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

D3.2: Flow Processing Platform Design and Early Implementation

Due date of deliverable: December 31, 2011

Actual submission date: September 28, 2012

Start date of project	1 October 2010
Duration	36 months
Lead contractor for this deliverable	University Politehnica of Bucharest
Version	Final, September 28, 2012
Confidentiality status	Public

Abstract

The Internet has grown over the last twenty years to the point where it plays a crucial role in today's society and business. However, its 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 infeasible: operators are paid to run stable, always available networks which is anathema to deploying new mechanisms. The CHANGE architecture aims to re-enable innovation in the Internet and is based around the notion of a *flow processing platform*. This document describes our experience so far in designing and implementing the CHANGE flow processing platform.

We first introduce the common API that all CHANGE platforms must comply to, describing the set of primitives that are important to flow processing. In addition, we present the basic architecture of the platform's controller which is predicated on a set of inter-communicating daemons. One of the primary functions of a platform is to decide how to allocate its resources to requests for processing, and this document describes work on resource allocation algorithms for this purpose.

We present netmap, a novel framework for high performance packet processing in userspace that we have developed. netmap allows users to enjoy the ease and safety of userspace development while also achieving very high performance - a simple application using netmap can source 14 million packets per second with just one core running at 1Ghz.

We then describe ClickOS and FlowOS, two novel operating systems that we are building from ground up to offer beter support for network processing: ClickOS aims at providing great isolation with very little overhead while FlowOS rethinks the architecture of the OS in the context of the flow processing.

Finally, we present the design, implementation and early evaluation of three primitives supported natively by the platform to help applications: load balancing, flow migration and routing.

Target Audience

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	Costin Raiciu, University Politehnica of Bucharest
Project Co-ordinator	Adam Kapovits, Eurescom
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2012 Participants in project CHANGE

Executive Summary

The Internet has grown over the last twenty years to the point where it plays a crucial role in today's society and business. However, its 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) achitecture 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 (attracting flows to off-path platforms is also feasible and discussed in a separate deliverable) 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.

The core of the CHANGE architecture relies, then, on these Flowstream platforms. In this document we describe such platforms in detail. We begin by giving a set of platform requirements that arise from the various use cases reported on in deliverable D2.3. From there we sketch out the basic hardware of a platform, including a programmable switch (e.g., Openflow), and a set of commodity-hardware servers called module hosts that contains the processing modules that do the actual flow processing. It is worth noting, however, that the CHANGE architecture does not dictate any particular implementation of the platforms as long as they conform to the *Flowstream* API, which we describe in detail in this document.

The last and principal component of a Flowstream platform is the software controller. The controller, while logically a single entity, is in actuality designed as a set of inter-communicating daemons. The separation of the controller into a set of daemons is presented in this document and 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.

Crucial to the optimal functioning of the platform is its ability to efficiently allocate its resources to the flow processing requests it receives. We present a novel resource allocation algorithm together with its evaluation in Chapter 2; the algorithm computes allocations for CHANGE platforms (containing tens of servers) in a few seconds; it also scales to very large platforms finding accurate allocations but in longer time periods.

Flow processing code runs in processing modules. These modules must isolate different users while offering a convenient programming environment and high performance (these are described in Chapter 3).

As our first choice for a processing module, we present the design, implementation and evaluation of netmap, a novel framework for high performance packet processing in userspace that we have developed. netmap

allows users to enjoy the ease and safety of userspace development, in contrast with in-kernel programming where software bugs have the potential to crash the entire OS. netmap achieves very high performance: a simple application using netmap can source 14 million packets per second with just one processor core running at 1Ghz. Further, we have integrated libpcap with netmap allowing unmodified applications to achieve 2x-4x speedups.

We then describe ClickOS, a novel operating designed to offer the great isolation provided by virtual machines with low overhead. ClickOS is a minimalistic OS running in Xen that achieves these goals. Preliminary tests show that we can instantiate as many as 1000 virtual machine instances on a single physical box, two orders of magnitude more than existing OSes.

Finally, we discuss FlowOS that rethinks the architecture of the OS in the context of the flow processing. In this deliverable we describe the FlowOS concept, its basic primitives and our early implementation.

Chapter 4 of this document presents the design, implementation and early evaluation of three primitives supported natively by the platform to help applications. The first primitive is load balancing. We design, implement and evaluate a novel load balancing algorithm that allows the CHANGE platform to overcome the lack hashing support in OpenFlow by leveraging prefix matching rules. The second primitive is flow migration: here we describe a novel algorithm that leverages the properties of flows to allow fast migration with zero downtime. The third primitive is IP routing: here we show how the Zipf distribution of bytes to flows in Internet traffic can be used to efficiently offload routing to the OpenFlow switch in the CHANGE platform.

List of Authors

Authors	Luigi Rizzo (UNIPI), Vladimir Olteanu (PUB), Costin Raiciu (PUB), Felipe Huici (NEC), Armin Jahanpanah (NEC), Mohamed Ahmed (NEC), Gregory Detal (UCL-BE), Steve Uhlig (TUB), Laurent Mathy (ULANC), Mehdi Bezahaf (ULANC), Abdul Alim (ULANC), Yves Deville (UCL-BE), Pham Quang Dung (UCL-BE)
Participants	PUB (Editor), UCL-BE, ULANC, NEC, TUB, UNIPI
Work-package	WP3: Flow Processing Platform Design and Development
Security	RESTRICTED (RE)
Nature	R
Version	1.0
Total number of pages	109

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 (attracting flows to off-path platforms is also feasible and discussed in a separate deliverable) 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.

The core of the CHANGE architecture relies, then, on these Flowstream platforms.

Platform Requirements. Flowstream platforms must meet a number of requirements:

- **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.

In this document we describe how we can build platforms that meet these goals, in particular the design of their software controller, the design of module hosts as well as primitives for the platforms.

We begin by giving a set of requirements for the platform, and give an overview of the basic hardware structure of a platform. From there we describe the controller itself, briefly describe its implementation and then focus on resource allocation algorithms that decide how to assign the available resources of a platform to incoming requests from users.

Next we analyze how a platform can run processing for multiple concurrent users while offering good isolation, utilization, performance and ease of programming. This deliverable reports on our early work on ClickOS and FlowOS, two novel operating systems we are designing to achieve isolation and performance for network processing.

We also describe *netmap* , a fully-developed novel framework that permits very fast packet I/O even for userspace applications on commodity operating systems. *netmap* has been implemented in FreeBSD for several 1 and 10 Gbit/s network adapters. In our prototype, a single core running at 1 GHz can send or receive 14.88 Mpps (the peak packet rate on 10 Gbit/s links). This is more than 20 times faster than conventional APIs.

The last part of this report details our work on enabling primitives needed by most network processing software. We describe how Flowstream platforms can support load balancing, flow migration and routing efficiently.

1.1 Flow Processing Platform Overview

As mentioned, Flowstream platforms need 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 [16][15][22]. These, coupled with programmable commodity hardware switches such as OpenFlow switches and virtualization technologies such as XEN [9], form the basis of one type of Flowstream platform (see figure 1.1).

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 [31]), netmap running in FreeBSD, a minimalistic OS such as ClickOS or FlowOS 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

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

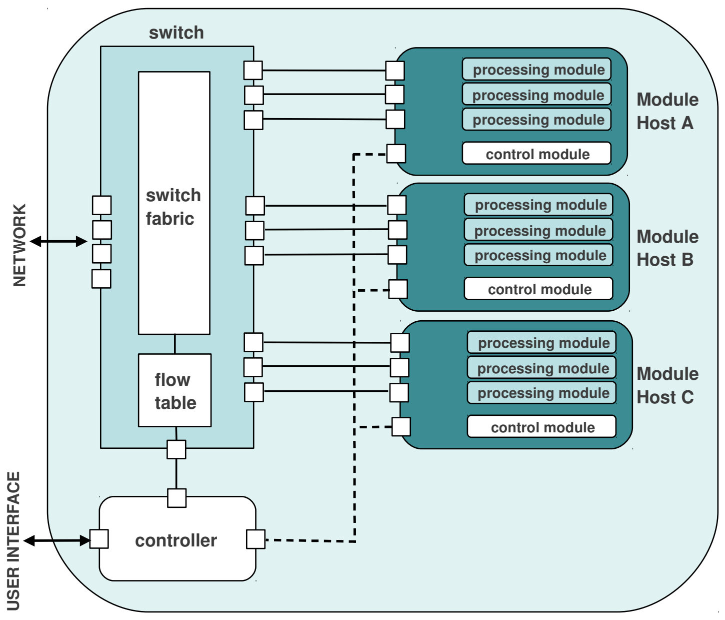


Figure 1.1: Flowstream Platform Overview.

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.

2 Platform Primitives

CHANGE platforms comprise one or more programmable switches and many more servers. In this chapter we discuss the common, basic primitives that CHANGE platforms need to implement in order to carry out their work. We begin by describing the Flowstream API: a set of function prototypes that any CHANGE platform should comply to, whether it is comprised of a large number of servers interconnected with Openflow switches or consist of a single computer (section 2.1.

Beyond this API, we discuss sets of basic functionality that improve the way platforms do network processing. First, a fundamental property of flow processing is the ability to scale up and down in response to varying traffic loads. To begin with, traffic needs to be **load balanced** among the processing modules. Openflow switches promise to deliver such functionality out-of-the-box, but current models fall short on two accounts. Ideally we should be able to select a traffic aggregate and balance it across the target hosts; the trouble is that hashing is not supported so traditional load-balancing techniques cannot be applied to traffic aggregates. We could insert per connection rules to achieve the same effect with the controller's help, but we are constrained by the switches' flow memory. Our solution to this problem is an algorithm that leverages Openflow prefix matching to achieve very good load balancing at small cost; this is described in Section 2.2.

To scale up and down CHANGE platforms need to be able to move processing around and its associated flow state. However, traditional migration techniques perform poorly when applied to network processing. We propose a new approach to migration that takes into account the fundamental properties of network processing, and provides a preliminary simulation analysis in section 2.3.

To support routing in CHANGE platforms we explore the design of a mixed hardware/software router in Section 2.4. The main idea is to have the Openflow switch forward the bulk of the traffic, and process the rest of the packets in software. The challenge here again is the limited rule memory of Openflow switches: the part of the forwarding table that can be offloaded to the switch is rather limited. We propose a novel algorithm that carefully selects the flow entries to be handled by the Openflow switch. We show through several large-scale traces that Internet traffic follows the Zipf property and experimentally show that it is possible to leverage this distribution to provide fast routing as a primitive in CHANGE platforms.

2.1 Formal Primitives: The Flowstream API

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 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.

2.1.1 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.

2.1.2 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.

Note that the current platform prototype implements the high-level API only.

2.1.3 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.

