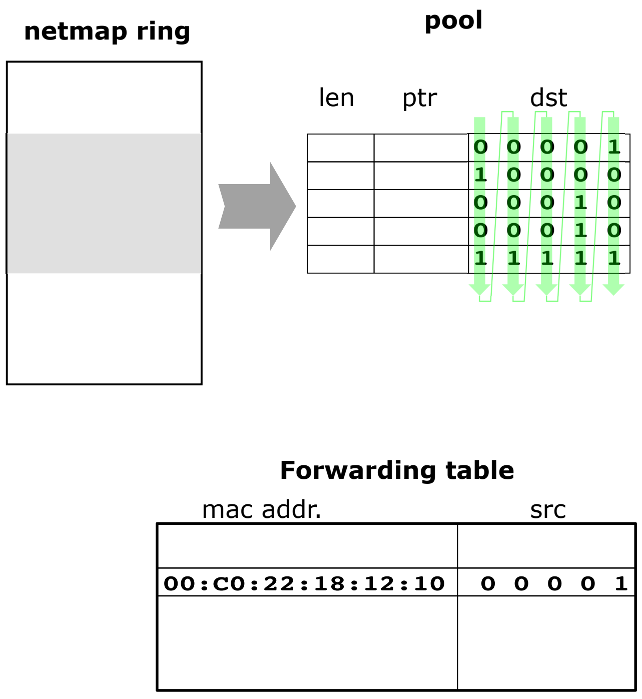

Figure 4.4: The data structures used in VALE to support prefetching and reduce the locking overhead. Chunks of metadata from the netmap ring are copied to a temporary array, while at the same time prefetching packets' payloads. A second pass then runs a destination lookup in the forwarding table, and finally the temporary array is scanned in column order to serve each interface in a single critical section.

```
            process_batch(pool, i, src_if);
            i = 0;
        }
    }
    if (i > 0)
        process_batch(pool, i, src_if);
}
```

4.1.3.3.1 Processing a batch The next processing stage involves updating the forwarding table, and computing the destination port(s) for each packet. The fact that the packet's payload has been brought into caches by the previous prefetch instructions should reduce the latency in accessing data. Furthermore, repeated source/destination addresses within a batch should improve locality in accessing the forwarding table. Once destinations have been computed, the final stage can forward traffic iterating on output ports, which satisfies our requirement of paying the cost of locking/unlocking a port only once per batch. The following pseudocode shows a simplified version of the forwarding logic.

```
void process_batch(pool, n, src_if) {
    // stage 2, compute destinations
    for (i = 0; i < n; i++) {
        pkt = pool[i].ptr;
        s = mac_hash(pkt->src_mac);
        table[s] = {pkt->src_mac, src_if};
        d = mac_hash(pkt->dst_mac);
        if (table[d].mac == pkt->dst_mac)
            pool[i].dst = table[d].src;
        else
            pool[i].dst = ALL_PORTS;
    }
    // stage 3, forward, looping on ports
    for (j = 0; j < max_ports; j++) {
        if (j == src_if)
            continue; // skip src_if
        lock_queue(j);
        for (i = 0; i < n; i++) {
            if (pool[i].dst & (1<<j))
                pkt_forward(pool[i].pkt, pool[i].len, j);
        }
        unlock_queue(j);
    }
}
```

In the actual implementation, the final stage is further optimized to reduce the cost of the inner loop. As an example, the code skips ports for which there is no traffic, and delays the acquisition of the lock until the first packet for a destination is found, to shorten of critical sections[3].

4.1.3.3.2 Avoiding multicast Forwarding to multiple destinations is a significant complication in the system, both in terms of space and runtime. The worst case complexity of stage 3, even for unicast traffic, is still $O(N)$ per packet, where N is the number of ports in the bridge. If a bridge is expected to have a very large number of ports (e.g. connected to virtual machines), it may make sense to remove support for multicast/broadcast forwarding[4].

To achieve this, the field used to store bitmap addresses can be recycled to build lists of packets for a given destination, thus avoiding an expensive worst case behaviour. Traffic originally directed to multiple ports (unknown destinations, or ARP requests/advertisements, BOOTP/DHCP requests, etc.) can be directed to a default port where a user-level process will respond appropriately, e.g. serving ARP and DHCP requests (a similar strategy is normally used in access nodes – DSLAM/BRAS – that terminate DSL lines).

4.1.3.3.3 Alternative forwarding functions We should note that stage 2 is the only place where the forwarding function is invoked. It is then relatively simple, and it will be the subject of future work, to replace it with alternative algorithms such as an implementation of a OpenFlow dataplane.

[3]Computing the first and last packet for each destination is a lot more expensive in the generic case, as it requires iterating on the bitmap containing destinations. Even more expensive, from a storage point of view, is tracking the exact set of packets for a given destination.

[4]This feature will be implemented in future releases.

4.1.3.4 Copying data

The final processing stage calls function `pkt_forward()`, which is in charge of queueing the packet on one of the destination ports. The simplest way to do this, and the one we use in VALE, is to copy the payload from the source to the destination buffer. Copying is especially convenient because it supports the case where the source and destination ports have buffers mapped in mutually-inaccessible address spaces. This is exactly one of the key requirements in VALE, where clients attached to the ports are typically virtual machines, or other untrusted entities, that should not interact in other ways than through the packets sent to them.

Ideally, if a packet has only a single destination, we could simply swap buffers between the transmit and the receive queue. However this method requires that buffers be accessible in both the sender and the receiver's address spaces and is not compatible with the need to isolate virtual machines.

While data copies may seem expensive, on modern systems the memory bandwidth is extremely high so we can achieve good performance as long as we make sure to avoid cache misses. We use an optimized copy function which is extremely fast: if the data is already in cache (which is likely, due to the `prefetch()` and the previous access to the MAC header), it takes less than 15 ns to copy a 60-bytes packet, and 150 ns for a 1514-bytes packet.

4.1.4 Implementation details

VALE is implemented as a kernel module, and is available from [33] for both FreeBSD and Linux as an extension of the netmap module. Thanks to the modular architecture of the netmap software, the additional features to implement VALE required less than 1000 additional lines of code, and the system was running on both FreeBSD and Linux from the very first version. The current prototype supports up to 8 switches per host (the number is configurable at compile time), with up to 64 ports each.

4.1.4.1 External communication

As presented, VALE implements only a local ethernet switch, whose use is restricted to processes and hypervisors running on the same host. The connection with external networks is however trivially achieved with one of the tools that are part of our netmap framework, which can bridge two arbitrary netmap interfaces at line rate, using an arrangement similar to that in Figure 4.5. We can use one netmap-bridge to connect to the

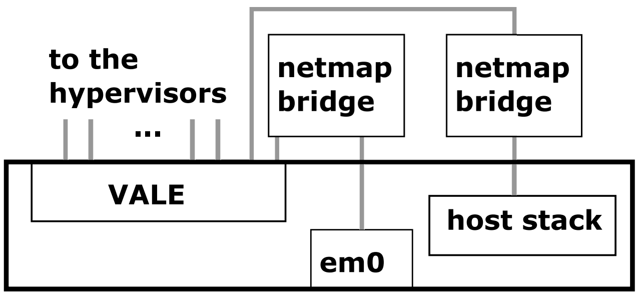

Figure 4.5: The connection between VALE, physical network interfaces and the host stack can be built with existing components (netmap bridge) which implement zero-copy high speed transfer between netmap-compatible ports.

host stack, and one or more to connect to physical interfaces.

Because the relevant code is already existing and operational with the desired performance, we will not discuss it in the experimental section. As part of future work, we plan to implement a (trivial) extension of VALE to directly access the host stack and network interfaces without the help of external processes.

4.1.4.2 Hypervisor changes

In order to make use of the VALE network, hypervisors must be extended to access the new network backend. For simplicity, we have made modifications only to two popular hypervisors, QEMU [5] and KVM [16], but our changes apply in a similar way to VirtualBox and other systems using host-based access to network interfaces. We do not foresee much more complexity in making VALE ports accessible to Xen DomU domains.

The QEMU/KVM backend is about 400 lines of code and it implements the open and close routines, and the read and write handlers called on the VALE file descriptor when there is traffic to transfer.

Both QEMU and KVM access the network backend by `poll()`'ing on a file descriptor, and invoking read-/write handlers when ready. For VALE, the handlers do not need to issue system calls to read or write the packets: the payload and metadata are already available in shared memory, and information on new packets to send or receive are passed to the kernel the next time the hypervisor calls `poll()`. This enables the hypervisor to exploit the batching, in turn contributing to reduce the I/O overhead.

As we will see in the performance evaluation, VALE is much faster than other software solutions used to implement the network backend. This extra speed may stress the hypervisor in unusual ways, possibly emphasizing some pre-existing performance issue or bugs. We experienced similar problems when modifying Open vSwitch to use netmap [30], and we found similar issues in this work.

Specifically, we encountered two performance-related bugs in the NIC emulation code in QEMU.

First of all, after some asymmetric tx versus rx performance results [32], we discovered that the code involved in the guest-side emulation of most network cards was missing a notification to the backend when the receive queue changed status from full to not-full. This made the input processing timeout-based rather than traffic based, effectively limiting the maximum receive packet rate to 100-200 Kpps. The fix, which we pushed to the QEMU developers, was literally one line of code and gave us a speedup of almost 10 times, letting us reach 1-2 Mpps range depending on the hypervisor.

The second problem involves the emulation of interrupt moderation, a feature that is fundamental to achieve high packet rates even on real hardware. Moderation is even more important in virtual machines where the handling of interrupts (which involves numerous accesses to I/O ports) is exceptionally expensive. We found that the e1000 emulation code implements the registers related to moderation, but then makes no use of them, causing interrupts to be generated immediately on every packet transmission or reception. We developed a partial fix [31] which improved the transmit performance by 7-8 times with this change alone.

We note that problems of this kind are extremely hard to identify (it takes a very fast backend to generate this much traffic, and a fast guest to consume it) and easy to misattribute. As an example, the complexity of device emulation is usually indicated as the main reason for poor I/O performance, calling for alternative solutions such as `virtio` [34] or other proprietary APIs [37].

4.1.4.3 Guest issues

A fast network backend and a fast hypervisor do not imply that the guest machines can communicate at high speed. Several papers in the literature show that even on real hardware, packet I/O rates on commodity operating systems are limited to approximately 1 Mpps per core. High speed communication (1..10 Gbit/s) is normally achieved thanks to a number of performance-enhancing techniques such as the use of jumbo buffers and hardware offloading of certain functions (checksum, segmentation, reassembly).

As we recently demonstrated [27], this low speed is not an inherent limitation in the hardware, but rather the result of exceeding complexity in the operating system, and we have shown how to achieve much higher packet rates using netmap as the mechanism to access the network card.

As a consequence, for some of our high speed tests we will use the network device in netmap mode *also within the guest*. The same reasons that make netmap very fast on real hardware, also help when running on emulated hardware: on both the transmit and the receive side, operations that need to be run in interpreted mode, or to trap outside the emulator, are executed once per each large batch of packets, thus contributing to improving performance.