2.1.4 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; here's an example:

```
BASESYS=xen.clickos

ft :: FromNet("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.

2.2 Load Balancing

When dealing with very large amounts of traffic that require a certain type of processing, a single processing module (e.g., an x86 machine) may not be enough. We've found that instantiating multiple servers that implement the same functionality while relying on the Flowstream platform's OpenFlow switch to evenly split traffic across them is a sensible solution.

2.2.1 Requirements

Our platform aims to cater to large amounts of traffic originating from a wide spectrum of IP addresses. As such, there are some special requirements that we need to satisfy:

- The traffic must be distributed evenly across the machines, proportionally to their capacities.
- The solution must be reactive. A machine may end up processing more than its fair share of the traffic due to fluctuations in traffic volume. When that happens, balance must be restored by the controller by redistributing the traffic without causing any disruptions.
- Adding or removing machines from the mix must not cause any disruptions.

- The platform's controller must not be a point of contention.
- The number of OpenFlow switch rules must be kept to a minimum.

2.2.2 Relevant OpenFlow features and limitations

OpenFlow switches process traffic according to a limited number of rules. The rules are chosen and installed by an out-of-band controller [39]. Individual packets that don't match any of the rules are forwarded to the controller pending its decision regarding their fate.
For the purpose of matching traffic, certain header fields are checked against predefined values specific to a rule. In addition to this, the source and destination IP addresses can have an arbitrarily long suffix wildcarded. In short, OpenFlow supports prefix matching for IPs and all-or-nothing matching for every other field.
Traffic matching a rule is processed according to the rule's actions, including:

- forward,
- drop,
- alter header field,
- add/strip/alter VLAN tag
- etc.

In theory, per-flow rules could be used for load-balancing and would provide great control, but this solution is constrained by the rule memory available in switches, currently ranging from a few hundreds to a few thousands. Furthermore, even if the flow table were sufficiently large, the controller's intervention would still be required for the first packet of every flow, adding extra latency and making it a point of contention.
To allow scale, rule count must not depend on the number of flows; ideally it should only depend on the number of server machines used.

2.2.3 Existing approaches

One common way of splitting traffic is hashing the 5-tuple of each packet. For instance, Equal Cost Multi-Path (ECMP) [56] splits flow across equal cost routes to the destination by choosing an outgoing route with a hash of each packet's 5-tuple. Version 1.0 of the OpenFlow standard [3], and consequently all commercially available OpenFlow switches, do not support hashing. The newly-released version 1.1 of the standard [4] allows for a ”switch-computed selection algorithm”, leaving decisions regarding its actual functionality up to the vendor; it is unclear what hashing functionality the switches will support.
5-tuple based hashing ensures that all packets belonging to a TCP flow will be forwarded to the same box, thus allowing stateful middleboxes. However, even if 5-tuple hashing were implemented, it is not sufficient for most middleboxes due to additional application-level constraints.

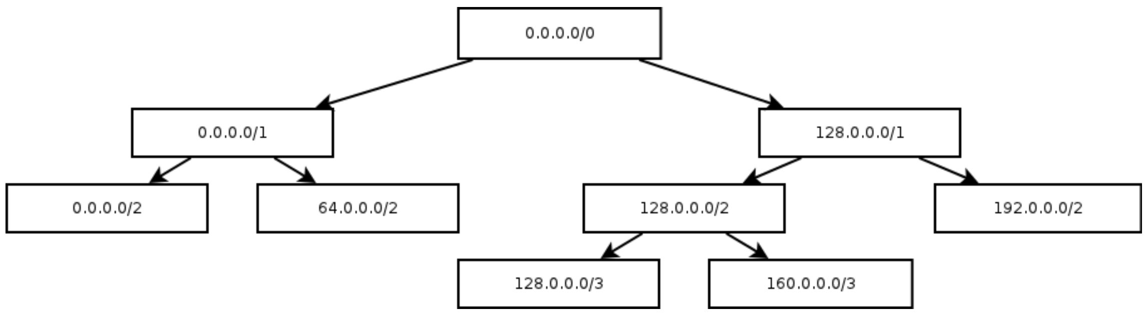


Figure 2.1: A small binary tree.

Take network address translation as an example, with a host initiating a TCP connection from private IP address X and port x. The current best common practices document requires NATs to allocate the same external IP Y and port y regardless of the destination address and port of the connection [21] (as long as these are not repeated). This is called endpoint-independent mapping and it aims to allow peer-to-peer applications to learn their external IP and port [21]. To implement this behaviour with our architecture we should load balance based on the source IP and port, rather than the 5-tuple. Next, consider scaling a data-center firewall with hundred of thousands of rules; a natural way to distribute this on many machines is to split rules based on destination addresses (and possibly port numbers). This requires load balancing based on the destination address rather than the 5-tuple. In short, load-balancing needs to move beyond the 5-tuple and be applicable to different fields in the payload.

A load-balancing algorithm making heavy use of prefix matching was recently proposed by Wang et al.[53] assuming traffic is uniformly distributed across IP addresses, and that the sum of server capacities is a power of two; these assumptions rarely hold in practice [43] [50] [54].

2.2.4 Our algorithm

To implement load balancing with a small number of rules we use the IP prefix matching support in OpenFlow switches. The basic idea is to use prefix-matching to split the traffic into large chunks. The IPv4 address space can be represented using a binary tree, with 0.0.0.0/0 as the root, 0.0.0.0/1 and 128.0.0.0/1 as its children etc. Splitting a node always yields two children whose network mask is longer by one bit. The first child will have the newly-acquired bit set to 0, while the other one will have it set to 1. The entire address space is fully covered by the leaf set of any arbitrarily-constructed tree.

Our algorithm needs accurate information regarding each leaf's load. It is run periodically by the controller and uses load data provided by the machines processing the traffic. The machines use the time between two consecutive runs to sample the traffic and gather said data.

In a nutshell, our algorithm greedily tries to assign as much traffic to the least loaded server that can fit it. It allocates large chunks of traffic to middleboxes using a best-fit strategy. Chunks that are too large to fit are split. The algorithm starts off with the root of the IP address space and selects a "default" server, which is assigned every unallocated prefix. It greedily attempts to assign the largest leaf to the computer that can accommodate the most traffic. If the leaf is too large to fit, it is split. The algorithm stops when the "default" computer is no longer overloaded.

We define imbalance as the proportionally most "overworked" computer's load over the load it was supposed to have. The algorithm attempts to keep the imbalance under a user-configured maximum.

This is a simplified and unoptimized version of the algorithm:

```
last_computer = pop(computers)
last_computer.load = root.load
while not overloaded(last_computer)
        computer = pop(computers)
        leaf = pop(leaves)
        if fit(leaf, computer)
                assign(leaf, computer)
                last_computer.load -= leaf.load
        else
                (left, right) = split(leaf)
                push(leaves, left)
                push(leaves, right)
        push(computer, computers)
```

`pop` always returns the computer with the most unused capacity for processing traffic, or the leaf with the most traffic.

A computer is considered to be overloaded when its load exceeds the load it was supposed to have by at least the same percentage that is indicated by the maximum imbalance.

In order to reduce the number of assignments, once the largest leaf's load falls below a certain threshold (dubbed the chunk threshold), the algorithm stops looking for leaves that will "fit" and starts looking for leaves that will not overload the computer. While the algorithm works towards the same maximum imbalance regardless of the chunk threshold's size, a small chunk threshold favors a more even chunk (leaf) distribution, while a large chunk threshold further reduces the number of chunks needed.

2.2.5 Implementation

The load-balancer assigns traffic to servers according the current traffic distribution and the servers' computing power, given as configuration parameters. It runs on a controller box and initially assumes traffic is uniformly distributed across the IP address space and assigns it to servers such that the imbalance is less than 1.1. As time passes the controller learns the real traffic distribution; if the imbalance grows the algorithm is rerun and the resulting rules are installed on the switch.

The controller is based on NOX and supports any stateless packet inspection/processing scheme, such as firewalls. Both controller and servers run vanilla Linux 2.6.

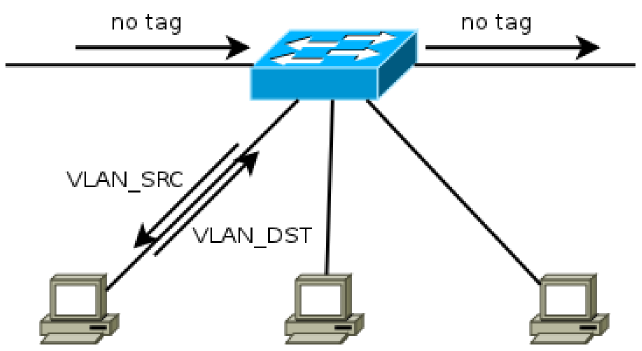


Figure 2.2: Path taken by a packet matched by its source IP.

To learn traffic distribution, the controller could use per-rule switch counters, but these only provide information about prefixes in use, not about traffic distribution *within* those prefixes. In cases where traffic is concentrated, per-rule counters are insufficient: a single overloaded prefix could be split many times to reduce imbalance.

To estimate traffic accurately we have implemented a Linux kernel module that runs on each server and estimates traffic distribution within its assigned prefixes. The accuracy is configurable and specifies the number of sub-prefixes that should be monitored. The implementation uses a high priority Netfilter hook that samples 1 in 100 packets that are forwarded by the machine and stores per-prefix counters in a radix tree. The controller monitors the traffic distribution by periodically polling servers.

2.2.6 Physical topology and OpenFlow rules

Packet forwarding. Two of the OpenFlow switch's ports are reserved for connecting to the outside world. The source IP address is used to match traffic going in one direction and the destination IP is used for the traffic going in the other direction.

Figure 2.2 depicts how a packet matched using its source IP is treated. The steps are:

(i) The switch receives the packet with no .1Q tag. Looking at the ingress port, the switch knows that its forwarding decision must be based on the packet's source IP. The packet is tagged with `VLAN_SRC` and forwarded to the appropriate machine.

(ii) The machine receives the packet and determines the direction it was traveling in based on its VLAN tag. After processing the packet, it is tagged with `VLAN_DST` and forwarded back to the switch.

(iii) The switch checks the packet's tag and forwards it out of the corresponding interface.

A packet traveling in the opposite direction receives analogous treatment by the platform.

This forwarding scheme opens up the possibility of making the machines act like routers-on-a-stick. Thus, each machine requires only one switch port.

For traffic matching purposes, only two rules are needed for each assigned prefix. Because our algorithm works by repeatedly assigning the largest chunk available, the computer that is set aside and gets the remain-

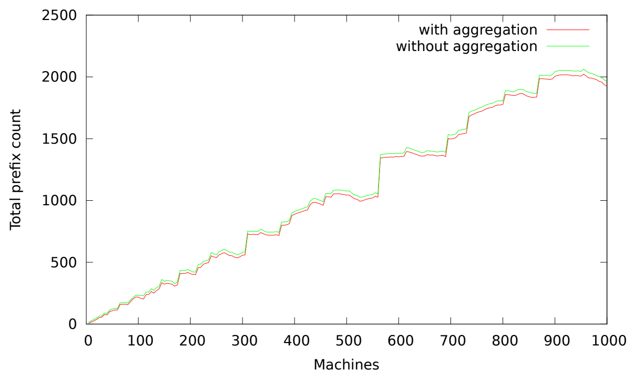


Figure 2.3: Total number of prefixes.

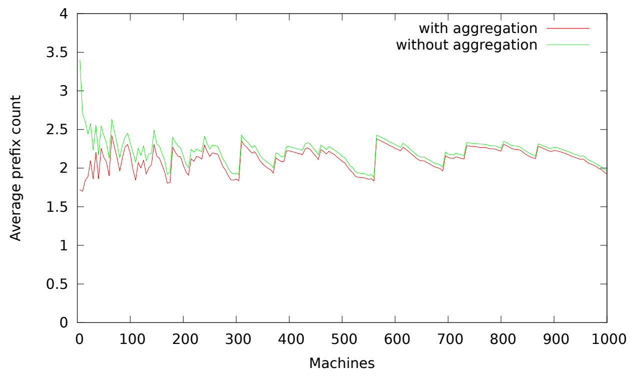


Figure 2.4: Average number of prefixes.

der leaves by default (`last_computer`) tends to be left with a large number of small chunks. These chunks can be summarized by a low priority set of rules that match any prefix (0.0.0.0/0).

2.2.7 Early evaluation and conclusions

We have used five different functions to simulate traffic distribution: 2 Gaussian bell curves, one constant function and 2 random distributions (in future work we will likely look into heavy-tailed distributions as well). All computers have the same capacity and the chunk threshold was set to 20% of the average load.

Early experiment results (Figure 2.3 and Figure 2.4) hint at a linear correlation between the number of machines and the number of allocated prefixes, yielding less than 3 prefixes per machine if the maximum imbalance is set to 1.1. As each prefix translates into 2 OpenFlow rules and typical switches have around 50 ingress ports, we can expect to use less than 300 rules for a full-blown deployment, which is well within switch's capabilities.

Figure 2.5 depicts the impact that the maximum imbalance has on the rule count. As the maximum imbalance nears a perfect value of 1, increasingly more nodes are split to achieve this goal.

The running time is less than 7ms when assigning prefixes to 1000 computers.

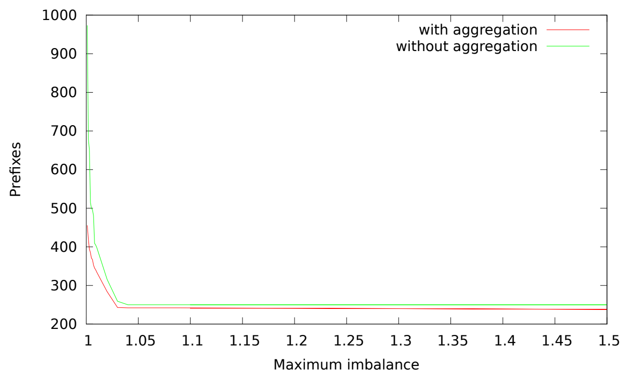


Figure 2.5: Impact of maximum imbalance on prefix count (100 machines, Gauss only).

2.3 Flow Migration

As mentioned, many services provided by middleboxes are stateful, and therefore, per-flow states must be kept. These states are modified upon processing certain packets belonging to the flow and may, in turn, affect the treatment that subsequent packets receive.

Failing to move these states when diverting traffic from one machine to another is potentially disruptive. Take TCP normalization as an example. Among other things, this particular type of packet processing makes sure that all TCP packets passing through are actually part of a connection. As such, every time a SYN packet hits the middlebox, a new flow state is created. Let's assume the controller reassigns an already-established connection to a different middlebox. The new middlebox will lack information about the connection; not having seen any SYN, it will consider all packets belonging to it to be spurious and drop them.

It is entirely possible to, for example, migrate a virtual machine that is processing traffic to a different physical machine. However, that would imply migrating the entire OS and all userland applications running on it, regardless of their relevance to flow processing. It is far more efficient to surgically target only the relevant states and leave the rest untouched.

2.3.1 Process migration algorithms

When looking for an algorithm for flow state migration, process migration algorithms [35] are the obvious starting point. They have also been successfully applied to live virtual machine migration [28].

Of particular importance to us are the downtime and the total migration time:

- During *downtime* there is no running instance of the process.
- The *total migration time* is the total time elapsed from the moment the migration is initiated until it is finished.

The three basic algorithms we will take a look at are the total copy algorithm, the demand page algorithm and the pre-copy algorithm.

2.3.1.1 Total copy

The total copy algorithm is the simplest of the three. The following steps are taken when migrating a process from host A to host B:

(i) The process is halted on A.

(ii) All of the process's pages are copied from A to B.

(iii) The process is resumed on B.

This algorithm has both the lowest total migration time (because everything is judiciously copied over) and the highest downtime.

2.3.1.2 Demand paging

The demand paging algorithm stands out when compared to the total copy algorithm because nothing is copied upfront. The process is simply halted on A and started on B. Any attempt by the process to access an untransferred page will result in it being pulled from B. The downtime is all but nonexistent, but the migration is not finished until all pages have been accessed and subsequently transferred (or the process finishes or is killed). There is also one more caveat to deal with: we can expect to find brief performance drops peppered throughout the process's execution as a direct result of page faults. Whenever a large number of page faults occur in rapid succession, the program's performance will degrade significantly.

2.3.1.3 Pre-copy

The pre-copy algorithm works by iteratively transferring pages while the process is still running on the original machine. Initially, all pages are copied. During every subsequent iteration only the pages dirtied since the last iteration are transferred. When the rate at which pages are copied equals or falls below the rate at which pages are dirtied, the process is stopped, the remaining pages are copied and finally the process is resumed on the destination machine.

2.3.1.4 The principle of locality

The two latter algorithms work reasonably well because of the principle of locality: if a page is accessed, it is likely that it will be accessed again soon. When transferring a process using the demand-paging algorithm, a large number of costly page faults tend to occur right after the transfer has begun and the process usually runs with reasonable performance after that. When using the pre-copy algorithm, the usually small number of pages that are frequently modified tend to be the ones dirtied in the last iterations before the transfer. Therefore, downtimes tend to be relatively small.

2.3.1.5 Why they don't work well for flow state migration

Per-flow states are updated in a way that depends solely on the order in which packets hit the middlebox and the principle of locality does not apply. In some cases, such as when counting the number of bytes transferred, they are updated with each processed packet. Furthermore, the states are usually independent of each other and can safely be partitioned. Therefore, it is possible for fine-grained migration to occur.

Let's assume that we're using the pre-copy algorithm to migrate some flow states. Instead of zeroing in on a frequently updated subset of states, the following things will happen between two successive iterations:

- a significant number of states will die off (because most flows are short-lived [43]),
- a significant number of new states will be established (for the same reason) and,
- a seemingly random subset of the existing states will be updated (because the principle of locality does not apply).

The demand paging approach would incur a significant overhead. The first states to be faulted across will not be the ones that are most likely to be read or modified frequently. Furthermore, most states will need to be transferred in a short amount of time, causing a big (albeit temporary) degradation in performance.

2.3.2 Our general-purpose algorithm

Like process migration algorithms, our flow state migration algorithm involves a period of downtime. Because affected traffic must be buffered during downtime, we have chosen to minimize it at the expense of total migration time.

It consists of two overlapping phases:

(i) The idle phase, which deals with short-lived flows and,

(ii) The freeze-and-copy phase, which deals with the rest.

2.3.2.1 Phase 1: The Idle Phase

The idle phase takes advantage of the fact that most flows are short-lived [43]. Rather than transfer those states, they can be kept on the original machine until they expire while, at the same time, allowing states for new flows to be established on the other machine.

This is what happens when starting to transfer states from A to B:

- All traffic is initially forwarded to A.
- A stops establishing any new states.
- All packets that hit A but don't have any matching state are forwarded to B.
- B treats the packets normally.

The two machines are basically daisy-chained. Everything the first machine can't process is handled by the other one. States associated with short-lived flows will gradually die off on A, while states for new flows will be established on B.

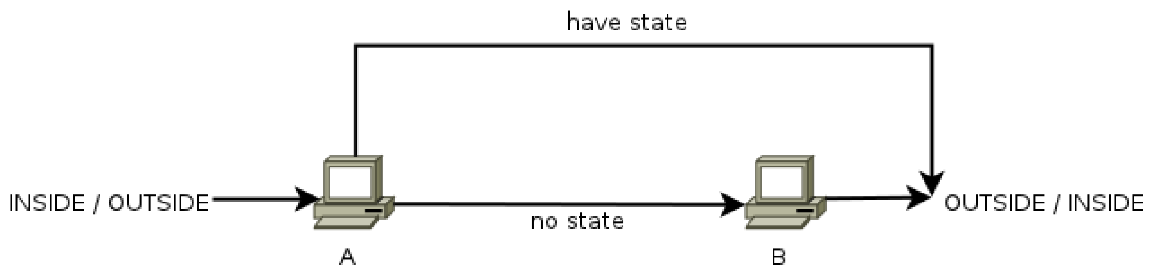


Figure 2.6: Daisy chaining.

2.3.2.2 Phase 2: The iterative freeze-and-copy phase

The second phase starts when the rate at which states expire on A falls below a certain threshold. The remaining states are less likely to die off soon and need to be transferred.

- A freezes some of its remaining states.
- All packets that would modify a frozen state (and those following them) are buffered by A.
- The frozen states are transferred to B.
- The buffered packets are forwarded to B.

If some per-flow states are linked (e.g., when running an FTP ALG), special care must be taken to not separate them. Other than that, states can be migrated in any order and at any granularity. Finer granularities incur lower downtime for each individual flow, while coarser ones speed up the migration.

2.3.2.3 Relaxing some restrictions

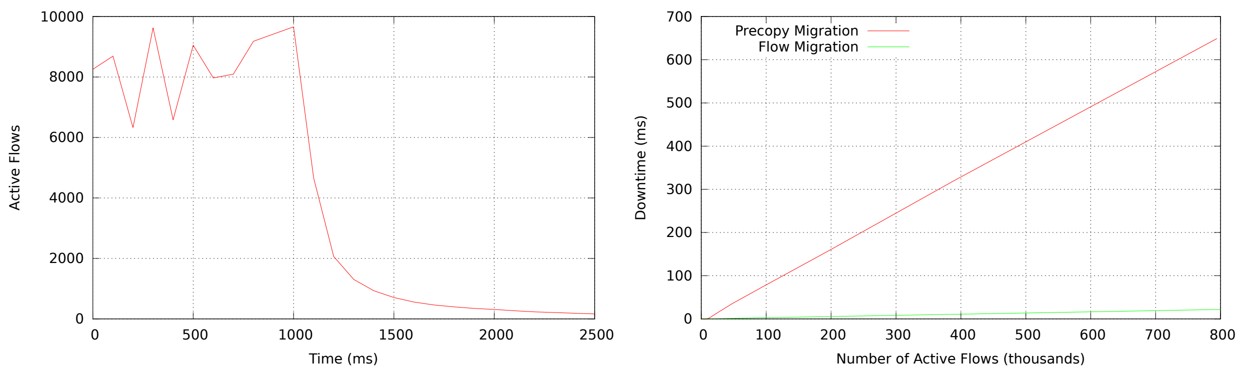
In some cases, transferring states can be greatly simplified by employing a relaxed policy toward expiration timers.

If the only thing that can happen to a flow state is for it to expire and its idle timer can be safely reset, that state's transfer is greatly simplified. The state is simply copied over and the timer is refreshed, as if a packet had gone through.

2.3.3 Experiments

We have run simulations with Poisson flow arrivals and flow durations Pareto-distributed with a mean of 200ms and a min of 100ms. Results in Figure 2.7(a) show that if we run the idle-phase during less than 1s, the number of states drops by two orders of magnitude.

We also implemented pre-copy migration as in [28] and compared the downtime of the two algorithms. Each flow state is 100B in size, and copying was done at 100Mbps during the pre-copy rounds and at 1Gbps during the stop and copy phase. Pre-copying only works when we have few states; as the number of states reaches hundreds of thousands flow, migration is two orders of magnitude faster (see Figure 2.7(b)).



(a) Number of Active Flows at A. (b) Migration Downtime

Figure 2.7: Flow Migration Simulation

2.3.4 Proof of concept: Carrier-grade NAT

We have implemented in Click a scalable carrier-grade NAT with reactive load-balancing as a proof of concept for our flow state migration algorithm. Current best practices [21] champion the use of paired IP address pooling and endpoint-independent mapping. These restrictions, combined with OpenFlow's limitations [3], have made us choose a many-to-one mapping of external IPs to servers: all flows originating from the same external IP must be migrated together.

Furthermore, our NAT has to keep global state in the form of lists of free ports. This global state must also be migrated. In our implementation this state is copied over at the start of the idle phase and reports of ports freeing up are sent regularly throughout this phase from A to B.

2.3.5 Future work

Implementing the NAT has driven our design for the flow migration algorithm. We are now finalizing the state-migration support in the NAT and will experiment with it in the near future.

Ideally, migration should be provided by the OS without changing the apps, and this is our end goal. It would allow us to migrate existing applications such as Bro or Snort without rewriting them, and giving our techniques wider applicability.

Achieving this goal is challenging. Running both source and destination processing in parallel is tricky as it requires efficient synchronization of accesses to global state. Done inneficiently, such synchronization can undo the performance benefits of flow-migration. Further, to implement daisy chaining, the OS must know which flows have state associated and which not; acquiring this knowledge without application support seems feasible only if both flow definitions and state expiration mechanisms are known to the OS.

2.4 Routing as a Platform Primitive

Can we provide routing as a primitive in CHANGE platforms? The answer is obviously yes provided that we do it all in software, but a purely software based solution would sacrifice performance and efficiency. A more appealing solution is to offload the bulk of routing to the programmable switch, while still using software to

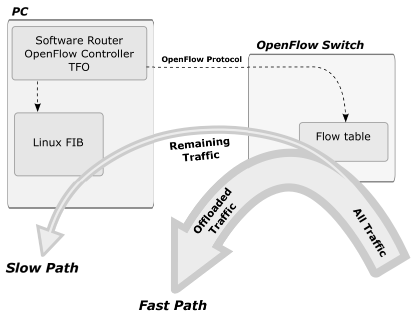


Figure 2.8: Traffic offloading system principle.

process a minority of packets - thereby capturing a balance between flexibility and high performance. In this section we show how routing can be performed efficiently within the CHANGE platform

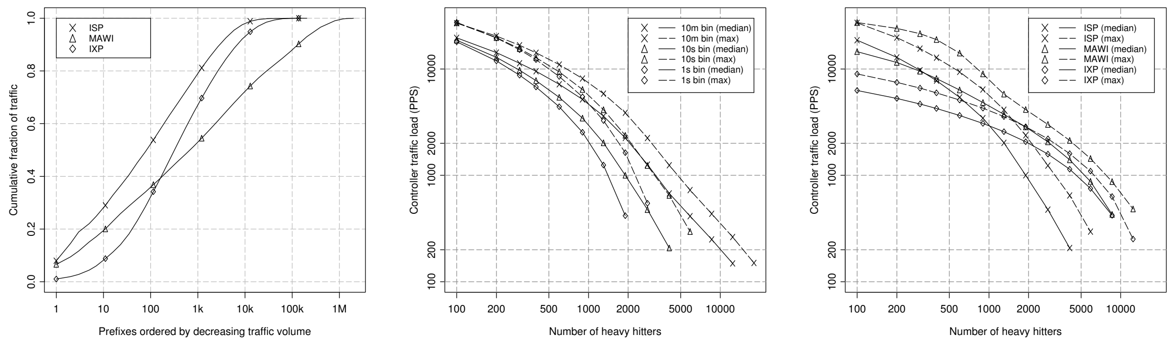
A reoccurring property of Internet traffic at various levels of aggregation is its consistency with Zipf's law [18, 54, 50, 10], i.e., the amount of traffic per flow is consistent with a Zipf distribution. A flow can be as fine as the traffic between a given IP source-destination pair or as coarse as all the traffic sent to a single network. For this reason, we postulate that a small fraction of flows capture most of the traffic, however, since heavy hitters are known to be volatile [51], it makes their selection for traffic engineering difficult [42].

We propose to utilize the Zipf properties of Internet traffic to offload most of the traffic from a complex controller to a simple forwarder. We show that by taking advantage of the properties of Internet traffic, a simple forwarder can (with limited communication overhead) be used to boost the performance of a complex forwarding system simply by offloading most of the traffic.

The principle behind such a controller-forwarder system lies in bundling the controller; a complex and flexible software-centric system, together with a forwarder; a high-performance hardware-centric packet forwarding device. This approach follows the split architecture idea, most of the control plane load should be handled by the software-oriented part of the system, while most of the packet forwarding should be performed by a fast and simple device.

Figure 2.8 illustrates the principle of the controller-forwarder system architecture with an IP router as a case study. The controller speaks routing and signalling protocols in the same fashion as route controllers on existing routers. It can take advantage of the extensive CPU and memory resources of commodity PCs. Furthermore, the controller makes use of a forwarder, a device optimized for high-performance packet forwarding, to offload most of the traffic. Through a communication channel, the controller modifies the set of heavy hitters which are offloaded and handled by the forwarder. In this paper we investigate the feasibility of such an offloading system based on Internet traffic traces.

We first show that in principle, the achievable offloading gain is in the order of more than 90 percent when offloading the traffic is contributed by 0.2% of the flows. However, the volatility in heavy hitters incurs significant communication overhead, which is contrary to our design goals of keeping the forwarder simple. To address this challenge, we propose our heavy hitter selection strategy called Traffic-aware Flow Offloading (TFO), which relies on traffic statistics over multiple time scales. Previous work [42, 51] has identified



(a) Fraction of traffic captured per prefix. (b) Impact of bin size on controller load (ISP). (c) Controller load with 10s bins (all traces).

Figure 2.9: Feasibility experiments: IP prefix popularity (a) and baseline controller load (b, c).

the volatility of heavy hitters in Internet traffic as a challenge for traffic engineering, and therefore to our controller-forwarder. In addition, we require that the benefit of introducing a forwarder is not counterbalanced by the complexity of the communication between the controller and the forwarder. Therefore, TFO trades off the overhead of modifying the set of heavy hitters against the expected increase in offloading gain.

Our results show that contrary to traditional route caching strategies [29], TFO is able to offload most of the traffic to the forwarder while keeping the number of modifications to the set of heavy hitters low. This enables us to discuss both the benefits as well as challenges that Internet traffic properties pose for router design. The contributions of this work are the following:

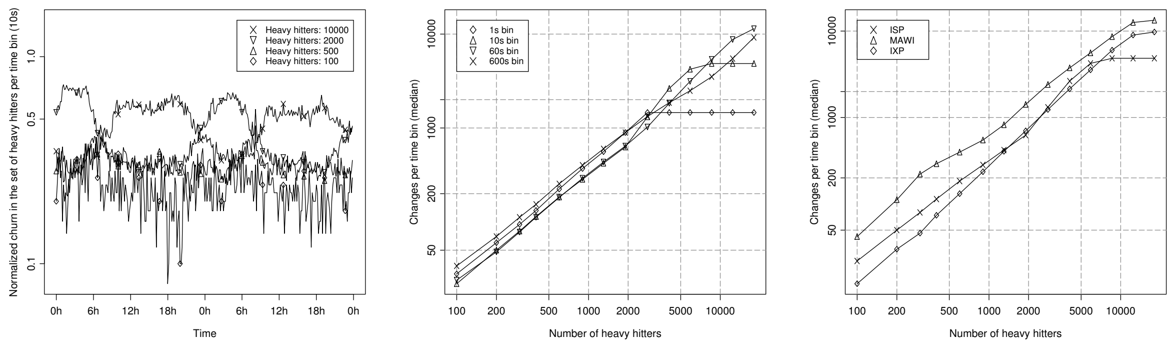
Traffic offloading: Based on traffic traces from a large European ISP, a large European IXP, and a transcontinental link we show the potential of traffic offloading and identify the challenges of heavy hitter selection strategies.

Traffic-aware Flow Offloading (TFO): We propose TFO, a heavy hitter selection strategy that leverages Zipf and the associated stability properties of heavy hitters across different time scales.

TFO and traditional caching: We evaluate TFO and compare it to traditional caching strategies. We observe a cross-over point. TFO outperforms traditional caching when the number of heavy hitters and the communication overhead have to be limited. However, when these constrains are relaxed traditional caching strategies might outperform TFO.

In addition, we release a library implementation of TFO as free software to the community [1], which we have used to implement an IP router by coupling a PC running a software router with an OpenFlow-enabled switch [39]. For a system description of our prototype we refer to [46]. The focus of this work however lies in understanding the potential of traffic offloading, which we believe is crucial not only for our example prototype but for a wide range of high-capacity, programmable packet forwarding systems.

[1]TFO software implementation: http://www.fibium.org/



(a) Churn over time (ISP). (b) Churn for different time bins (ISP). (c) Churn for the three traces.

Figure 2.10: Churn rate when using the bin-optimal strategy.

2.4.1 Feasibility study

In this section we perform feasibility experiments based on Internet traffic traces to understand the fundamental challenges and quantify the potential benefits of traffic offloading. We start by revisiting the properties of Internet traffic as seen from three different vantage points in the network. Then, we run evaluations under two optimistic assumptions: (i) future traffic is known and (ii) heavy hitter modifications are instantaneous. The results suggest a need to reduce the churn in the selected heavy hitters in order to limit the communication overhead, they also confirm previous results [51] showing that the variability of heavy hitters differs across ranks. Both observations shall be taken into account when designing a heavy hitter selection strategy such as TFO.

Datasets

We base our study on traffic traces from three different network locations (Table 2.1):

Residential ISP: The ISP trace relies on passive, anonymized packet-level observations of residential DSL connections collected at an aggregation point within a large European ISP. Overall, the ISP has roughly 10 million (4%) of the 251 million worldwide broadband subscribers [41]. The monitor operated at a broadband access router connecting customers to the ISP's backbone. The trace dates from August 2009 and covers 48 hours.

Transcontinental link: The traffic trace from a transcontinental link between Japan and USA also consist of passive, anonymized packet-level observations. The trace was collected in April 2010, covers 3.5 days, and is publicly available from the MAWI working group of the WIDE project [27].

European IXP: In contrast, the trace from a major European IXP consists of anonymized sFlow records which were captured in April 2011. The sFlow probes were configured to export sampled packets at a sampling ratio of 1:2^{14}. Due to the sampling, some underestimation of small flows is possible, and therefore there may be some bias towards the heavy hitters [11, 17].

While anonymizing the ISP and IXP traces, we map each destination IP address to its longest-matching prefix based on publicly available BGP data. The same technique could not be applied to the trace from the

	ISP	IXP	Backbone
Network	residential	hundreds of ASs	transcontinental link
Speed	1 Gbps	1.5 Tbps	150 Mbps
Duration	2 days	4 days	3.5 days
Sampling	none	$1{:}2^{14}$	none

Table 2.1: Dataset details.

transcontinental link, but fortunately the anonymization uses consistent scrambling of /24 prefixes [27].

These three datasets allow us to look at the problem of offloading from different perspectives: network location, network size, and time. This gives us confidence that our results can be generalized beyond our samples. Moreover, we note that our proposal exploits traffic properties that have already been observed in the past and are not expected to change, as the popularity of Internet flows is expected to retain its consistency with Zipf's law.

2.4.1.1 Consistency with Zipf's law

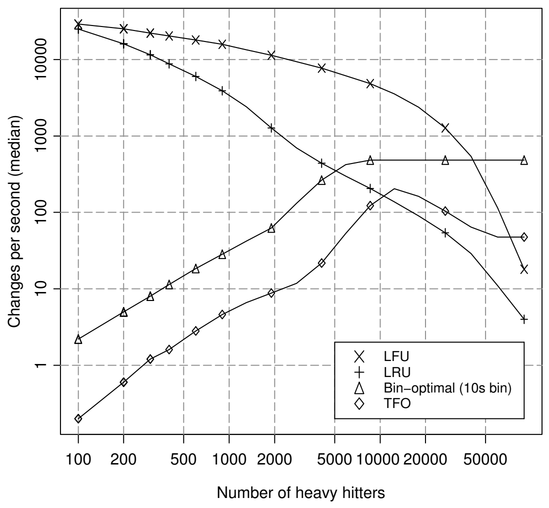
Since we aim at leveraging the Zipf properties of Internet traffic for traffic offloading, we first confirm that the three traces are consistent with Zipf's law.

Figure 2.9(a) illustrates the relative share of the total traffic as a function of the number of top destination prefixes. These results confirm that all three traces exhibit the Zipf property—a limited number of prefixes account for a majority of the traffic. However, one difference between the traces lies in the number of heavy hitters whose traffic has to be aggregated to capture a desired fraction of the traffic. For example, the most popular 10 destination prefixes of the residential ISP trace and the transcontinental trace already capture more than 20% of the total traffic, but less than 10% in the case of the IXP. With 1,000 prefixes one can capture more than 50% of the traffic. This holds for all three traces. For lower ranks the amount of traffic that each prefix is contributing decreases rapidly. The next 1,000 prefixes combined capture less than 5% of the traffic. Another difference is that the transcontinental link sees traffic for a larger number of prefixes. However, the least popular half of the prefixes contribute less than 1% of the traffic. This may be an artifact of the anonymization as described above and the resulting assumption of a flat /24 address space.

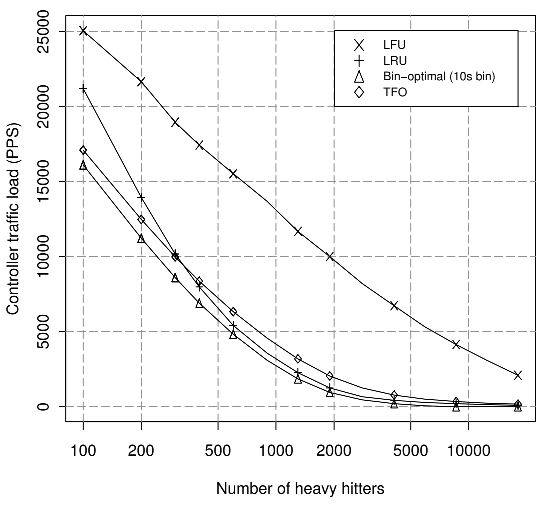
As we will discuss later, the controller typically cannot afford delegating the responsibility for too many flows to the forwarder, because of the associated communication overhead. The most relevant range seems to lie between a few hundred to a few thousand heavy hitters. The above results confirm previous work [29, 48, 51, 26], which has shown that it is possible to offload most of the traffic with a limited number of heavy hitters. However, these works do not study how to derive a heavy hitter selection strategy which trades off the expected benefit of modifying the heavy hitters against the extra work of applying those changes. Instead, previous work has focused on dynamic heavy hitter detection mechanisms [55, 12].