4.1.5 Performance evaluation

We have measured the performance of VALE on a few different multicore systems, running both FreeBSD and Linux as the host operating systems, and QEMU and KVM as hypervisors. In general, these experiments are extremely sensitive to CPU and memory speeds, as well as to data layout and compiler optimizations that may affect the timing of critical inner loops of the code. As a consequence, for some of the (many) tests we have run there are large variations (10-20%) of the experimental results, also due to slightly different versions of the code or to the synchronization of the processes involved. We also noted a steady and measurable decline in QEMU/KVM network throughput (about 15% on the same hardware) between versions 0.9, 1.0 and 1.1.

This said, the difference in performance between VALE and competing solutions is much larger (4..10 times) than the variance on the experimental data, so we can draw correct conclusions even in presence of noisy data.

For the various tests, we have used a combination of the following components:

- **Hardware and host operating systems:**
  i7-2600K (4 core, 3.2 GHz) + FreeBSD-9;
  i7-870 (4 core, 2.93 GHz) + FreeBSD-10;
  i5-750 (4 core, 2.66 GHz) + Linux 3.2.12.
  i7-3930K (6 core/12 threads, 3.2 GHz) + Linux 3.2.0.
  In all cases RAM is DDR3-1.33 GHz and the OS is running in 64-bit mode.

- **Hypervisors:** QEMU 1.0.1 (both FreeBSD and Linux); KVM 1.0.1/1.1.0 (Linux).
- **Network backends:** TAP with/without vhost-net (Linux); VALE (Linux and FreeBSD).
- **Guest network interface/OS:** plain e1000, e1000-netmap (Linux and FreeBSD); virtio (only Linux).

Not all combinations have been tested due to lack of significance, unavailability, or bugs which prevented certain configurations from working. We should also mention that, especially for the data reporting peak performance, the numbers we report here are conservative. During the development of this work we have implemented some optimizations that shave a few nanoseconds from each packet's processing time, resulting in data rates in excess of 20 Mpps at minimum packet sizes.

Following the same approach as in the description of the system, we first benchmark the performance of the virtual local ethernet, be it our VALE system or equivalent ones. This is important because in many cases the clients are much slower, and their presence would lead to an underestimate of the performance of the virtual switch.

4.1.5.1 Bridging performance

The first set of tests analyzes the performance of various software bridging solutions. The main performance metric for packet forwarding is the throughput, measured in *packets per second* (pps) and *bits per second* (bps).

pps is normally the most important metric for routers, where the largest cost factors (source and destination lookup, queueing) are incurred on each packet and are relatively independent of packet size. Bridges and switches, however, need *also* (but not *only*) a characterization in bps, because they operate at very high rates and often hit other bottlenecks such as memory or bus bandwidth. We will then run our experiments with both minimum and maximum ethernet-size packets (60 and 1514 bytes).

Note that many *bps* figures reported in the literature are actually measured using jumbograms (8-9 Kbytes). The corresponding *pps* rates are between 1/6 and 1/150 of those shown here.

The traffic received by a bridge should normally go to a single destination, but there are cases (multicast or unknown destinations) where the bridge needs to replicate packets to multiple ports. Hence the number of active ports impacts the throughput of the system.

In Figure 4.6 we compare the forwarding throughput when using 60 and 1514 byte packets, for three technologies: TAP plus native Linux bridging[5], our VALE bridge, and NIC/switch-supported bridging (in this case we report the theoretical maximum throughput on a 10 Gbit/s link).

Since the traffic directed to a bridge can be forwarded[6] to a single output port (when the destination is known), or to multiple ports (for multicast/broadcast traffic, or for unknown destinations), we expect a decrease in the forwarding rate as the number of active ports grows.

[5] we also tested the in-kernel Open vSwitch module, but it was always slightly slower than native bridging.

[6] In principle we should also consider the case of traffic being dropped by the bridge, but this is always much faster than the other cases (as an example, VALE can drop packets at almost 100 Mpps), so we do not need to worry about it.

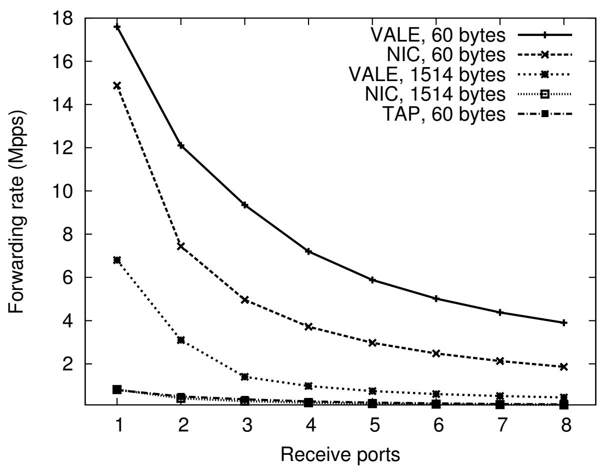

Figure 4.6: Forwarding rate versus number of destinations. VALE beats even NIC-based bridging, with over 17 Mpps (vs. 14.88) for 60-byte packets, and over 6 Mpps (vs. 0.82) for 1514-byte packets. TAP is at the bottom, peaking at about 0.8 Mpps in the best case.

Indeed, all the three solutions (VALE, NIC, TAP) expose a $1/N$ behaviour. For VALE and TAP this is because the forwarding is done by a single core while the work is proportional to the number of ports. For switch-based bridging, the bottleneck is instead determined by the bandwidth available on the link from the switch, which has to carry all the replicas of the packet for the various destinations. A similar phenomenon occurs for NIC-based bridging, this time the bottleneck being the PCI-e bus.

In absolute terms, for traffic delivered to a single destination and 60-byte packets, the best options available for Linux bridging achieve a peak rate of about 0.80 Mpps. Next comes NIC-based forwarding, which is limited by the bandwidth on the PCI-e interconnection between the NIC and the system. Most 10 Gbit/cards on the market use 4-lane PCI-e slots per port, featuring a raw speed of 16 Gbit/s per port per direction. Considering the overhead for the transfer of descriptors, each port has barely enough bandwidth to sustain line rate. In fact, as we measured in [27], packet sizes that are not multiple of a cache line size cannot even achieve line rate due to extra traffic generated to read and write entire cache lines.

The curves for VALE are still above, peaking at 17.6 Mpps for a single destination and 60-byte packets, again decreasing as the number of receivers grows. Here the bottleneck is given by the combination of CPU cycles (needed to do the packet copies) and memory bandwidth.

The numbers for 1514-byte packets are even more impressive. Linux bridging is relatively stable at a low value (packet size is not a major cost item). NIC based forwarding is still limited to line rate, approximately 820 Kpps, whereas VALE reaches over 6 Mpps, or 72 Gbit/s.

4.1.5.1.1 Per packet time Using the same data as in Figure 4.6, Figure 4.7 shows the per-packet processing time used by VALE depending on the number of destinations. This representation gives a better idea of the time budget involved with the various operations (hashing, address lookups, data copies, loop and locking overheads). Among other things, these figures can be used to determine a suitable batch size so that the total processing time matches the constraint of latency-sensitive applications.

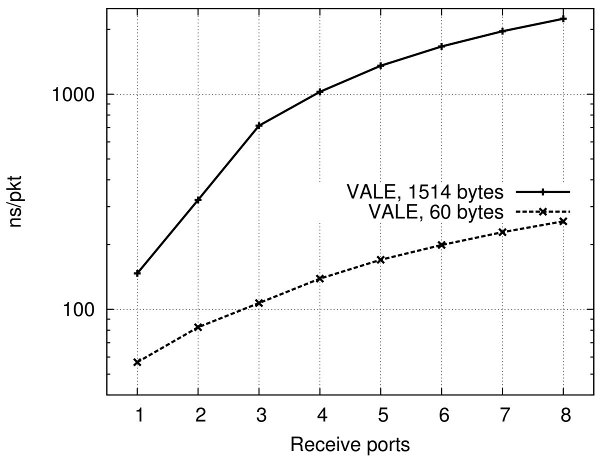

Figure 4.7: Per-packet time versus number of destinations for VALE. See text in Section 4.1.5.1.1 for details.

The figure shows that for one port and small packets the processing time is 50-60 ns per packet when using large batches (128 and above). With separate measurements we estimated that about 15 ns are spent in computing the Jenkins hash function, taken from the FreeBSD bridging code, and approximately 15 ns are also necessary to perform a copy of a 60-byte packet (with data in cache). The cost of copying a 1514 byte packet is estimated at about 150 ns.

As mentioned in Section 4.1.3.4, VALE uses data copies in all cases. In the case of multicast/broadcast traffic, the only alternative to copying data would be to implement shared readonly buffers and a reference-counted mechanism to track when the buffer can be freed. Apart from the complications of the mechanism, the cost in accessing the reference count from different CPU cores is likely comparable or greater than the data copy costs, even for 1514-byte packets.

4.1.5.2 Effectiveness of batching

The huge difference in throughput between VALE and other solutions depends on both the use of the netmap API, which amortizes the system call costs, and also on the processing of packets in batches in the forwarding loop. Figure 4.6 has been computed for a batch size of 1024 packets, but as we will show, even smaller batch sizes are very effective.

There are in fact three places where batching takes place: on the TX and RX sides, where its main goal is to amortize the cost of the system call to exchange packets with the kernel; and within the kernel, as discussed in Section 4.1.3.3, where the goal is to reduce locking contention.

The following experiments show the effect of various combinations of these three batching parameters. Experiments have been run on a fast i7-3930K using Linux 3.2, and unicast traffic between one sender and one receiver; the CPU has multiple cores so we expect sender and receiver to run in parallel on different cores.

Figure 4.8 shows the throughput versus kernel batch size, when both sender and receiver use a large value (1024) to amortize the system call cost as much as possible. Starting at about 5 Mpps with a batch of 1, we achieve dramatic increase even with relatively small kernel batches (e.g. 8 packets), and there is a diminishing return as we move above 128 packets. Here we have the following dynamics: the kernel, with its small batch size, acts as the bottleneck in the chain, waking up the receiver frequently with small amounts of packets. Because it uses a large batch size, the receiver cannot become the bottleneck: if during one iteration it is too slow in processing packets, in the next round it will be able to catch up draining a larger set of packets.

Figure 4.9 shows the behaviour of the system with different receiver batch sizes, and kernel and sender using a batch of 1024. In this experiment the receiver is the bottleneck, and the main cost component here is the system call, which however always finds data available so it never needs to sleep (a particularly costly operation). As a consequence, a small batching factor suffices to reach the peak performance.

Finally, Figure 4.10 shows the effect of different transmit batch sizes. The kernel is always using a batch of 1024. The top curve, with a receive batch of 1024, resembles the behaviour of Figure 4.8. The kernel (and the receiver) indeed behave in a similar way in the two cases, because the sender in the first place is feeding the bridge with only a few packets at a time. The initial region, however, shows lower absolute values because the interval between subsequent invocations of `process_batch()` now includes a complete system call, as opposed to a simple iteration in `tx_handler()`.

The bottom curve, computed with a receive batch of 1 packet, gives a better idea of where the bottleneck lies depending on the tx and rx batch sizes. A small batching factor on the receiver will definitely limit throughput no matter how the other components work, but the really poor operating point – about 2 Mpps in these experiments – is *when the entire chain processes one packet at a time.* It is unfortunate that this is exactly what happens in practice with most network APIs.