2.4.1.2 Remaining load at the controller

In our context, traffic offloading aims at limiting the rate at which the controller needs to forward packets by offloading most of the traffic load to a specialized forwarding device. By design, the controller is optimized



(a) Churn rate in comparison (ISP).



(b) Controller traffic load (ISP).

Figure 2.11: Controller load and churn for different strategies (ISP).

for running complex routing protocols but often it does not satisfy the requirements of packet forwarding. Some known limitations of controllers include the available forwarding bandwidth and delay.

Our goal is to limit the amount of traffic the controller has to handle in order to avoid long packet queues and packet loss. As a baseline, we now quantify the amount of traffic the controller has to handle based on our traffic traces.

For this purpose, we rely on a (practically infeasible) heavy hitter selection strategy called bin-optimal. The bin-optimal strategy assumes that future traffic is known, and that the set of heavy hitters to be used on the forwarder can be updated instantly. The bin-optimal strategy uses fixed time bins. We choose 1s, 10s, and 10 minutes as time bins to capture the varying stability properties of heavy hitters across different time scales. The bin-optimal strategy selects for each time bin the top n heavy hitters, where n is the number of entries that the forwarder keeps.

Figures 2.9(b) and 2.9(c) illustrate the traffic load in packets per second (PPS) which remains at the controller vs. the number of heavy hitters which are delegated to the forwarder, on a log-log scale. Overall, we see that even for small values of n (in the order of a few thousands), the traffic load at the controller is relatively low. When taking the ISP trace as an example, with $n = 1,000$ the median traffic load at the controller is 3,400 PPS (with a maximum of 6,400 PPS) which corresponds to a relative reduction in load of 92%. With shorter time bins the offloading gain can be further increased. Figure 2.9(b) shows both the median as well as the maximum traffic rate at the controller for the three considered time bins.

Figure 2.9(c) compares the controller traffic load for all three traces under different numbers of n and fixed 10s time bins. We observe that with all considered traces, the offloading gain increases rapidly with n. All three traces incur only up to around 1,000 PPS at the controller when offloading the top 5,000 prefixes.

2.4.1.3 Churn in heavy hitters

Heavy hitters enable significant offloading but are also challenging because of their dynamics [42, 50, 51, 54, 10]. We examine the impact of variability in the heavy hitters when delegating prefixes to the forwarder. The bin-optimal strategy relies on the foreknowledge of heavy hitters for each time bin. In this case, the churn in the heavy hitters stems entirely from the traffic's inherent variability.

We now quantify the corresponding number of changes to the set of heavy hitters between consecutive time bins, as incurred by the bin-optimal strategy. Figure 2.10(a) shows the fraction of heavy hitters that are modified for time bins of 10s, as a function of time for the ISP trace. For small values of n such as a few hundred we observe churn in the order of 15$\%$. When increasing n up to a few thousand, around half of the heavy hitters are modified. As expected, we notice that changes among the top ranked prefixes occur much less often than among the lower ranked ones. We also point out that the churn is subject to a time of day effect. Interestingly, the direction of the time of day effect inverts as the number of heavy hitters is increased. This is due to the reduced variability and traffic volume during the night.

As shown on Figure 2.10(b) for the ISP trace, relying on larger bin sizes does not significantly reduce churn. Moreover, on Figure 2.10(c), we see that even though our network observation points are quite diverse, similar trends apply with regards to churn. We explain the larger churn in case of the MAWI trace again with the assumption of a flat /24 address space.

2.4.2 Traffic offloading with TFO

In Section 2.4.1, based on Internet traffic observations, we have identified the main challenges behind heavy hitter selection strategies for traffic offloading: maintaining a high offloading ratio while limiting the changes to the set of heavy hitters. In this section we first examine how well traditional caching strategies perform in the context of the offloading problem (Section 2.4.2.1). Traditional caching strategies are not appropriate in our context due to their high churn and the fast reaction times they require. This leads us to develop and evaluate our heavy hitter selection strategy called Traffic-aware Flow Offloading (TFO). Based on our traffic traces, we show that TFO achieves low churn, one order of magnitude less than the bin-optimal strategy from Section 2.4.1.2, while achieving a similarly high offloading gain.

2.4.2.1 Limitations of traditional caching

Traditional caching techniques react on individual misses: when a cache miss occurs, the missing object is fetched and then placed into the cache. During the time interval between the event of a cache miss and the completion of a cache update, incoming packets need to be buffered or will lead to packet re-ordering or loss. The main difference among traditional caching techniques lies in the strategy to determine which cache entry to evict and replace upon a cache miss. Two common cache replacement strategies are least recently used (LRU) and least frequently used (LFU). We implement both strategies based on the optimistic assumption that cache replacement is instantaneous, to be favorable toward their offloading gain.

The curves for LRU and LFU in Figure 2.11(a) show the amount of heavy hitter changes per second as a

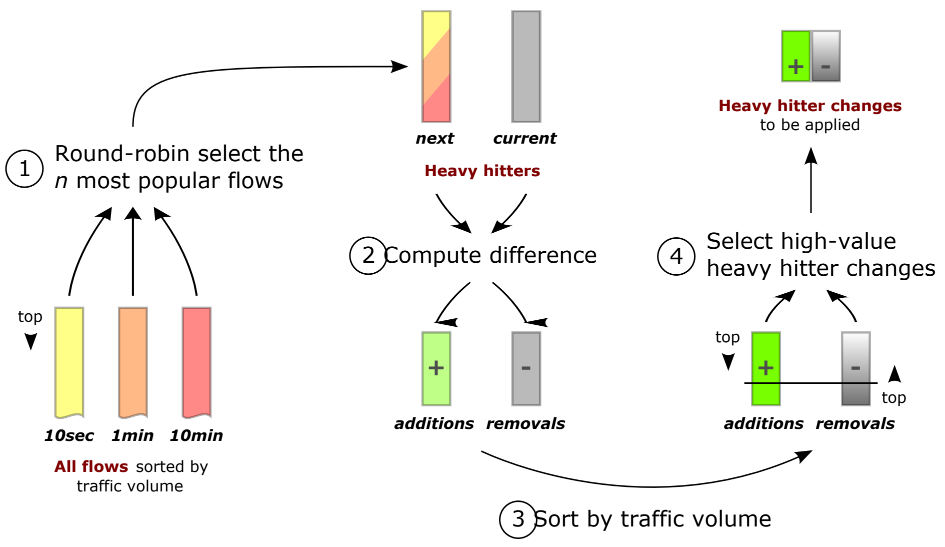
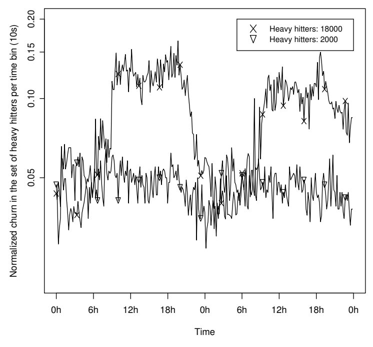
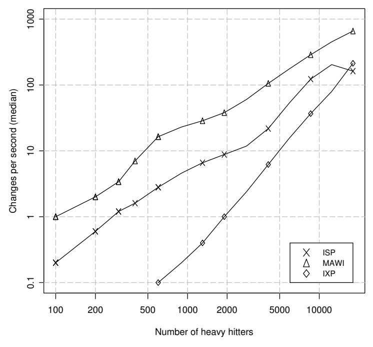


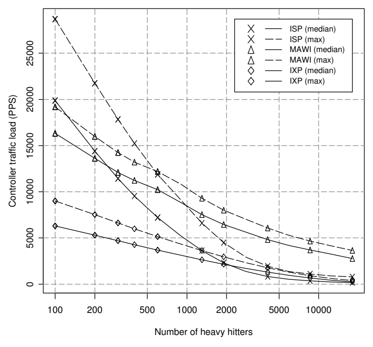
Figure 2.12: Traffic-aware Flow Offloading.



(a) Churn over time (ISP).



(b) Churn for the three traces.



(c) Controller traffic load (all traces).

Figure 2.13: TFO evaluation.

function of the number of heavy hitters, n. LRU outperforms LFU, consistent with previous work on route caching [29]. Figure 2.11(a) also shows the results for the bin-optimal strategy based on 10s bins. Both LRU and LFU imply a significantly higher churn rate as compared to bin-optimal for up to a few thousand of heavy hitters. For larger values of n, traditional caching can outperform bin-optimal in churn rate, this is when n gets close to the total number of active flows. With regards to the remaining traffic load at the controller as shown on Figure 2.11(b), LFU exhibits the worst results. For $n > 200$, all strategies except LFU result in similar remaining loads at the controller.

In summary, these results indicate that the traditional caching strategies LRU and LFU suffer from a major limitation, namely the high rate at which heavy hitters need to be changed, especially for values of n ranging up to a few thousands, i.e., the range of n in which offloading is likely to be relevant.

2.4.2.2 Traffic-aware Flow Offloading

In this section, we propose our heavy hitter selection strategy: Traffic-aware Flow Offloading (TFO). The design criteria of TFO stem from our observations in the earlier sections. The goal is to handle most traffic at the forwarder while limiting the number of changes to the heavy hitters. The idea behind TFO is to leverage the stability properties of heavy hitters, which have been shown to depend on their rank [51]. Therefore, we maintain traffic statistics at multiple time scales that enable TFO to take into account short-term as well as long-term trends for heavy hitter selection. To compute a new set of heavy hitters, we further take into account the set of heavy hitters that are currently in place, in order to trade-off the cost of changing the set of heavy hitters with the expected increase in offloading gain.

Figure 2.12 illustrates the four different steps of TFO. We start by taking top heavy hitters from each time scale in a round-robin manner, until we reach a pre-defined value of n. The round-robin pre-selection gives equal chances to heavy hitters that are highly ranked at different time scales. Second, we compute the difference between the current heavy hitters at the forwarder and the set of heavy hitters from the first step. This gives us two sets of heavy hitters: those to be installed and the ones to be removed. The last two steps are key to reduce the churn while ensuring fast reaction times. In the third step, we sort these two sets into lists according to their traffic share in inverse orders, with respect to the smallest time scale. Finally, in the fourth step, we pick such modifications in which the additions are ranked much higher than the removals, e.g., they differ in traffic share at least by a factor of 2. This fourth step gives priority to the heavy hitters which are already in place, effectively reducing the churn. Consequently, the chosen factor affects the trade-off between offloading gain and communication overhead. In future work, we want to study how to optimally determine this factor. Note, that sudden high-ranked flows can (and should) still enter the set very quickly, i.e., in one round of set computation.

2.4.2.3 Evaluation of TFO

We now evaluate the performance of TFO based on our three traffic traces. Figure 2.11(b) shows, as a function of n, the remaining traffic load at the controller for the ISP trace. The offloading effectiveness of TFO is marginally worse compared to the bin-optimal strategy. However, the bin-optimal strategy involves a churn rate which is about an order of magnitude higher as compared to TFO, see Figure 2.11(a). When comparing the evolution of churn over time of the bin-optimal strategy on Figure 2.10(a), against TFO on Figure 2.13(a), we observe that TFO changes only less than 10% of the heavy hitters between any two time bins whereas bin-optimal exceeds 70%, for the ISP trace with $n = 2,000$. When letting n grow to values as large as 18,000, we observe that the rate of churn varies heavily and follows a strong time of day pattern in cases of TFO and bin-optimal. Smaller values of n, such as 2,000, are less sensitive to time of day effects and thus show only limited variations. We conclude that an offloading system that aims too high in its offloading gain will face strong churn rate variations. The key lies in understanding the trade-off when choosing n: On the one hand, n must be large enough such that the heavy hitters can capture enough traffic to retain a low enough traffic load at the controller. On the other hand, one should strive to keep n as small as possible to keep the associated rate of churn and its variation low.

Figure 2.11(a) also compares TFO with the traditional caching strategies LRU and LFU. For $n < 10,000$, TFO outperforms both LRU and LFU in terms of churn rate. However, at around $n = 10,000$ we observe a cross-over point at which LRU starts to outperform TFO. At such large values of n, the rate of churn also exhibits strong variations over time.

Figure 2.13(b) presents the median number of changes to the heavy hitters as a function of n for the three traces. We observe an impact of traffic aggregation on the rate of churn: the MAWI trace leads to the highest churn rate followed by the ISP trace, and finally the IXP one with the least amount of churn. Also, the top 500 heavy hitters in the IXP trace are so stable that their median churn is 0 over the four days. This result is important with respect to the scalability of the offloading problem, as our results suggest that the higher the traffic aggregation, the lower the churn.

Figure 2.13(c) compares the traffic load at the controller for the three different traces when using TFO. The figure shows the median and maximum controller load as a function of n, in log-linear scale. The ISP trace gains most with increasing n. This is consistent with Figure 2.9(a), where the top heavy hitters for the ISP capture a larger fraction of the total traffic as compared to the other traces. For $n = 6,000$ we observe an offloading gain of more than 99%, resulting in a median of only 500 PPS at the controller.

The MAWI trace has the widest CDF curve on Figure 2.9(a), which leads to the worst offloading gain except for very small n. Due to the anonymization technique of the MAWI trace, we consider a flat /24 address space which potentially slices heavy hitters into many /24 prefixes. Hence, the MAWI results are penalized in terms of offloading gain per heavy hitter. However, they still result in a very limited packet rate at the controller. For example, with $n = 6,000$, TFO still achieves an offloading gain of 86%, leaving only 4,200 PPS (median) at the controller.

In contrast to MAWI, the IXP trace presents a very smooth S-shaped CDF curve on Figure 2.9(a). This stems from the level of aggregation at the observation point, which represents traffic exchanged between hundreds of ASs. Figure 2.13(c) shows that for most values of n, the IXP trace leads to the lowest controller load compared to the other traces. However, as mentioned earlier, due to the low packet sampling rate of this trace we expect a bias in favor of the observed traffic share of the heavy hitters. For $n = 6,000$, TFO achieves an offloading gain of 90%, leaving less than 1k PPS (median) at the controller.