4.1.5.3 Processing time versus packet size

Coming back to the comparative analysis of different bridging solutions, we now move to the evaluation of the effect of packet size on throughput. Same as in Section 4.1.5.2, we run the test between one sender and one receiver connected through a bridge (VALE or native linux bridging), and for variable packet sizes. The measurement is done in the best possible conditions (which, for VALE, means large batches). Also, it is useful to study the effect of packet sizes by looking at the time per packet, rather than absolute throughput.

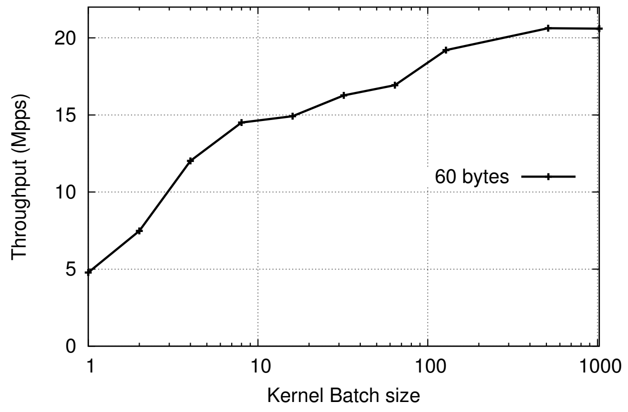

Figure 4.8: Throughput versus kernel batch size. Sender and receiver use batch=1024.

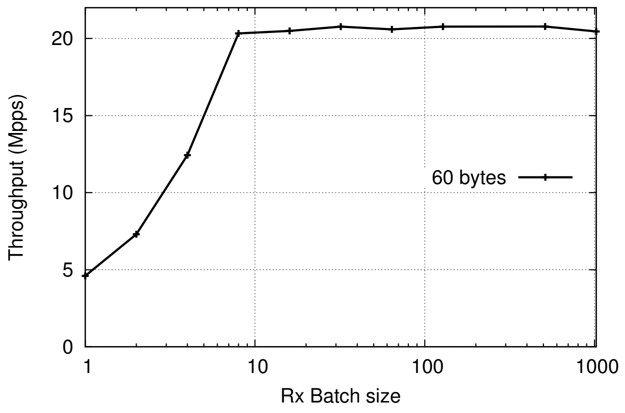

Figure 4.9: Throughput versus receive batch size. Kernel and sender use batch=1024.

Figure 4.11, presents the packet processing time versus the packet size. At these timescales the most evident phenomenon is the cost of data copies. VALE and TAP operate at the speed of the memory bus. This is confirmed by the experimental data, as the two curves have a similar slope. TAP of course has a much

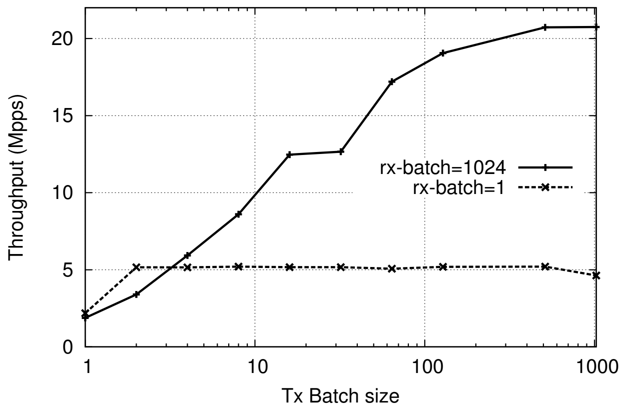

Figure 4.10: Throughput versus transmit batch size. Kernel uses batch=1024.

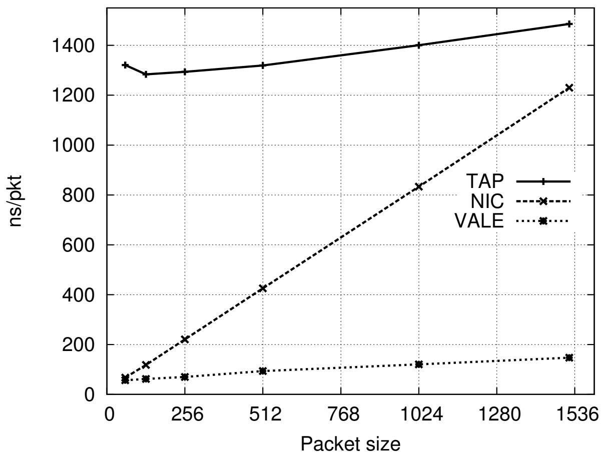

Figure 4.11: Per-packet time versus packet size. This experiments shows the impact of data copies. VALE and TAP have a similar slope, as they operate at memory bus speed. NIC-based bridging operates at a much lower (PCI-e or link) speed, hence producing a steeper curve.

higher base value, which is the root cause of its overall poor performance. The curve for NIC-based bridging, instead, is much steeper. This is also expected, because the bottleneck bandwidth (PCI-e or link speed) is several times smaller than that of the memory bus.

The curve for TAP presents a small dip between 60 and 128 bytes, which has been confirmed by a large number of tests. While we have not investigated the phenomenon, it is not unlikely that the code tries (and fails) to optimize the processing of small packets.

4.1.5.4 Latency

We conclude our performance analysis with an investigation on the communication latency. This test is mostly pointing out the poor performance of the operating system's primitives involved, rather than the qualities of the bridging code. Nevertheless it is important to know what kind of latency we can expect at best between communicating processes on the same system.

Figure 4.12 shows the round trip time between two processes talking through VALE (bottom curve) or a TAP bridge (top curve). In this experiment a "client" transmits a single packet and then blocks until a reply is received. The "server" instead issues a first system call to receive the request, and a second one to send the response. The actual amount of processing involved in the process (in the sender, receiver and forwarding code) is negligible, well below 1 μs, which means that the cycle is dominated by the system calls and the interactions with the scheduler (both client and server block waiting for the incoming packet).

As expected, VALE is almost unaffected by the message size (in the range of interest), as there is only a single, and very fast, copy in each direction. TAP instead uses at least two (and possibly three) copies on each direction, hence the different slope.

Care should be taken in comparing these numbers with other RPC mechanisms (e.g. InfiniBand, RDMA, etc.) designed to achieve extremely low latency. These system in fact exploit a number of latency-removal techniques that often require direct hardware access (such as exposing device registers to userspace to save the

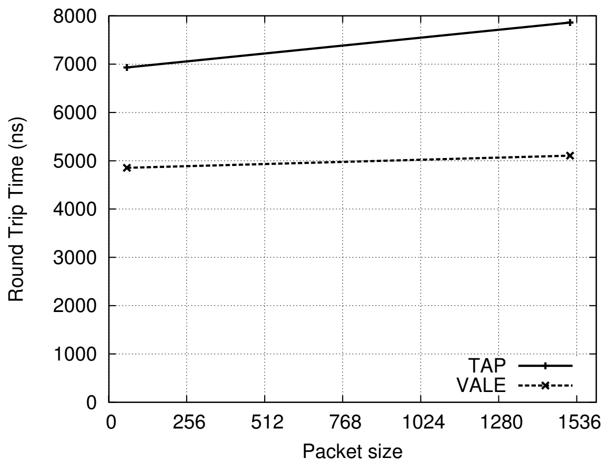

Figure 4.12: Round trip time in the communication between two processes connected through a linux native bridge (top) or VALE (bottom). In both cases, the times are dominated by the cost of the `poll()` system call on the sender and the receiver.

Configuration	Speed, Mpps				Notes
	TAP		VALE		
	tx	rx	tx	rx	
Raw bridge speed	.90	.90	19.7	19.7	
e1000 QEMU	.018	.014	.023	.023	
e1000 KVM	.020	.020	.024	.024	
netperf virtio QEMU	.012	.010	.023	.012	
virtio KVM	.400	.300	.480	.470	kvm 1.0.1
virtio KVM	.370	.300	1.200	–	kvm 1.1.1
virtio KVM vhost	.600	.580			
pkt-gen e1000 QEMU	.490	.490	**2.400**	**1.850**	
pkt-gen e1000 KVM	.550	.550	**3.470**	**2.550**	
pkt-gen e1000 KVM	.500	.490	**2.300**	**2.000**	1514 b
vmxnet3 ESX	.800	.800	-	-	see [36]
vmxnet3 vSphere	.800	.800	-	-	see [37]

Table 4.1: Communication speed between virtual machine instances for different combinations of source/sink (netperf if not specified), emulated device, and hypervisor. Some combinations were not tested due to software incompatibilities. The numbers for VMWare are extracted from [37] and [36] and refer to their own software switch.

cost of system calls), are not efficient (such as running a busy-wait loop on the client to remove the scheduler cost), and rely on hardware support (e.g. in RDMA the the server side is completely done in hardware, thus cutting almost half of the processing cost).

Some improvements could be achieved also in our case, e.g. spinning on a shared memory location to wait for data, but our goal is mostly to show what level of performance can be achieved with safe and energy-efficient techniques without hardware assist.

4.1.6 Running a hypervisor on a fast bridge

The final part of this paper measures how the hypervisor and the guest OS can make use of the fast interconnection provided by VALE. All tests involve:

- a traffic source and sink, running in the guest. We use `netperf`, a popular test tool which can source or sink TCP or UDP traffic, and `pkt-gen`, a netmap-based traffic source/sink which generates UDP traffic;
- an emulated network device. We use `e1000` (emulating a 1 Gbit/s device) and `virtio`, which provides a fast I/O interface that can talk efficiently to the hypervisor;
- a hypervisor. We run our tests with QEMU and KVM;
- a virtual bridge. We use native linux bridging accessed with TAP+vhost, and VALE.

The performance of some of the most significant combinations is shown in Table 4.1. In some cases we were unable to complete the tests due to incompatibilities between the various options.

All tests were run on an i7-3930K CPU running Linux 3.2 in the host, and with Linux guests.

In the table, the top row reports the raw speed of the bridge for both TAP and VALE. This serves as a baseline to evaluate the virtualized versus native throughput.

The next two rows report standard configurations with netperf and e1000 driver, running on QEMU or KVM. In both cases the packet rate is extremely low, between 14 and 24 Kpps. The interrupt moderation patches [31] bring the transmit rate to 56 and 140 Kpps, respectively, though we are still far from the the throughput that can be achieved on real hardware (about 1.3 Mpps in this configuration). Measurements on QEMU show that, on each packet transmission, the bottom part of the device driver emulation consumes almost 50 μs, a big part of which is spent for handling interrupts.

The virtio driver improves the situation at least when running on top of KVM. Here the use of VALE as a backend gives some improvements[7], although limited by the fact that both source and sink process only one packet at a time.

With such a high per-packet overhead, replacing the TAP bridge with the (much faster) VALE switch can only have a very limited effect on performance. The recipe for improving performance is thus to make better use of the (emulated) network device. The netmap API comes to our help in this case, and the numbers using `pkt-gen` prove that.

When running `pkt-gen` on top of e1000+QEMU (or KVM), the throughput increases by a large factor, even with the TAP bridge (from 20 to ≈500 Kpps). The main reasons are that `pkt-gen` accesses the NIC's registers very sparingly, typically once or twice per group of packets, thus making the emulation a lot less expensive. The improvement is even more visible when running on top of the VALE, which can make use of the aggregation in the guest, and issue a reduced number of system calls to transfer packets from/to the host.

This experiment shows that even without virtio is in principle possible to transfer minimum-size packets at rates that exceed the speed of a 1 Gbit/s interface, and for 1514-byte packets we reach 20 Gbit/s even without virtio.

A lot of these performance improvements come from the use of larger batch sizes when talking to the backend. We are investigating solutions to exploit this operating regime, both with the help of modified device drivers in the host, and implementing mechanisms similar to interrupt moderation within the VALE switch.

4.1.6.1 Comparison with other solutions

QEMU and KVM are neither the only nor the fastest hypervisors on the market. Commercial solutions go to great lengths to increase the speed of virtualized I/O devices. To the best of our knowledge, solutions such as vSphere [37] and ESX [36] are among the best performer on the market[8], claiming a speed of about 800 Kpps between two virtual machines (which translates to slightly less than 10 Gbit/s with 1514-byte packets). The vendor's documentation [37] reports up to 27 Gbit/s TCP throughput with jumbo frames (9000 bytes) which should correspond to packet rates in the 4-500 Kpps range.

These numbers cannot be directly compared with the ones we achieved on top of VALE. Even though we get higher packet rates, we are not running through a full TCP stack on the guest; on the other hand we have a much worse virtualization engine and device driver to deal with. What we can still claim, however, is that we are able to achieve the same level of performance of high-end commercial solutions.

[7]the low number for KVM+VALE on the receive side seems due to a livelock in KVM 1.1.1 which we are investigating
[8]possibly not the only ones to provide such speeds.

4.1.7 Concluding remarks

We have presented the architecture of VALE, a high speed Virtual Local Ethernet freely available for FreeBSD and Linux, and given a detailed performance evaluation comparing VALE with NIC-based bridging and with various existing options based on linux bridging. Additionally, we have developed QEMU and KVM modifications that show how these hypervisors, with proper traffic sources and sinks, can make use of the fast interconnect and achieve very significant speedups.

We should note that VALE is neither limited to use with virtual machines, nor to pure ethernet bridging. Indeed, the fact that VALE ports use the netmap API, and the availability of `libpcap` emulation library for netmap, means that a VALE switch can be used to interconnect various packet processing tools (traffic generators, monitors, firewalls etc.), and this permits performance testing at high data rates without the need of expensive hardware.

As an example, we recently used VALE switches to test a high speed version of the ipfw and dummynet traffic shaper [6], including the QFQ packet scheduler [7]. In this environment we were able to validate operation at rates exceeding 6 Mpps, well beyond the ability of regular OSes to source or sink traffic, let alone pass it to applications.

Also, the code that implements the forwarding decision (which in VALE is simply a lookup of the destination MAC address) can be trivially replaced with more complex actions, such as a software implementation of an OpenFlow switch. The main contribution of VALE, in fact, is in the framework to move traffic efficiently between ports and the module implementing forwarding decisions.

We are confident that, as part of future work, we will be able to make VALE compatible with better hypervisors and emulated device drivers (virtio and similar ones), and make its speed exploitable also by the network stack in the guest. At the high pps rates supported by VALE, certain operating systems functions (schedulers, synchronizing system calls) and emulator-friendliness need to be studied in some detail to identify possible performance bottlenecks.

The simplicity of our system and its availability should help the identification and removal of performance problems related to virtualization in hypervisors, device drivers and operating systems.

4.2 Ministack

Netmap provides fast packet I/O into user space memory, and VALE extends this to provide a multi-port fast software bridge. Ministack takes this further, building a fast framework that allows concurrent processing modules to run different, isolated, and potentially customized, network stacks. The rest of this section describes its design, implementation and early evaluation results.

4.2.1 Motivation

Networking stacks have been implemented in the kernel since the inception of Unix to ensure good performance, security and isolation between user processes. As networks have gotten faster and faster, there has been a trend towards offloading certain tasks to hardware (e.g., checksumming and segmentation) in order to accelerate packet processing. However, this speed-up results in reduced flexibility: we are essentially embedding the functionality of today's protocols in hardware at the expense of future ones or even to the detriment of extensions to current ones [17]; this has the end result of making the network harder to evolve.

In this abstract we argue that instead of moving network functionality down to the kernel and hardware, it should be shifted *up*, into user-space. Such a change would make development and deployment of new protocols easier, and would allow optimizations of the network protocol suited to the specific needs of an application [11]. Indeed, we could envision several variations of a commonly-used protocol such as TCP to be running concurrently.

The question now is: is it possible to move the network stack into user-space while still achieving high throughput and retaining the necessary security and isolation guarantees provided by the kernel? Recent developments show that using the *netmap* API it is possible to achieve high performance when moving packets to and from a NIC and user-space, while removing a lot of common in-kernel overheads such as `skbuf/mbuf` management and system calls [29].

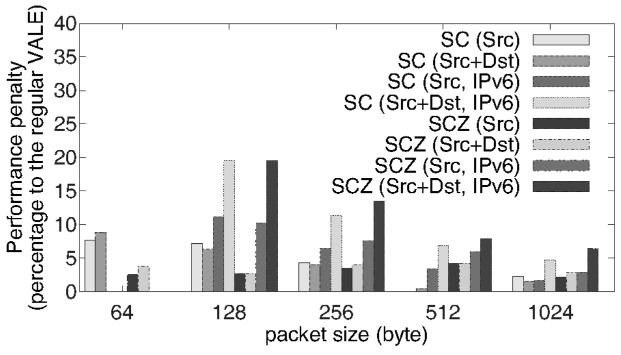

Figure 4.13: Ministack proof of concept evaluation. SC means that each packet is created separately (as opposed to re-using the same packet). SCZ means zero-copy: the packets are directly created in the memory-mapped NIC buffer rings.

To confirm this, we generated a TCP flow on top of *netmap* on a system with a 10Gb Intel NIC and a Core i7 CPU. The set-up yielded (figure 4.13) line-rate for packets larger than 128-bytes, 7.84 Mp/s (out of 8.22) for 128-byte packets, and 11.24 Mp/s (out of 14.2) for 64-byte packets. These results show that we can saturate a 10Gb pipe from user-space without having to rely on hardware offloading.

In the rest of this section we describe *MiniStack*, a proof-of-concept system that allows applications to use fast, secure, user-level network stacks.

4.2.2 MiniStack Overview

Besides high performance, allowing unprivileged applications to run either their own or a general-purpose stack in user-space means that the operating system must isolate them from each other and prevent them from sending spurious packets out into the network. The *netmap* framework provides a simple memory-mapped I/O model between the NIC and user-space, meaning that applications can (1) snoop or overwrite each other's packet buffers and (2) spoof header fields such as the IP source address.

To address the first issue, we need separate packet buffers for each application. For this, MiniStack extends VALE [20], a kernel-level, Ethernet switch which exposes netmap-based virtual ports. MiniStack assigns one or more VALE ports to an application, and forwards (copies) packets between the packet buffer of the application's port and the port that the NIC is connected to. As an initial test, we measured the throughput between a single VALE application port and a NIC for both outgoing and incoming packets, and achieved 13.7 Mp/s (outgoing) and 13.6 Mp/s (incoming) for 64-byte packets, and line-rate for larger packets.

Regarding the second problem, MiniStack once again extends VALE's functionality in order to implement a packet filter. Applications register the desired source IP address (which must match one of the IP addresses on the system), protocol and port numbers. Note that this assumes IP (both IPv4 and IPv6 are supported), and that the protocols running on top of it use port numbers; this is likely to be true in most cases (e.g., TCP, UDP, SCTP, DCCP, etc), but if needed different mechanisms can be implemented.

Once registered, the application sends packets and MiniStack verifies that the fields in them agree with those that the application had registered, otherwise drops them. On the receive-side, MiniStack implements a demultiplexing mechanism, delivering the packets to the "right" application/port based on the registered port numbers.

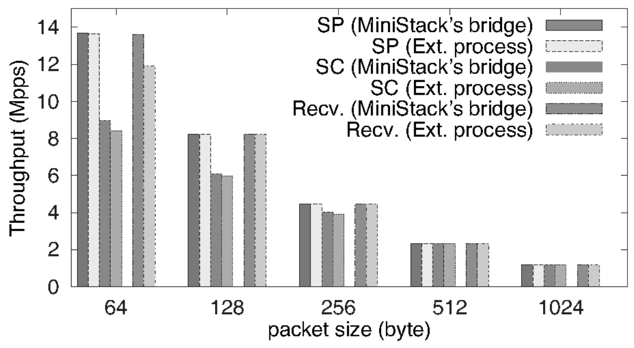

Figure 4.14: Ministack performance.

With all of this in place, MiniStack achieves 9.6 Mp/s for 64-byte packets and line-rate for most other packets sizes on receive; and 8.4 Mp/s for 64-byte packets and line-rate for packets equal to or larger than 512 bytes on transmit (see figure 4.14).

4.3 FlowOS

In this section we describe FlowOS. The key insight into this work is that while a large portion of the network processing we do is in terms of flows (for instance, think of a simple monitoring application that keeps per-flow statistics), programmers spend a significant portion of time developing code to deal with packets. FlowOS abstracts away from this packet-centered world, presenting programmers with flows as the main primitive. Thus, FlowOS provides a framework for easily implementing flow-based processing modules.

In this section we provide a detailed description of the design and implementation of FlowOS, and further include a FlowOS manual.

4.3.1 Motivation

The use of middlebox processing services is increasingly popular as modern enterprises needs become more ubiquitous to improve security and performance of their networks. Nowadays, these middleboxes consist mainly in specialized expensive box devices plugged to the network and carry out one crucial network functionality at the time such as load balancing, packet inspection and intrusion detection. In the context of the european project CHANGE [9], the Flowstream platforms [13], which are a set of programmable switches and x86 servers, are deployed on the network and susceptible to receive input traffic at any time. These platforms can be along the path between the source and the destination as they can be off-path (traffic attraction). Received traffic must be processed by user-defined processing functionality running inside. We propose in this short paper an overview of FlowOS, an incremental deployable architecture based around the notion of flow processing platform. Using FlowOS, users can remotely instantiate different kind of processing in the same platform at the same time and on the same flow (Figure 4.15).

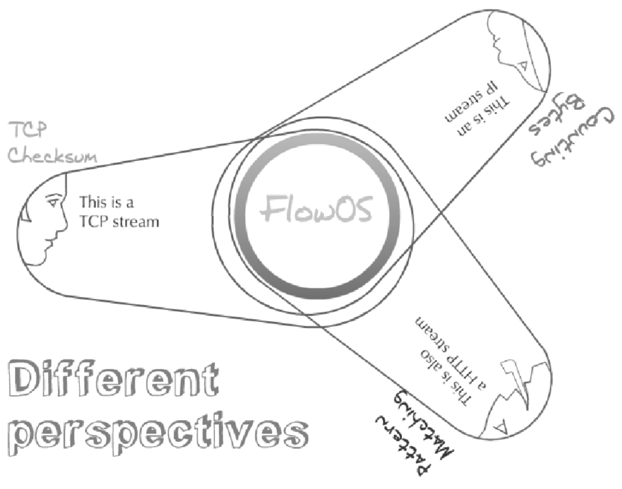

Figure 4.15: Different perspectives of FlowOS.