Note that real traffic traces, especially those as large as the IXP one, contain certain types of attacks such a DDoS, scans, and other types of anomalies. Our results show that despite the presence of such events, neither the churn nor the offloading gain seem affected, even when considering the maximum churn and maximum traffic load at the controller.

2.4.3 Discussion and related work

We revisited the Zipf property of Internet traffic and derived a traffic offloading strategy that achieves high offloading gain while limiting the overhead caused by churn in the heavy hitters. With recent work which aims to improve the forwarding performance of software routers [22] as well as their port density [15], we believe that traffic offloading is a viable alternative to traditional router design.

Currently, routers maintain large numbers of forwarding entries in specialized memory for fast per-packet lookups. Routers require ever increasing amounts of such specialized memory, which typically is either costly and / or energy hungry. Furthermore, the rate at which the forwarding entries need to be changed is significant, especially due to routing protocols dynamics [34, 52, 33]. For example, reported BGP update rates are in the order of tens or hundreds per second on average with peaks of several thousands per second, for about $350,000$ prefixes. Even with such a limited churn relative to the number of prefixes, the implication of BGP updates on the data plane can be an issue [33, 7]. Compared to today's BGP update rates, the dynamics in heavy hitters is much lower, both in relative and absolute numbers. Furthermore, BGP updates are expected to have limited impact on the heavy hitters as popular prefixes have been shown to be stable [25].

Our work shows that an offloading system has potential to improve router designs, by further decoupling the tasks of the route controller with those of the forwarding engines. The main advantage of the offloading approach is to limit the interactions between the control and data planes to what is strictly necessary from the viewpoint of the traffic.

By taking advantage of the forwarding capabilities of programmable forwarding engines such as OpenFlow [39], NetFPGA [36], and ATCA [8], we believe that IP router designs which leverage open source routing software and traffic offloading may challenge existing router designs. As we plan to apply traffic offloading and TFO to other domains, we intend to incorporate TFO into FlowVisor [47] to complement its slicing policy.

3 Controller Architecture

In this section we describe the basic architecture of the Flowstream platform controller.

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 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.

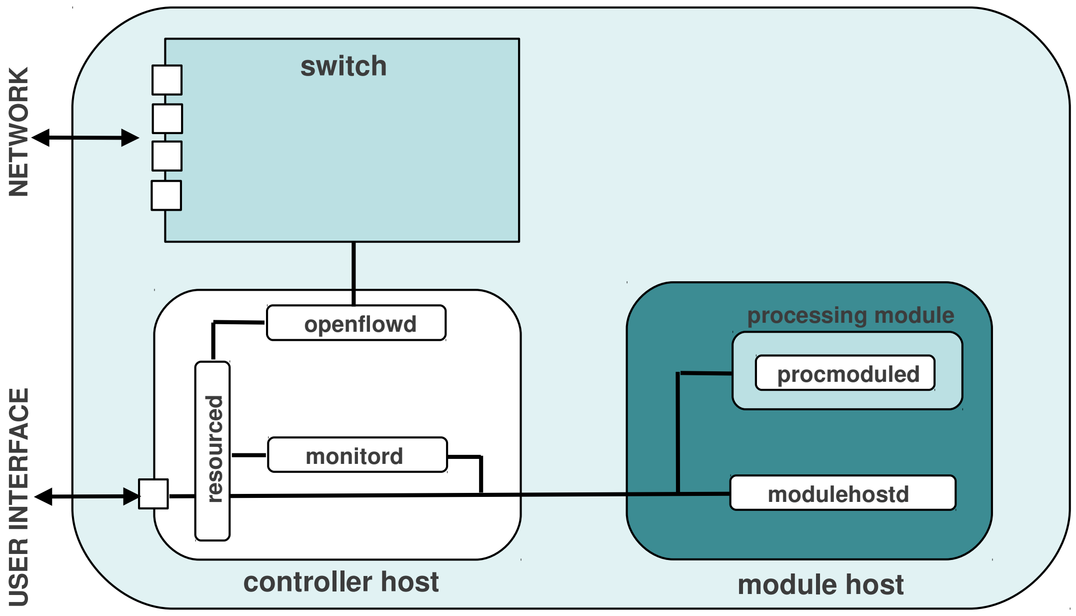


Figure 3.1: Platform controller architecture overview, only control-plane connections are shown.

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

- **Resource daemon**: This is the controller's main daemon, through which Flowstream users/clients submit *allocation requests* (refer to figure 3.1. This daemon takes care of receiving such requests and deciding whether to install them and how to allocate resources to them. This daemon contains the resource allocation algorithms described in section 3.2. 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 [20] 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 should also provide 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.

3.1.1 Daemon Architecture and Communications

At a high level, a daemon consists of a stand-alone process running an XML-RPC server. We choose 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.

A task or a Command? 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 implemented, 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.

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.

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.

Hot Patching. The controller's daemons will 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 command 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 must be reloaded.

The implementation of such a mechanism is likely to be done by providing an "upgrade" task common to all daemons that takes care of receiving files representing new tasks, writing those files to the relevant directories and reloading the classes in case these had already been loaded.

3.2 Resource Allocation

We consider the problem of exploiting of network services, e.g., firewalls, load balancers, intrusion detection (IDS), on a Flowstream platform satisfying some performance criteria. More precisely, we are given a Flowstream platform and a set of network services, each network service consists of a set of processing modules communicating with each other. The problem we tackle is to specify which server accommodates a given processing module and to specify the communication paths between elements (servers, I/O nodes) in the Flowstream platform satisfying the constraints over the capacity of links, the capacity of the servers of the Flowstream platform, and constraints over the delay of the network services.

3.2.1 Problem Description

This section is an attempt to provide an accurate (to some degree) description of the problem at hand. The problem has two main components: the Flowstream platform and the network services. The goal is to map the elements of network services to the Flowstream platform satisfying some criteria.

Flowstream platform. A Flowstream platform is built out of programmable switches (e.g., OpenFlow switches) interconnecting commodity servers which support virtualization of processing functions. All computation takes place on the servers, each of which is characterized by the number of CPU cores it contains, its memory capacity and its internal communication bandwidth. All CPU cores have the same characteristics and a shared memory model is used (i.e., all CPU cores within a server have shared access to the same memory and bandwidth).

Some of the switches are denoted as Top-of-Rack (TOR) nodes. Each server within a rack is directly connected to the corresponding TOR node. There is no direct communication between servers: all communication between servers must pass through their corresponding TOR.

A Flowstream platform is connected to a (given) number of I/O Nodes, each of which corresponds to a specific connection to the Internet with an associated bandwidth.

Every link in the Flowstream platform is assumed to be bi-directional with a given bandwidth in each direction. (In other words, its equivalent to have one separate link on each direction). In addition, there is a given delay associated with each link in the Flowstream platform. Unless otherwise specified by the input data, we will assume that both directions have the same bandwidth and delay.

Network Services. During its operation the Flowstream platform receives a number of Network Services (NS) that need to be processed. Each network service is represented by a DAG of three types of node: Source nodes, Processing Modules (PM) and Sink nodes. Each PM corresponds to a computation related to that service, while Source and Sink nodes take care of the I/O of the NS.

Each processing module requires a number of CPU cores (that can be fractional) and an amount of memory. If there is a directed link from a processing module P to a processing module P', this means that P needs to send a (given) number of packets to P'. If no link is present between two processing modules, there is no direct traffic between them.

Every NS has one or more Source nodes: all links leaving the Source nodes represent incoming traffic from the outside to the NS (there is no link in the DAG entering the Source nodes). It also has a number of Sink nodes: all links entering the Sink nodes represent outgoing traffic of the NS (there is no link of the

DAG leaving the Sink nodes). These Source and Sink nodes will be located on I/O nodes of the Flowstream platform.

Now consider any NS. Every directed link between two nodes in that NS is labeled by the (given) amount of traffic (number of packets) that needs to be sent along that link.

Network services are categorized into different NS Types. Each NS type corresponds to a network flow pattern that characterizes many real network services operating on the network. In other words, there are multiple instances of each NS type corresponding to different clients. Of course, all network services of a given type have the same characteristics. Within the scope of this project, we are not going to focus on NS types. We will simply deal with a given list of NS requests, not taking into account which type they are instances of.

Note that the number of incoming packets for an NS node is not necessarily equal to the number of outgoing packets. For example, a mirror PM may create multiple copies of some incoming packets; or a PM may decide to drop packets.

Objectives. Any solution to this problem must provide three things: determine a set of communication paths between servers, assign PMs to servers, and select I/O nodes for each sink node. This must be done while satisfying a number of constraints presented in later sections.

Note that I/O nodes can only be used for communication with the internet, and cannot be used for internal communication inside the Flowstream platform.

Assignment of Network Services to Servers For any network service NS given to the system, the solution must assign each processing module of that service to a server of the Flowstream platform. It is possible to allocate PMs of the same NS to servers of different racks. The granularity is at the server level: we can only choose the server that accommodates a PM, not the actual core in that server.

Selection of I/O Nodes for Sink Nodes As mentioned earlier, for each Sink node, there is a set R of I/O nodes on which the Sink node can be located. A solution must select which I/O node in R will be assigned to each Sink node.

Constraints. Before listing the constraints, we first need to describe the process by which NS are handled. Consider two elements (e.g., Source node, PMs, Sink node) within the same NS, which have a directed link in the DAG: $P \rightarrow P'$. Let w be the traffic that has to be sent from P to P', and let S and S' be servers to which they are respectively assigned to. Also, let $h(S, S')$ be the path between these servers. Then, an amount of traffic equal to w needs to be reserved on each link along $h(S, S')$.

For PMs running on the same server, we only consider the (given) delay for sending data from one PM to the other. We do not enter into the details of how this communication is performed (for instance, we do not consider any monitoring process between pairs of PMs within the same server).

Now we are ready to state the problem constraints:

(i) The total number of CPU cores used by PMs of any NS in any particular server must be smaller or equal to that servers number of available CPU cores

(ii) The total amount of memory required by PMs of any NS in any particular server must be smaller or equal to that servers memory capacity

(iii) The total amount of communication between linked pairs of PMs on the same server cannot exceed the servers total internal communication bandwidth

(iv) On any link of the network, the total amount of traffic reserved must not exceed the links bandwidth in that direction (This includes server-TOR links)

(v) The total weighted delay on any specified path on a PMs DAG must respect its given upper bound.

Network metrics. We finally describe a number of metrics that can be used to evaluate any solution to this problem, and some possible objectives for the solution.

Service Delay For each directed link in a network service DAG, we define its delay as the sum of delays along the path in the Flowstream platform graph that connects its end-points, with respect to an assignment of PMs to servers. If the two end-points of the link are on the same server, then the links delay is equal to a given value associated with the server (this value is an input to the problem).

Note that processing time is not taken into account in the delay computation. The reasoning is that, since all processors have the same characteristics, and since we need to perform that processing anyway, processing time can be considered a constant in the system (and therefore there is nothing to optimize there).

The delay of a given NS is defined as the longest weighted path in its DAG, in which links are labeled by their corresponding delay. Note that delay does not depend on packet sizes.

Approximate Service Delay In order to avoid computational issues, a metric to consider is the approximate service delay, defined as the sum of delays over all the directed links in its DAG. Clearly, this is an upper bound on the actual service delay.

Typical Service Delay An alternative to avoid computational issue is to have a specified service delay, such as a particular path in the DAG, which is representative of the service delay. A typical service delay would then be part of the specified network service, and could involve a particular path, a sum or a max function.

Total Delay This is defined as the sum of service delay over all NS in the system. Standard, approximate or typical service delay, may be used for this. In the last two cases, we may speak of Total Approximate Delay or Total Typical Delay. Once we settle to a particular type of service delay definition, we can simply speak of Total Delay (with respect to the chosen type of delay definition.)

Maximum Bandwidth Usage Given a path selection and an assignment of PMs to servers, each link in the Flowstream platform has a fraction of its bandwidth reserved. We define maximum bandwidth usage as the maximum fraction over all links in the network.