4.3.2 Overview of FlowOS

The original Internet architecture lacks the concept of a flow, treating traffic as independent packets. Moreover, Operating Systems naturally deal with packets, which are an artifact of the network - apart from the ease of moving them around in the network, there is nothing ”logical” about packets: applications manipulate flows. As soon as we start reasoning about ”applications” or anything else in the network, we need flows.

In current OSes, flows are really materialized at the socket interface in user space. That is not good for high-performance in network flow processing. FlowOS is based on new programming models and abstractions that are better suited to processing network traffic as flows. We design FlowOS with both an execution model that gives us much higher performance, and much greater flexibility as to the definition of what a flow is, than

[9]http://www.change-project.eu

the socket interface.

Each input packet matching some criteria defined by the user is placed in a specific structure which is shared between all processing modules [10] that interact in a parallel manner with this flow. In fact, all these processing modules are placed inside a single pipeline named processing pipeline, where they can see the same flow as multiple independent flows, without having to bother about even the existence of other bytes (Figure 4.16). Thus, the potential performance benefits are twofold. First, it provides an easier parallelism through a better adapted programming model (each processing module in our execution pipes has a "window of flow data" within which it does what it wants, with the synchronization between neighbouring processing modules being done through "increasing/shrinking" those windows). Second, the packet-to-flow-to-packet translations is done efficiently by the system itself (in a packet oriented system, this must be done by the modules themselves, resulting in possibly things being done several times).

When a user defines a flow, FlowOS registers an RX handler to the input port to capture incoming IP packets. A *packet classifier* on the input side, which uses the same parameters defined on OpenFlow [21] to classify packets, is responsible for putting IP packets into appropriate flows. It comes with a set of protocol parsers, which are responsible for parsing IP packets and constructing virtual streams for a flow. In FlowOS, a flow contains one or more streams where each stream represents a protocol. On the output side, FlowOS has a *packetizer*, which is responsible for repacketizing data. If some processing modules changes the content of a stream, then the packetizer has to reflect these changes to packet headers since processing modules do not know about underlying packets. It then sends IP packets to the TX handler, which subsequently sends them to output interfaces.

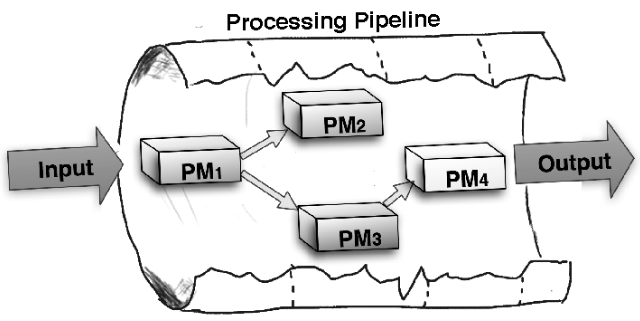

Figure 4.16: Example of processing pipeline, where PM_1 processes first received bytes and releases them to PM_2 and PM_3 that run concurrently. Finally, PM_4 is served by PM_3.

FlowOS constructs flows in the shared kernel space in order to avoid the overhead of copying IP packets from kernel to user space. Each stream consists of a sequence of virtual buffers delimited by the `start` and `end` pointers in SK_BUFFs [11]. A Flow enters into the processing pipeline at one end called the `tail` and leaves it at the other end called the `head`. At a certain point in time, a flow is characterised by its `head` and `tail` pointers.

[10]We define a processing module as one network functionality such as load balancing.

[11]SK_BUFFs are the buffers in which the linux kernel handles network packets.

A flow is then fed to a *pipeline* of processing modules where streams get processed by one or more processing modules. A processing module (PM), which is also a kernel module, processes a specific stream of the flow (e.g., IP, TCP, application). Each PM runs as an independent kernel thread to process the stream of a flow. Note that a PM thread sleeps when there is no data to process in a stream.

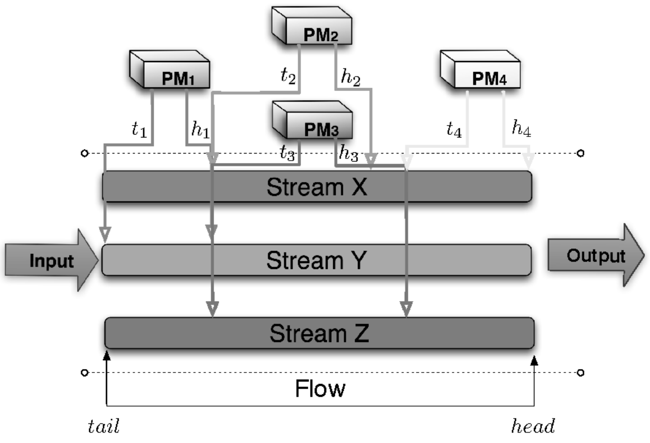

Figure 4.17: Pointers management inside the processing pipeline.

FlowOS allows users to configure PMs to run concurrently or sequentially or a combination of them. Since streams are shared among PMs, it is important to synchronize PMs in order to access streams in right order. It is the administrator's responsibility to define the correct order and inter-dependency of PMs. In a processing pipeline, a PM i has access to a particular section "window" of the flow being processed. This window is delimited by h_i and t_i pointers. Figure 4.17 illustrates relationships among these PMs pointers inside the processing pipeline. When a set of PMs are processing concurrently (PM_2 and PM_3 in Figure 4.17), the last one of them to move its h_i pointer from a byte has to release it (PM_3 releases data to PM_4 in the illustration). Different processing modules have been written and tested for FlowOS. The preliminary results show that these modules work perfectly in concurrent mode on FlowOS.

4.3.3 Manual

This section provides a reference manual to FlowOS (plese see figure 4.18 for a detailed diagram of how the system works).

The CLI is used to interact with FlowOS, which uses the NetLink socket interface to send commands to the FlowOS controller. The FlowOS controller is responsible for creating, deleting, modifying flows, attaching or detaching PMs to flows, communicating with local and remote PMs, etc. FlowOS RX handler captures incoming IP packets from the NIC, classifies packets to flows, and parses IP packets to realize streams. A flow is then fed into a processing pipeline for processing. A processing pipeline consists of one or more PMs, which process the flow either sequentially or in parallel or a combination of the both according to the user's specification. Once a flow has been processed by all the PMs of the processing pipeline, FlowOS TX handler sends IP packets (after reconciliation) out to appropriate output interface. The encapsulated UDP server enables FlowOS to communicate with remote FlowOS instances. It also allows FlowOS to be managed

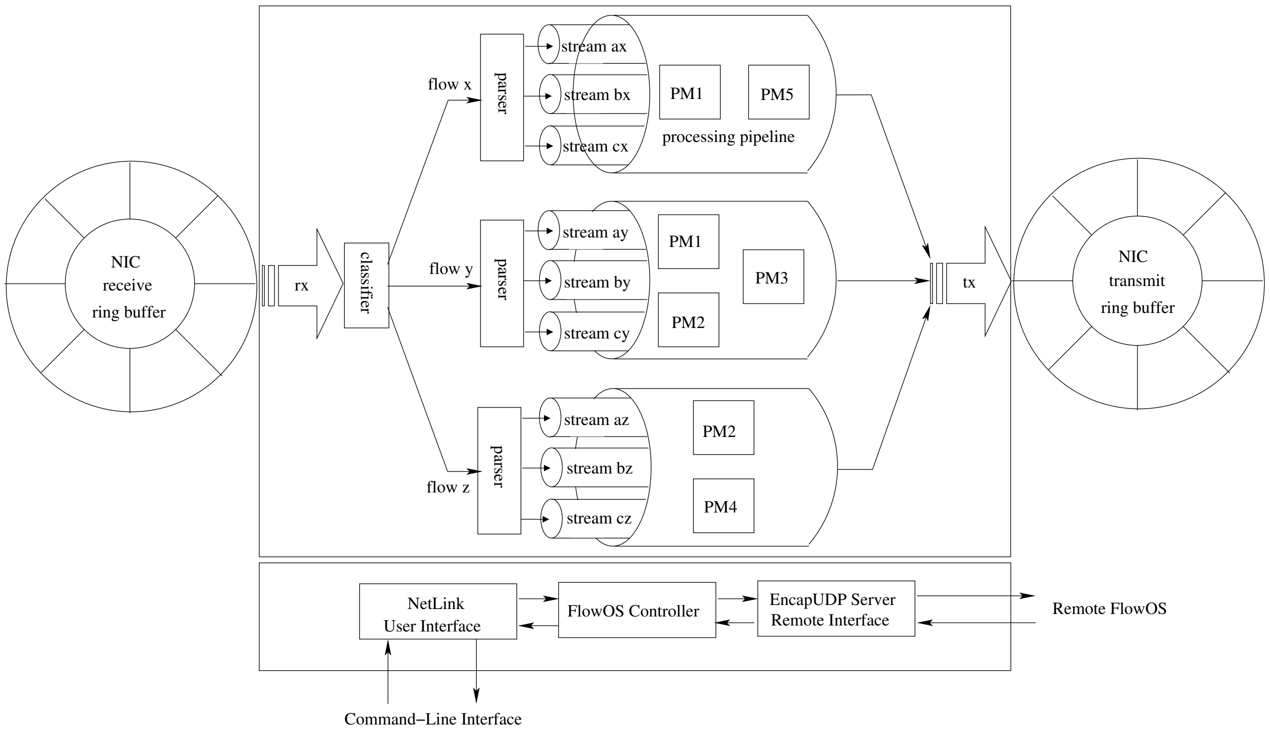

Figure 4.18: FlowOS architecture.

remotely.

4.3.3.1 FlowOS Directory Structure

FlowOS has the following directory structure. The FlowOS root directory contains bootstrap.sh, configure.ac, Makefile.am, README files and doc, include, src sub-directories. The include directory contains FlowOS header files necessary for writing processing modules and the user interface program. The doc sub-directory contains user manual and documentation for FlowOS developers. Finally, the src sub-directory contains FlowOS source code.

The src sub-directory is further organized into cli, kernel, protocols, and pmodules sub-directories. The cli sub-directory contains a simple command-line user interface program called cmd. The command-line cmd can be used to configure and manage FlowOS and processing modules. Section 4.3.3.4 explains currently available commands, their syntax, and functions.

The kernel sub-directory contains FlowOS source files which implements NetLink socket user interface, encapsulated UDP socket interface for inter-FlowOS communication and remote user interface, FlowOS controller, and FlowOS RX and TX handlers.

Since FlowOS needs to parse a protocol to construct streams to be processed by the PMs. It implements protocol parsers for IP, TCP, UDP, and all the supported application protocols. The protocol sub-directory contains source files for protocol parsers.

Finally, the pmodules sub-directory contains all the PMs. This is the default source directory for FlowOS PMs. The source code of each PM is located in a sub-directory named after PM. Currently, we have two simple PMs called ipcsum and tcpcsum, which verifies IP and TCP checksums respectively of each IP packet and each TCP segment.

4.3.3.2 Building FlowOS

In order to build FlowOS, one needs Linux autotools including autoconf, automake, and Linux headers. In FlowOS root directory, we have a shell script bootstrap.sh, which runs autoconf to generate configuration file. Once the configuration file is generated, users can configure FlowOS by running the configure script optionally specifying the kernel build path as

```
./configure [--with-kbuild=<path to linux build directory>]
```

The default kernel build directory is `/lib/modules/`uname -r`/build`. The configuration script checks dependencies and generates necessary Makefiles. Then the user issues make to build FlowOS and all the processing modules.

```
make
```

Note: FlowOS has currently been tested for Linux kernel 3.x.x only.

4.3.3.3 Running FlowOS

FlowOS can be configured to be loaded automatically when Linux boots, but for the time being we prefer to load it manually using `insmod`.

```
insmod $FLOWOS_ROOT/src/kernel/flowos.ko
```

When FlowOS is started, it initializes itself by creating NetLink socket interface for console users, encapsulated UDP server which runs on port 54321, FlowOS controller thread which processes local and remote commands, and FlowOS TX handler thread which is responsible for reconciliation of TCP/IP packets before sending out to the network.
Note: FlowOS does not create any flow, start any PM, or register an RX handler by default.

4.3.3.4 Managing FlowOS

FlowOS comes with a simple command-line interface program "cmd" that resides in the `src/cli` subdirectory of the FlowOS root directory. Note that we do not install FlowOS and processing modules in the standard modules directory because the code has not been tested thoroughly. So, we leave the CLI program in the source directory and it requires the environment variable `FLOWOS_ROOT` to set to the FlowOS root directory to work properly. Note that FlowOS can be managed locally or remotely. All the FlowOS commands take an optional `--host` parameter, which specifies the IP address of the FlowOS host where the command will be executed. By default, all commands are executed at the localhost.

4.3.3.4.1 Creating a flow The first thing to do with FlowOS is to define a flow. Currently, a flow can be defined using five attributes (borrowed from OpenFlow) namely IP source address, IP destination address, TCP/UDP source port, TCP/UDP destination port, and the input port. Note that all of these attributes are optional except the input port. The syntax of create flow command is

```
./cmd create flow [--name <flow name>] [--sip <source IP>]
      [--dip <destination IP>] [--tcpsrc|--udpsrc <source port>]
      [--tcpdst|--udpdst <destination port>] --inp <input port>
```

Here name is a unique name given to the flow, if not specified FlowOS generates one like `flow<n>` where n is an integer, sip and dip specify the source and destination IP addresses respectively, tcpsrc/udpsrc and tcpdst/udpdst specify the TCP/UDP source and destination ports respectively, and inp specifies the input port for the flow. Note that TCP and UDP ports are mutually exclusive.

When a flow is created, FlowOS registers an RX handler to the input port of the flow. Subsequent flows defined for the same port, does not need to register RX handler.

4.3.3.4.2 Creating processing pipeline Once a new flow is created, we have to define a processing pipeline for it so that PMs can be attached to the flow. Note that PMs can process a flow sequentially or in parallel or in combination of them. In order to create a processing pipeline, currently we have to specify the number of sequential stages of the pipeline. Within a given stage, we can have any number of concurrent processes. The syntax of create pipe line command is

```
./cmd create pipeline --flow <flow name> --stages <n>
```

Here `<flow name>` is the name given when the flow is created and `<n>` is the number of sequential processing stages.

4.3.3.4.3 Attaching PMs to flows A flow needs to be configured with PMs to be processed with. A pm can be attached to one or more flows at the same time. When a PM is attached to a PM a new thread of the PM is instantiated and placed on the processing pipeline at appropriate position. The syntax of attach pm command is

```
./cmd attach --pm <PM name> --flow <flow name> [--pos <n>]
```

where `<PM name>` is the name of the PM to be attached to flow mentioned in `<flow name>` and the optional pos specifies the sequential stage of the processing pipeline where the PM is to be placed. By default, the position is at stage 0.

4.3.3.4.4 Deleting a flow If a flow is no longer necessary to process, it should be removed from FlowOS in order to save memory and processing. When a flow is deleted from the system, the associated processing pipeline is removed from the system automatically. The PMs attached to the flow are stopped and if the flow solely employs a PM, the associated kernel module is unloaded from the memory. If the flow is the sole user of an input port, the RX handler is unregistered from the port when the flow is deleted. The syntax of the delete flow command is

```
./cmd delete flow --name <flow name>
```

where the `<flow name>` is the name given to the flow during creation.

4.3.3.4.5 Examining FlowOS status Once FlowOS is up and running, it might be necessary to examine the status of FlowOS or a PM. FlowOS show status command can be used to retrieve current status of FlowOS of a running PM. The syntax of show status command is

```
./cmd show status [--flow <flow name>] [--pm <PM name>]
```

When the command is issued without any arguments, it displays FlowOS status. Currently, FlowOS status consists of the list of configured flows, the list of running PMs, and the number of IP packets processed so far. The detailed information of a flow can be obtained by specifying the `--flow` parameter (currently not implemented). Finally, the status of PM can the obtained by specifying both `--flow` and `--pm` options.

4.3.3.5 Stopping FlowOS

Since FlowOS is a loadable kernel module, it can only be stopped by unloading it from the memory by using `rmmod` command.

```
rmmod flowos
```

When FlowOS is unloaded, it closes NetLink socket interface and encapsulated UDP server socket. All the flows are deleted, which subsequently stops all the PMs, unloads associated kernel modules, and unregisters RX handlers. FlowOS then stops the FlowOS TX handler and finally the FlowOS controller.

4.3.3.6 Discussion

This version of FlowOS does not support modification of flow contents and packet reordering. We have been testing features like adding new content to a flow, deleting bytes from a flow, and replacing bytes in a flow. We also have optional packet reordering for TCP flows. We will release revised version shortly with these features.

4.3.4 Conclusion and future work

FlowOS is a novel OS that we have started to build from ground up with the purpose of supporting flow processing. After design and implementation phases, we have already implemented some processing modules that seems working perfectly in a parallel way. Next steps of the present work include performing a more thorough and rigorous analysis, using complex scenarios and comparing to the usual per packet processing.

4.4 ClickOS

So far we have presented a number of processing module technologies developed in the CHANGE project. The first, netmap and VALE, aimed at providing fast packet I/O to user-level, and so provide the basis for the other processing module systems. Ministack leverages this work, creating a framework that allows different module to safely co-exist while having each run a different (perhaps customized) network stack. FlowOS provides a different approach, giving its users the ability to program in terms of operations to flows, rather than packets. ClickOS, on the other hand, provides support for a virtualized processing module, providing good isolation and the possiblity of fast migration of functionality.

4.4.1 System Description

In this section we describe ClickOS, beginning with some background regarding Xen and Mini-OS before moving on to discussing ClickOS' control and data planes. Note that we use the terms virtual machine, guest domain (or domain) and domU interchangeably.

4.4.1.1 Background

Xen is split into a privileged domain called dom0 that (among other tasks) controls the hypervisor and hosts device drivers[12], and guest domains, the users' virtual machines (also knows as domUs).

In order to move perhaps untrusted code out of dom0, Xen introduced the notion of *stub domains*. Stub domains are small guest domains based on Mini-OS, a minimalistic OS provided with the Xen sources. Developers link software packages against Mini-OS, in essence providing the OS' main run loop; in our case, the software package is the Click modular router. In order to build and link Click and Mini-OS we first needed to have a Linux-independent `c++` cross-compiler (Click is written in `c++`). To do so, we built a new toolchain (gcc, ld, ar, etc) that uses the platform-independent `newlibc` library instead of `glibc`. In addition, we adapted certain parts of Click to work in a Mini-OS environment instead of a Linux or FreeBSD one.

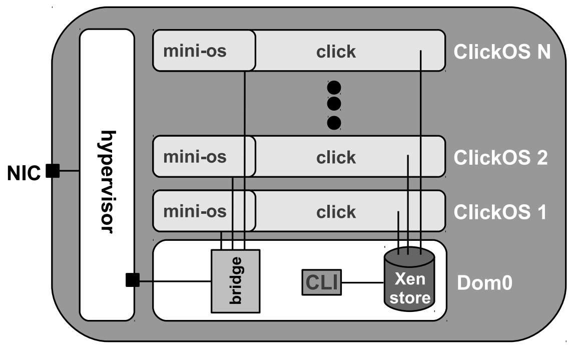

Figure 4.19: ClickOS Architecture Overview

Mini-OS itself is a paravirtualized guest OS, in that it implements the basics needed to operate in a Xen environment: grant tables for sharing memory with other domains including dom0, a netfront driver for packet I/O, event channels (Xen interrupts) and Xen bus and store drivers. In addition, Mini-OS removes some of the costs of running a full-fledged OS such as Linux. First, it has a single address space, eliminating the overheads that arise from a kernel/user separation and its resulting system calls. Further, it runs a co-operative scheduler, and so limits context switch costs.

[12]Strictly speaking, the device drivers are hosted in the driver domain, but in practice dom0 frequently also acts as the driver domain.

4.4.1.2 Control Plane

The system consists of a set of ClickOS virtual machines (vms), each composed of Click version 2.0.1 running on top of Mini-OS (figure 4.19). We chose Click because its modular architecture and large number of available modules (called *elements*) make it ideal for quickly developing middleboxes.

To run Click, users provide a configuration, essentially a text file specifying a graph of inter-connected elements. Once running, they can access read/write *handlers*, internal variables that can change the state of an element at run-time (e.g., the *AverageCounter* element has a read handler to get the number of packets seen so far, and a write handler to reset that count). Click relies on the `/proc` filesystem or sockets to provide these mechanisms; because these do not exist in ClickOS, we must provide an equivalent way of implementing them.

The ClickOS control plane consists of three parts. First, a python-based CLI runs in dom0 and takes care of, among other tasks, creating and destroying ClickOS guest domains. When a guest domain boots, a Mini-OS control thread is created (the second part of the control plane). This thread adds an entry to the Xen store, a `/proc`-like database shared between dom0 and all running guest domains (figure 4.19). The control thread then watches for changes to the entry. When a Click configuration string is written to it, it takes care of creating a new thread and running a Click instance within it, meaning that several Click instances can run within a single ClickOS domain.

The third part of the control plane consists of a new Click element called *ClickOSControl*. It talks, on one end, to all elements in a given configuration and to the Xen store on the other end. The CLI then provides users with an interface to read and write to the handlers via the Xen store and *ClickOSControl*. All these operations on the ClickOS side of things are executed in the control thread mentioned above.

4.4.1.3 Data Plane

Xen has a split network driver model, where a netback driver running in a driver domain (usually dom0) talks to hardware devices and exports a common, ring-based API; and a netfront driver running in a guest domain (e.g., ClickOS) talks to the netback driver via shared memory (i.e., the ring). This split model allows guest domains to have access to hardware devices without having to themselves host their drivers.

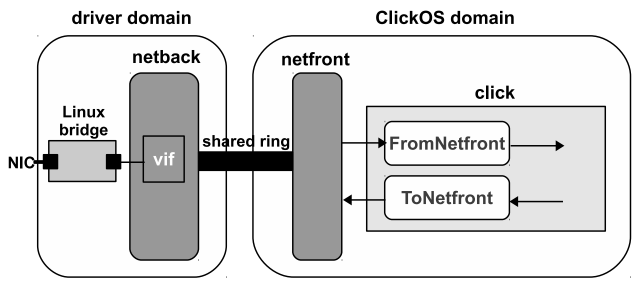

Figure 4.20: ClickOS Data Plane.