Average Bandwidth Similarly, we can define the average bandwidth as the average fraction of bandwidth reserved.

Objectives. Finally, here is an enumeration of possible objectives for a solution

Minimize Maximum Bandwidth Usage One first approach is to try to minimize the maximum bandwidth utilization. This is not very desirable, because it tends to focus on a few bad nodes, rather than the whole network.

Minimize delay Another possible objective, which seems to be the most desirable from the network operators point of view, is to try to minimize delay, while satisfying bandwidth requirements.

Satisfaction Problem Instead of optimizing a metric, we could alternatively cast the problem as a satisfaction problem. The idea is to select paths, assign PMs to servers and choose I/O nodes, while respecting all the constraints. In fact, this is rather a family of problems, since it will depend on the selection of delay bounds for NS paths of interest.

3.2.2 Problem Model

For a clearer understanding of the solution approach, we abstract the above-mentioned problem description, so that the problem can be viewed as a graph-embedding problem, in which the nodes of a number of DAGs (the network services) are mapped to the nodes of a large directed graph (the Flowstream platform), under some constraints.

Flowstream platform. Under this view, the Flowstream platform is a directed graph, $\mathcal{D} = (F, E)$, whose nodes F are either different types of switches of the Flowstream platform (I/O nodes, OpenFlow switches) or servers. We denote:

- $F_{\text{I/O}}$ the set of I/O nodes of $\mathcal{D}$ ($F_{\text{I/O}} \subset F$).
- F_{SERVER} the set of servers of $\mathcal{D}$ ($F_{\text{SERVER}} \subset F$).

Each node $f_i \in F$ has a CPU and memory capacity, C_i and M_i, respectively. These capacities are non-zero only for servers: this is because all computation takes place at the server level, and therefore, for our purposes, it is not important to consider the internal memory and CPU power of the switches, since these are not used in any computation (they are only used for packet forwarding).

Each link $e_i \in E$ is associated with two attributes: its bandwidth B_i and its delay D_i.

Note that, the above model can be used to handle in a uniform way the different kinds of bandwidth of the problem description, such as internal server bandwidth, server-to-TOR bandwidth, inter-switch bandwidth.

Network Services. In an analogous fashion with the Flowstream platform graph, we can represent the set of all network services as a single directed graph $\mathcal{S} = (G, H)$, each connected component of which will be the DAG describing a different network service. As mentioned in earlier sections, each node in a network service node DAG can be any of three types:

Source: located at a node in $F_{\text{I/O}} \in \mathcal{D}$. It receives data from outside.

PM: Processing module. It processes data on a node $F_{\text{SERVER}} \in \mathcal{D}$

Sink: located at a node in $F_{\text{I/O}} \in \mathcal{D}$. It sends data to outside.

We similarly partition the nodes of $\mathcal{S}$ according to their type:

$$G = G_{\text{SOURCE}} \cup G_{\text{PM}} \cup G_{\text{SINK}}$$

With each type of node, we associate a different piece of data.

- For each $g_i \in G_{\text{PM}}$, corresponding to a processing module, there is a corresponding CPU requirement, c_i, and memory requirement, m_i.
- For each $g_i \in G_{\text{SOURCE}}$ there is a single (given) node $U_i \in F_{\text{I/O}}$, on which g_i is located.
- Finally, for each $g_i \in G_{\text{SINK}}$ there is a (given) subset of nodes $V_i \subseteq F_{\text{I/O}}$ on which g_i can be located.

With respect to the links in H, there are only three requirements:

- They form a directed acyclic graph
- There is no link of the form (r, h), $h \in G_{\text{SOURCE}}$
- There is no link of the form (h, r), $h \in G_{\text{SINK}}$

Each network service link $h_i \in H$ is characterized by its traffic requirement, b_i, which specifies how much bandwidth it requires from each Flowstream platform link that handles it.

Objective. Each network service corresponds to a tree of computations, in which data is received from some I/O nodes of the Flowstream platform, it is processed on one or more of its servers and the results are sent to some of the I/O nodes, in order to be forwarded to the outside world.

Therefore, the main objective is to assign a proper Flowstream platform node to each network service node. Since not all network service nodes are assigned to the same node, another objective is to decide which directed paths to use, in order to communicate data between Flowstream platform nodes. We now describe these two objectives in little more detail.

Network Service Node Mapping.

This can be seen as a function $\mathtt{s} : G \rightarrow F$, which assigns to each network service node $g_i \in G$ a Flowstream platform node $\mathtt{s}(g_i)$, under the following constraints:

- $g_i \in G_{\text{SOURCE}} \;\Rightarrow\; \mathtt{s}(g_i) = U_i$
- $g_i \in G_{\text{PM}} \;\Rightarrow\; \mathtt{s}(g_i) \in F_{\text{SERVER}}$
- $g_i \in G_{\text{SINK}} \;\Rightarrow\; \mathtt{s}(g_i) \in V_i$

Flowstream platform Path Selection.

This consists in computing a directed path between any two servers of the Flowstream platform and between any server and I/O node. More formally, let $\mathcal{P}$ the set of all directed paths over the Flowstream platform graph $\mathcal{D}$. A directed path is defined a sequence of directed edges, so that, for any two consecutive edges in the sequence, the destination of the first edge is the source of the second edge. Also, let

$$F_p = (F_{\text{SERVER}} \times F_{\text{SERVER}}) \cup (F_{\text{I/O}} \times F_{\text{SERVER}}) \cup (F_{\text{SERVER}} \times F_{\text{I/O}})$$

Then, we are looking for a function

$$\mathtt{path} : F_p \rightarrow \mathcal{P}$$

that assigns to each pair of nodes $(f_i, f_j) \in F_p$ a directed path, $\mathtt{path}(f_i, f_j) \in \mathcal{P}$.

Constraints. In order to better describe the problem's constraints, we introduce some short-hand notation

- $\mathtt{USING}(e \in E) = \{(g, g') \in H : e \in \mathtt{path}(\mathtt{s}(g), \mathtt{s}(g'))\}$
- $\mathtt{DELAY}(h \in H) = \sum_{e_i \in E : h \in \mathtt{USING}(e_i)} D_i$

The problem can then be stated as a satisfaction problem, in which one must compute a network service node assignment and a Flowstream platform path selection so that the following constraints are satisfied.

- CPU and Memory Constraints:

$$\forall f_j \in F_{\text{SERVER}} : \sum_{g_i \in G : \mathtt{s}(g_i) = f_j} c_i \le C_j$$

$$\forall f_j \in F_{\text{SERVER}} : \sum_{g_i \in G : \mathtt{s}(g_i) = f_j} m_i \le M_j$$

- Bandwidth Constraints:

$$\forall e_j \in E : \sum_{h_i \in \mathtt{USING}(e_j)} b_i \le B_j$$

- Delay-Constrained Network Service Paths: We are given a set of paths $\mathcal{T}$ over the network service graph, each of which starts from a node in G_{SOURCE} and ends at a node in G_{SINK}. Each $t_i \in \mathcal{T}$ has a delay bound z_i. The constraint states that:

$$\forall t_i \in \mathcal{T} : \sum_{h \in t_i} \texttt{DELAY}(h) \leq z_i$$

Here is the rationale behind the delay constraints. For each network service, we would like to limit the total delay between receiving data from the I/O nodes and sending back the result of their processing. This delay depends both on the mapping of network service nodes to nodes of the Flowstream platform and the Flowstream platform path selection.

In order to have a manageable model, each network service is represented by a number of paths (rather than *all* the paths) from a Source to a Sink node, along with a bound on the total delay once this path is embedded into the Flowstream platform graph.

3.2.3 Local Search algorithm

For ease the presentation of the algorithms, without loss of generality, we suppose that TOR are indexed by $1, \ldots, nbTor$, each TOR has k servers attached to it. The set of servers attached to the TOR i are indexed by $(i-1) * k + 1, \ldots, i * k$. The number of servers of the Flowstream is: $nbServers = k * nbTor$. Each server j has a CPU capacity C_j and memory capacity M_j. The local search algorithm typically starting from an initial solution and iteratively move from a current solution to one of its neighbors. A move is represented by a pair $\langle g_i, j \rangle$ in which the PM g_i is assigned to the server j.

Initial solution. We describe in this section a greedy round robin algorithm for establish an initial solution. The goal is to locate each PM on a server such that the sum of violations of all constraints is as small as possible. The sources and sinks of network services are placed randomly on I/O nodes of the Flowstream. Naturally, the PMs of each network service should be placed on a server or on some servers connected to a same TOR. This can guarantee much the delay constraints. The main idea of the greedy algorithm is to consider at each step all PMs of a network service (i.e., a connected component) of $\mathcal{S}$ and try first to place these PMs on a server or some servers attaching to the same TOR satisfying the constraints on CPU and Memory. If this trial is failed, we try other servers attached to different TOR.

The algorithm maintains a pointer p representing the current selected server. For each considered PM g_i, we try to find a first server from the position p for PM i in a round robin way (i.e., iteratively try a server j with the order $p, p+1, ..., nbServers, 1, 2, ..., p-1$) such that the assignment does not violation any the constraints. If no such server is found, the algorithm selects a server j such that the sum of violations of all constraints is minimal when the PM g_i is assigned to server j. Then the PM g_i is assigned to the server j and p is assigned to j.

Note that this greedy round robin search is good for small instances (small data centers and small number of network services). But with large data center, for instance, with 10000 servers and 12000 network services, the execution time is substantially increased. To avoid this situation, we do the following improvement. We perform a greedy heuristic that avoid scanning all servers for each considered processing module. We partition the set of servers of the data center into cluster, for instance, each cluster is a set of all servers of the same rack. We perform the round robin search in two levels: the rack level and the servers (within a rack) level. We consider each network service at each iteration. For each processing module of the considered network service, we perform a round robin search within current rack. If the algorithm fails to find a suitable server, then we consider a next rack as the current rack for the next network service. This improved strategy is applied in the following experiments section.

This greedy round robin algorithm performs better than the simple greedy algorithm that considers PMs randomly and for each PM and selects the best server for each PM at each step in term of both computation time and solution quality. This has been shown in our preliminary results.

Tabu search. The initial solution computed by the greedy round robin algorithm has a number of violations of the constraints. The tabu search presented in this section aims at improving the initial solution.

The tabu search is given in Algorithm 1 in which s denote the current solution i.e., the assignment of PMs to servers. We denote $s[g_i, j]$ the solution created from s by reassigning the PM g_i to the server j. We denote $v(s)$ the number of violations of all the constraints of the problem. s^* denote the best solution found so far (line 1). At each step of the search, we select a PM g_i and a server j for the assignment. There four constraints of the problem which are grouped into two kinds:

- The first kind is the combination of the capacity constraints on CPU, Memory of servers, and on Bandwidth of links. We denote CMB this combined constraint. The violations of the CMB constraint is the sum of the violations of the constraints on CPU, Memory, and Bandwidth.
- The second kind is the delay constraint on network services.

There are two situations:

- The CMB constraint is violated (i.e., the violations of the CMB constraint is greater than 0), we perform the moves that reduce the total violations by changing the PMs that contribute to its violations (lines 4-13). Precisely, we first select the PM g_i that contributes much to the violations of the CMB constraint and then a server j such that $v(s[g_i, j])$ is minimal (lines 4-5). This multistage selection aims at reducing the computation overhead. If g_i is not tabu or the $s[g_i, j]$ is better than s^* then we take it (lines 7-8). Otherwise, we select the PM that is *not tabu* and that contributes much to the violations of the CMB constraint. We then select the server j such that $v(s[g_i, j])$ is minimal and perform the reassignment (lines 10-13).
- Otherwise, we perform the moves that reduce the violations by changing the PMs that contribute to the violations of the delay constraint (see lines 15-19). We first select randomly one of three bounded

paths having most violations of the delay constraint (line 15). Variable *Cand* (line 16) is the set of PMs of the selected path p. Line 17 computes a set $\mathcal{M}$ of acceptable moves which is a set of pairs of a PM g_i and a server j such that g_i is not tabu or $s[g_i, j]$ is better than s^*. Line 18 selects the best move and lines 19-20 perform the move.

Algorithm 1: TabuSearch()

Input: A Flowstream $\mathcal{D}$ and a set of network services $\mathcal{S}$ and the initial solution s
Output: A solution s^* that is better than s