Under a typical Xen set-up, a network card is linked to a virtual network device called a *vif* via a regular Linux bridge (figure 4.20). When a packet is received, it is forwarded to the *vif* whose MAC address matches that of the packet's destination. The device then takes the packet and queues it at the netback driver. At a later point in time, one of the netback driver threads picks up the packet and puts it on the shared ring, notifying the netfront driver in the process; packets sent out by ClickOS follow a similar path in the opposite direction. Out of the box, the Mini-OS netfront driver performs rather poorly. To improve it, we introduce three mechanisms: (1) we change the driver's receive function to poll for packets from the Mini-OS thread running Click, rather than be interrupt driven, and we introduce a burst parameter to process packets in batches; (2) we re-use the grants that receive buffers are given and keep them for the lifetime of the network device (a grant is Xen's way of allowing two domains to share memory); and (3) we do a fast re-fill of request entries in the shared ring, since the netback driver cannot send packets to the netfront unless requests are available on the shared ring. We provide a performance evaluation of these mechanisms in section 4.4.2.

In addition, we add support for multiple vif devices per ClickOS domain (the standard Mini-OS netfront only supports one for the entire guest domain). In order to interact with the netfront driver, we created two new Click elements, *FromNetfront* and *ToNetfront*. The first of these takes care of initializing a network device and, each time it is scheduled, retrieves *burst* number of packets from the netfront, where burst is a configuration parameter. The *ToNetfront* element is pretty straightforward, simply calling the netfront's transmit function.

4.4.2 Evaluation

We now present results evaluating various aspects of ClickOS. We perform all tests on a couple of x86 commodity servers, each with two quad-core Intel Xeon E5620 2.4GHz processors, 24GB of memory and an Intel x520-t 10Gb adapter. One server acts as a packet generator and sink, and the other runs Xen 4.1.2 and the ClickOS domains. We use "Kp/s" to mean thousands of packets per second.

4.4.2.1 Image Size

ClickOS is compiled with most of the available Click elements (216/282), many of the remaining ones requiring a file system to work (we are in the process of porting a simple file system to increase the number of available elements). The compressed ClickOS image is 1.4MB in size, and the virtual machine needs a minimum of 5MB to run. This shows ClickOS's small footprint: in a quick test we were able to have as many as 1,010 dummy virtual machines before errors (not related to memory exhaustion) occurred.

4.4.2.2 Start-up Times and Migration

In order to quickly instantiate network processing, ClickOS needs to be able to boot-up and be migrated fast. We begin by instructing Xen to create the domain, after which Mini-OS/ClickOS boots, and we create an entry in the Xen store for click configurations. At this point we attach any necessary network devices. Once a Click configuration arrives we launch a new thread running Click. In our tests, going through this entire process of creating a ClickOS virtual machine takes on average 100 milliseconds, with minimum times of around 70 milliseconds (see figure 4.21).

Regarding migration, ClickOS's small image size and fast boot-time mean that moving a virtual machine

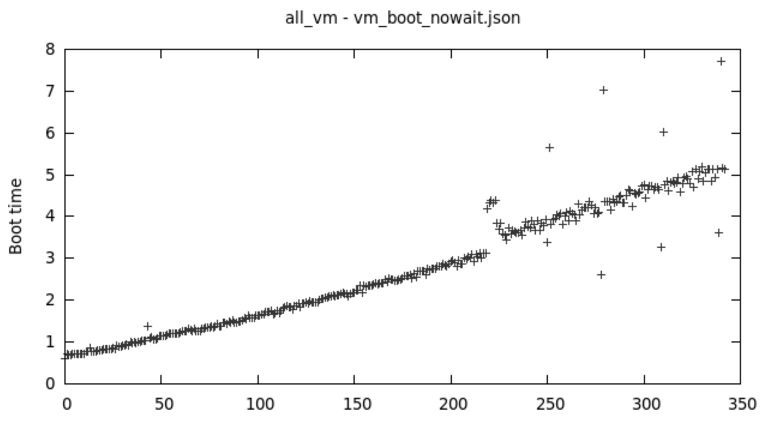

Figure 4.21: ClickOS boot times for 350 concurrent virtual machines (in seconds).

without much state (e.g., only a few forwarding rules for an IP router or a few firewall rules) takes on the order of hundreds of milliseconds.

4.4.2.3 Data Plane Performance

We evaluate the performance of the ClickOS receive path with and without the three netfront optimizations mentioned in section 4.4.1.3 (polling for packets, re-using grants and quickly re-filling request entries).

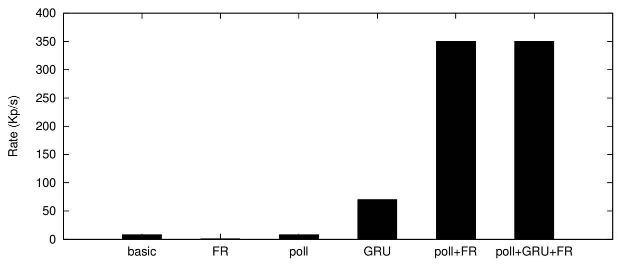

Figure 4.22: ClickOS throughput. GRU stands for grant re-use, and FR for fast refill

With maximum-sized packets and no optimizations, the Mini-OS netfront achieves a steady rate of only 8 Kp/s (figure 4.22).The next three bars are for rates when only one of the three mechanisms is used, and show that an important gain can be had by including only grant re-use (70 Kp/s). Using poll and fast refill yields the best performance, equivalent to when all three mechanisms are in place. We hypothesize that this is because at this point the bottleneck is somewhere else, and so the gains that adding grant re-use gives go unnoticed. The final rate of about 350 Kp/s, equivalent to 4.2Gb/s, represents an important increase with respect to the basic netfront driver; this is also better than the 2.9Gb/s we experienced and was reported in [26] for a Linux vm.

Introducing similar mechanisms in the transmit path resulted in a send rate of 383 Kp/s, roughly 4.6 Gb/s. Having instrumented the ClickOS netfront's transmit path we know that ClickOS can offer as many as 18 million packets per second to the shared ring. This, and the receive results, point to bottlenecks in the netback driver which we are currently working to remove.

4.4.2.4 Processing Delay

Beyond raw throughput, for ClickOS to be viable as a middlebox platform it should not add significant amounts of delay to the packets it processes. To quantify this, we measured the time it takes for a packet to be sent from a source computer and be bounced back by a computer running ClickOS. We then conducted the same experiment using a Linux domU with Click.

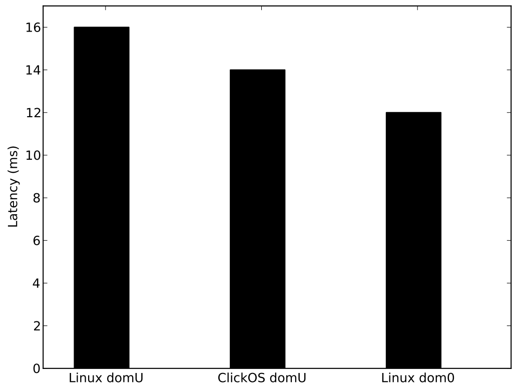

Figure 4.23: Average delay overheads.

Figure 4.23 shows that, as expected, measuring latency at dom0 results in the smallest delays. Also as expected, running packets through to a guest domain incurs further delay, with a Linux domU fairing slightly worse than ClickOS; these figures agree with the results in [40], which describe a 10ms delay coming from Xen's credit scheduler.

4.4.3 Middleboxes

We consider ClickOS to be a good platform for developing middleboxes and for porting existing ones to it in relatively short time frames. To show this, we have put together a wide range of middleboxes. We ported the first two from existing c/c++ code, a process that took roughly a day or two each. The remaining ones we created in even less time by using existing Click elements. Note that the memory footprint for all of these is small: they can all be run within a vm to which only 5MB of memory are allocated (this figure assumes that the middleboxes have little state in them, e.g., a few forwarding rules for the IP router).

Software BRAS: We implemented a Broadband Remote Access Server (BRAS) stateful middlebox that can do session set-up, including PPPoE discovery, LCP options negotiation, basic PAP authentication, and PCIP negotiation and IP assignment. We're currently improving the prototype to include other features such as traffic shaping and monitoring.

Carrier-grade NAT: We ported the code from [23] to ClickOS. This middlebox implements a scalable carrier-grade NAT with reactive load-balancing, where internal (IP, port) tuples are dynamically mapped onto their external counterparts.

Firewall: A stateless firewall that filters based on IP header fields using a tcpdump-like syntax.

DNS proxy: A transparent DNS proxy.

DHCP server: A DHCP server that can handle discover, release and request message types.

Load balancer: A simple load balancer that splits traffic across a set of interfaces based on a packet's 5-tuple.
IP router: A standards-compliant IPv4 router.

4.5 Conclusion

This chapter presented a number of technologies developed in the CHANGE project aimed at implementing the processing modules that perform the actual flow processing in platforms. Netmap and VALE provide fast network I/O and bridging capabilities, forming the basis for the other technologies. Ministack leverages VALE to provide a framework for isolated, fast, processing modules running customized network stacks in user space. Further, ClickOS gives support for fast, virtualized processing modules, while FlowOS sets the stage for easy, flow-based processing module development.
While the netmap and VALE work is rather mature and might soon reach an end, Ministack, ClickOS and FlowOS are still being actively developed; we will report on their progress in further deliverables.

5 Platform Deployment

Previous chapters described the requirements and design of the platform, both from an external and internal point of view. We now dedicate the rest of this document to describing the actual implementation of the platform that is provided in the software distribution that accompanies this deliverable.

5.1 Platform Hardware Description

As previously mentioned, a platform's hardware can consist of an entire data-center, a single rack of x86 servers connected via programmable switches or something smaller like a blade server. In essence, the software abstracts from this by only assuming the existence of some sort of programmable switch (hardware or software) that can pipe together processing modules sitting on module hosts. The only other assumption is the availability of a server able to execute Python, the programming language that the platform controller is coded in.

While the software can support different kinds of platforms, so far we have focused on *single-host* platforms, that is, an entire platform implemented on a single x86 server. The main reason behind this is that it simplifies the deployment story when trying to set-up the CHANGE wide-area testbed (see section at the end of this chapter) since it would be hard to have partners make an entire rack of computers available. Having said that, it is likely that at least for the boot camp there will be a larger version of the platform (i.e., with several servers) available.

The platform consists, then, of a single x86 server running the controller software. It has a single module host (itself), and uses Openvswitch [38], a software programmable switch that implements the Openflow protocol among others, to pipe traffic between the platform's external interfaces and the processing modules it runs.

For this first version of the platform we have chosen ClickOS as the system for implementing the processing modules. In essence, the x86 server runs Xen, and the platform controller runs in Xen's dom0, while the processing modules are each a Xen virtual machine. Further, each processing module has (at least one) virtual interface that gets added to Openvswitch when the virtual machine is created. Again, the platform's software is not bound to only ClickOS, and it is possible that in the future we will support other systems like FlowOS for instantiating processing modules.

5.2 Platform Software Description

The distribution contains a tree structure consisting of the following directories and files:

- `cmds`: The commands for the various controller daemons.
- `configs`: Contains three types of configuration files: (1) platform description files describing a platform's hardware; (2) base system description files describing the software/OS running on a platform; and (3) allocation description files, samples of the type of processing that can be instantiated on a platform.

- `core`: The main files of the platform controller.
- `daemons`: The platform daemons.
- `pms`: The various processing modules that a platform supports go here.
- `tasks`: The tasks for the various controller daemons.

In addition, the distribution comes with the file `flowstream_start.py` that takes care of starting the platform's controller (i.e., all of its daemons) and the file `flowstream_exec.py` which allows for the installation of an allocation request (i.e., network processing on the platform). Usage for this is described in the next section.

5.3 Brief User's Manual

In this section we give a brief explanation of what's required to run a CHANGE platform, including how to install its software. We further cover how to get the controller running and how to install and delete allocation requests (i.e., network processing) from it.

5.3.1 Requirements

The platform controller is implemented in Python, so a somewhat recent version of it is needed (at least version 2.6). The platform's current version uses ClickOS, for which Xen 4.1.2 is required. Further, the software expects an Openvswitch switch to be in Xen's dom0, and for Xen's networking scripts to be adapted such that the virtual machines' virtual network interfaces are added to the switch. At least one physical interface (the one that the platform is supposed to receive packets from) should be added to this switch.

5.3.2 Installation

Begin by unpacking the distribution tar ball with:

```
tar -xvzf flowstream.tgz
```

Next, you'll need to modify a few things in the following platform configuration file:

```
configs/platformdesc/platdesc_singlehost_xen.xml
```

Mostly all you need to change are the paths to "xencfg" and "clickoslib" to point to the directories containing the ClickOS Xen configuration and front-end API files, respectively. You might also want to modify the directories that daemons log messages to (look at the `log_dir` tags).
With this in place, you're now ready to run the CHANGE platform.

5.3.3 Running the Platform

To start with, set the `PYTHONPATH` environment variable to point to the platform's code:

```
export PYTHONPATH=[path/to/platform/srcs]
```

The distribution comes with a script for starting all the necessary daemons:

```
usage: flowstream_start.py [description file] [(module_host=yes|no)]}
```

Make sure you run the process as root and that you use a full path for the description file (this is the file you modified in the previous section). Set the `module_host` parameter to "no" (this is to distinguish from a platform that contains multiple servers/module hosts). If successful, you should see console output stating that the daemons are running. Note that the daemons will log messages to the directories specified in the platform description file.

Once the controller is running, we're ready to ask the platform to instantiate network processing. The first step would be to create an allocation request configuration file (refer back to 2.0.4 and have a look at `mirrortest.conf` in the allocation configuration directory for examples). Once this is done, we can the following command to install it:

```
usage: python flowstream_exec.py [command] [(params)...]
```

For example, to install a configuration called "test.conf", we would run:

```
python flowstream_exec.py InstallAllocation test.conf
```

If successfull, the command will print out the id of the (now running) allocation. We can then later use this id to delete the allocation from the platform:

```
python flowstream_exec.py DeleteAllocation [allocation id]
```

5.4 Wide-Area Network Testbed

In order to test the CHANGE architecture in a realistic environment (e.g., one containing middleboxes, delays, jitter, etc) we have started to deploy CHANGE platforms at various partner sites. We had initially considered using Planetlab for this, as this would have given us a larger number of platforms and perhaps more geographical diversity. However, Planetlab's boxes are rather restrictive: they run very old distributions, making it hard to install some of the basic packages that our software rely on (even installing a somewhat recent version of Python is challenging!); they do not allow users to tinker with network interfaces (even virtual ones like tap devices); and clearly they cannot support a system like ClickOS which requires Xen. Still, the controller software includes support for running a "Planetlab platform" using user-level Click to instantiate processing modules in case it is needed in the future.

At this point in time we have set-up or are in the processing of setting up CHANGE platforms at the following partner locations:

- NEC
- PUB

- UCL-BE
- TUB

It is likely that in the future we will add other platforms.

6 Conclusions

In this deliverable we covered the requirements, design and main implementation of the CHANGE platform. In terms of hardware, a CHANGE platform can range from an entire data center, a rack of servers interconnected by a programmable switch, a blade server, or to a single x86 server. The platform's controller software can potentially support all of these, exposing a common interface so that different types of CHANGE platforms can inter-operate.

This document further provided a detailed description of the controller as well as a brief user's manual depicting how to set up a platform and have network processing installed on it. So far the software distribution focuses on a single-server platform running ClickOS for its processing modules, but it would be relatively easy to extend this platform to setups containing more servers (as we might do for the CHANGE boot camp). Beyond ClickOS, the deliverable discussed other processing module technologies developed within the project: netmap, VALE, Ministack and FlowOS.

As future work, we're currently in the process of setting up a wide-area testbed composed of CHANGE platforms at (initially at least) four different geographical locations. The platform software is already running at one of these, and we are currently finishing the install at other locations with the goal of testing the CHANGE architecture as described in deliverable D4.4.

A Software Releases

In this brief appendix we provide an overview of which parts of the code described in this deliverable will be released (or already have been) as open source. In cases where we won't release a particular piece of work, we give reasons why.

- **Netmap**: The source code to Netmap is available as open source at `http://info.iet.unipi.it/~luigi/netmap/`
- **VALE**: The source code to VALE is available as open source at `http://info.iet.unipi.it/~luigi/vale/`
- **FlowOS**: The source code to FlowOS will be released next year. The reason for not doing so just yet is that we feel that it is not stable enough at this point to release to the public (though it's getting there!).
- **Flowstream platform software**: This software will be released as open source next year; this would give us time to make it more mature and document it better.
- **ClickOS**: NEC, the developer of ClickOS, might exploit the work carried out as a product. As a result, it is possible that parts of the work will not be released as open source (on the other hand, this will likely result in CHANGE having direct impact in industry). Part of the ClickOS work is built on the GNU license (e.g., modifications to Xen), so we're likely to release at least parts of it.
- **Ministack**: The story is similar to that of ClickOS (though it would be in any case too early in Ministack's development to release any of it as open source): NEC might exploit this work for one of its products.

Bibliography

[1] http://virt.kernelnewbies.org/MacVTap.

[2] http://www.linux-kvm.org/page/VhostNet.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R.Neugebauer, I.Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*. ACM Press, October 2003.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP'03*, Bolton Landing, NY, USA, pages 164–177. ACM, 2003.

[5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference ATC'05*, Anaheim, CA. USENIX Association, 2005.

[6] M. Carbone and L. Rizzo. An emulation tool for planetlab. *Computer Communications*, 34:1980–1990.

[7] Fabio Checconi, Paolo Valente, and Luigi Rizzo. QFQ: Efficient Packet Scheduling with Tight Guarantees. *IEEE/ACM Transactions on Networking*, (to appear, doi:10.1109/TNET.2012.2215881), 2012.

[8] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of USENIX SOSP 2009*, Big Sky, MT, USA, October 2009.

[9] Yaozu Dong, Jinquan Dai, Zhiteng Huang, Haibing Guan, Kevin Tian, and Yunhong Jiang. Towards high-quality I/O virtualization. In *SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 12:1–12:8. ACM, 2009.

[10] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of ACM CoNEXT 2008*, Madrid, Spain, December 2008.

[11] M. Fiuczynski et al. An Extensible Protocol Architecture for Application-Specific Networking. In *Proc. USENIX ATC*, 1996.

[12] Michael Goldweber and Renzo Davoli. VDE: an emulation environment for supporting computer networking courses. *SIGCSE Bull.*, 40(3):138–142, June 2008.

[13] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. Flow processing and the rise of commodity network hardware. 39(2):20–26, March 2009.

[14] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Mart
'ın Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38:105–110, July 2008.

[15] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of ACM SIGCOMM 2010*, New Delhi, India, September 2010.

[16] Ryan A. Harper, Michael D. Day, and Anthony N. Liguori. Using KVM to run Xen guests without Xen. In *2007 Linux Symposium*.

[17] M. Honda et al. Is it Still Possible to Extend TCP? In *Proc. ACM IMC*, pages 181–192, 2011.

[18] Eddie Kohler, Robert Morris, Benjie Chen, John Jahnotti, and M. Frans Kasshoek. The click modular router. *ACM Transaction on Computer Systems*, 18(3):263–297, 2000.

[19] John R. Lange and Peter A. Dinda. Transparent network services via a virtual traffic layer for virtual machines. In *16th Int. Symposium on High Performance Distributed Computing*, HPDC'07, pages 23–32, Monterey, California, USA, 2007. ACM.

[20] Luigi Rizzo. VALE, a Virtual Local Ethernet. `http://info.iet.unipi.it/~luigi/vale/`, July 2012.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. 38(2):69–74, March 2008.

[22] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in xen. In *USENIX Annual Technical Conference ATC'06*, Boston, MA. USENIX Association, 2006.

[23] V. Olteanu and C. Raiciu. Efficiently migrating stateful middleboxes. In *Proc. of ACM SIGCOMM Poster and Demo Session*, 2012.

[24] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. In *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[25] Himanshu Raj and Karsten Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *Proc. of HPDC'07*, pages 179–188, 2007.

[26] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *Proc. ACM VEE, 2009*, VEE '09, 2009.

[27] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference ATC'12*, Boston, MA. USENIX Association, 2012.

[28] L. Rizzo. Revisiting network I/O APIs: the netmap framework. *Communications of the ACM*, 55(3):45–51, 2012.

[29] L. Rizzo. Revisiting Network I/O APIs: The netmap Framework. *ACM Queue*, 10(1):30–39, Jan. 2012.

[30] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *Infocom 2012*. IEEE, 2012.

[31] Luigi Rizzo. Email to qemu-devel mailing list re. interrupt mitigation for hw/e1000.c, 24 july 2012. https://lists.gnu.org/archive/html/qemu-devel/2012-07/msg03195.html.

[32] Luigi Rizzo. Email to qemu-devel mailing list re. speedup for hw/e1000.c, 30 may 2012. https://lists.gnu.org/archive/html/qemu-devel/2012-05/msg04380.html.

[33] Luigi Rizzo and Giuseppe Lettieri. The VALE Virtual Local Ethernet home page. http://info.iet.unipi.it/∼luigi/vale/.

[34] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review*, 42(5):95–103, 2008.

[35] Sunay Tripathi, Nicolas Droux, Thirumalai Srinivasan, and Kais Belgaied. Crossbow: from hardware virtualized nics to virtualized networks. In *1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09, pages 53–62, Barcelona, Spain, 2009. ACM.

[36] VMWare. Esx networking performance. http://www.vmware.com/files/pdf/ESX_networking_performance.pdf.

[37] VMWare. vSphere 4.1 Networking performance. http://www.vmware.com/files/pdf/techpaper/PerformanceNetworkingvSphere4-1-WP.pdf.

[38] Open vSwitch. Open vswitch. `http://openvswitch.org/`.

[39] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. Xenloop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 109–118, New York, NY, USA, 2008. ACM.

[40] Jon Whiteaker, Fabian Schneider, and Renata Teixeira. Explaining packet delays under virtualization. *SIGCOMM Comput. Commun. Rev.*, 41(1):38–44.

[41] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX 2008 Annual Technical Conference, ACT'08*, pages 15–28, Boston, Massachusetts, 2008. USENIX Association.

[42] Binbin Zhang, Xiaolin Wang, Rongfeng Lai, Liang Yang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Evaluating and Optimizing I/O Virtualization in Kernel-based Virtual Machine (KVM). In *NPC'10*, pages 220–231, 2010.