```
1  $s^* \leftarrow s$;
2  while timeout is not reached and $v(s^*) > 0$ do
3      if sum of violations of constraints on CPU, Memory, and Bandwidth > 0 then
4          $g_i \leftarrow$ select the PM that contributes much to the violation of the CMB constraints;
5          $j \leftarrow$ select the server such that $v(s[g_i, j])$ is minimal;
6          if $g_i$ is not tabu or $v(s[g_i, j]) < v(s^*)$ then
7              $s \leftarrow s[g_i, j]$;
8              make tabu $g_i$;
9          else
10             $g_i \leftarrow$ select the not-tabu PM that contributes much to the violation of the CMB constraints;
11             $j \leftarrow$ select the server such that $v(s[g_i, j])$ is minimal;
12             $s \leftarrow s[g_i, j]$;
13             make tabu $g_i$;
14     else
15         $p \leftarrow$ select randomly one of three bounded paths that have most number of violations of the delay
           constraint;
16         $Cand \leftarrow$ set of PM of $p$;
17         $\mathcal{M} \leftarrow$ set of $\langle g_i, j\rangle$ ($g_i \in Cand$ and $j \in F_{SERVER}$) such that such that $g_i$ is not tabu or
           $v(s[g_i, j]) < v(s^*)$;
18         $\langle g_i, j\rangle \leftarrow \text{argMin}_{\langle g'_i, j'\rangle \in \mathcal{M}}\ v(s[g'_i, j'])$;
19         $s \leftarrow s[g_i, j]$;
20         make tabu $g_i$;
21     if $v(s) < v(s^*)$ then
22         $s^* \leftarrow s$;
23 return $s^*$;
```

3.2.4 Experiments

We perform a simulation test on two Flowstreams: the first one is small with only one switch and 40 servers and the second one is large data center.

3.2.4.1 A Basic Flowstream Platform

We first test the model on a basic Flowstream platform with one switch and 40 servers. We need to adapt this platform in order to be able to apply the generic problem model described above: The platform now has one I/O node and one ToR switch to which 40 servers are attached. Internal links of servers are assumed to be 10Gbps. Each link connecting a server to the ToR switch has bandwidth of 1Gbps. Each server has 8 cores and 2048MB of memory. The delay constraint is not considered in this case because it does not make sense for this network topology, where all delays are the same.

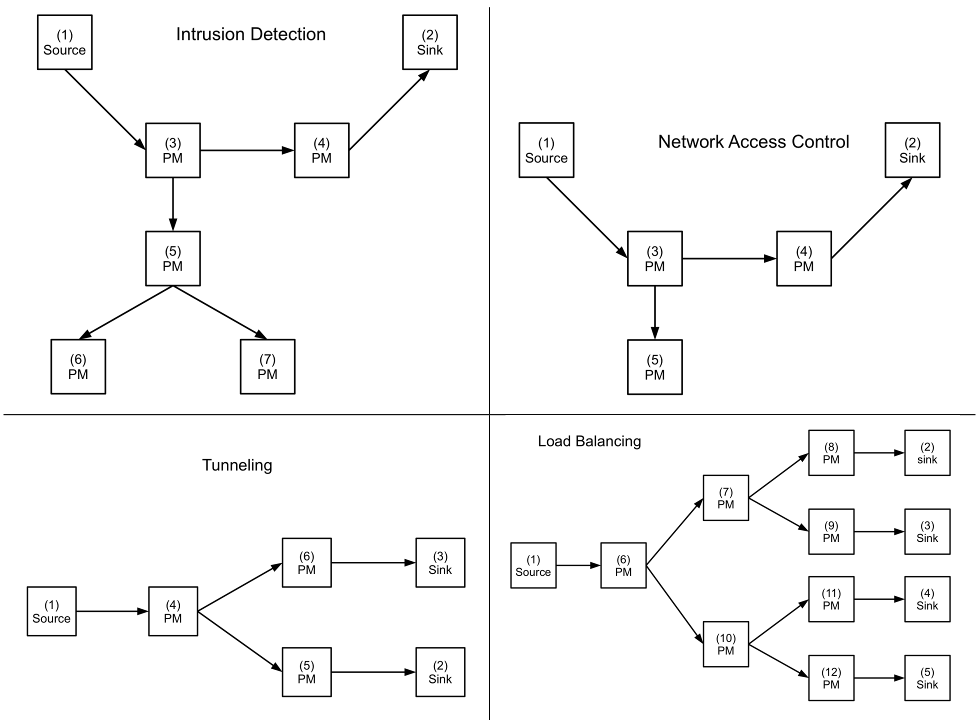


Figure 3.2: Network service templates

Name	$\sharp$IO	$\sharp$PM	cpu	memory	traffic (pps)
intrusion detection	2	5	$\{0.5, 2\}$	100..1024	38800..100000
load balancing	5	7	$\{0.7, 1, 1.5\}$	100..200	25000..100000
network access control	2	3	$\{0.5, 1\}$	100..200	96000..100000
tunnelling	3	3	$\{1.5, 2.5\}$	200..1024	50000..100000

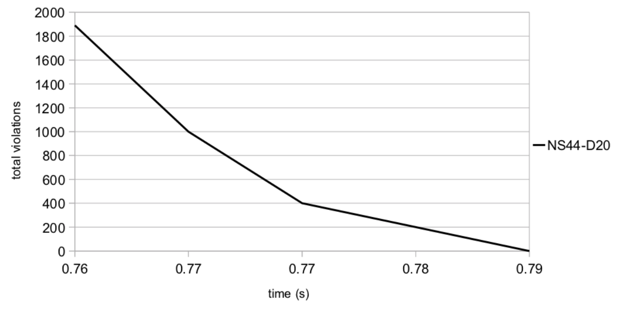
Table 3.1: Description of network service samples

We take four network service examples including intrusion detection, load balancing, network access control, and tunneling which are described in Table 3.1 (see the topology in Figure 3.3). The size of each packet is assumed to be 1024B. From these network service samples, we generate $k \in \{10, 11, 12, 13\}$ instances for each type. In total, we have $\{40, 44, 48, 52\}$ network services.

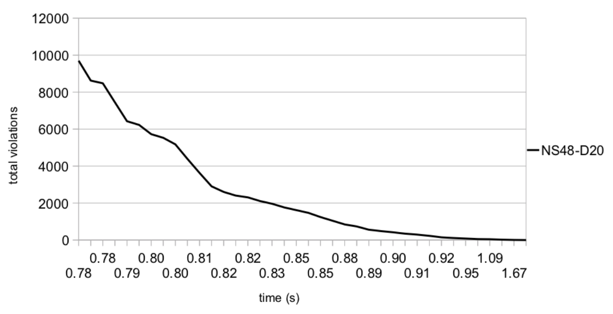
Experimental results. For instance with 40 network services, the greedy round robin finds a feasible solution (i.e., solution that satisfies all the constraints) in about 0.7 seconds. The tabu search is thus not necessary. For instances with 44, 48, the greedy round robin cannot find feasible solutions, but the tabu search respectively finds feasible solutions in 0.79s and 1.68s. The algorithms cannot find feasible solution for instances with 52 network services. In this case, there are too many network services, so the capacity of CPU and memory of servers cannot satisfy all these network services. Figure 3.3 shows the evolution of the best solutions found in term of the total violations of all the constraints of the problem.

3.2.4.2 Large Flowstream

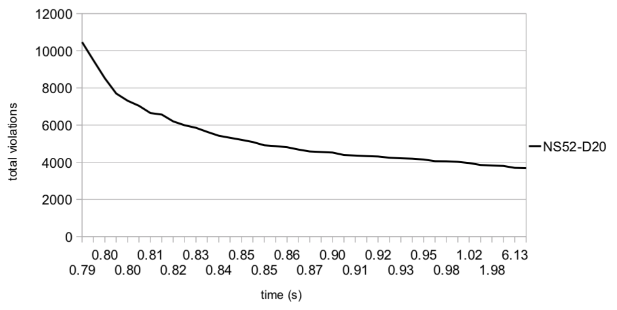
We consider a Flowstream as a large data center for testing the local search algorithm including 32 core switches, 32 intermediate switches, 500 ToR switches and 10000 servers. Each core switch is connected to 5 I/O switches. Each link connecting a server to a ToR has bandwidth of 1Gbps. The internal bandwidth of



44 network services

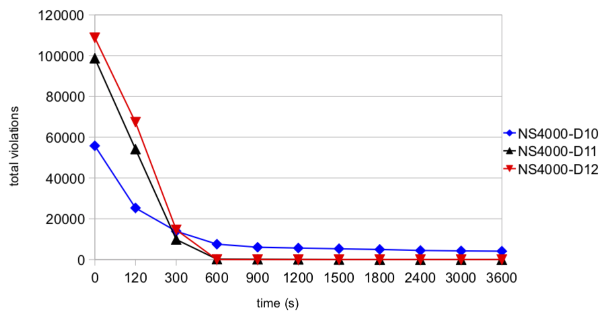


48 network services

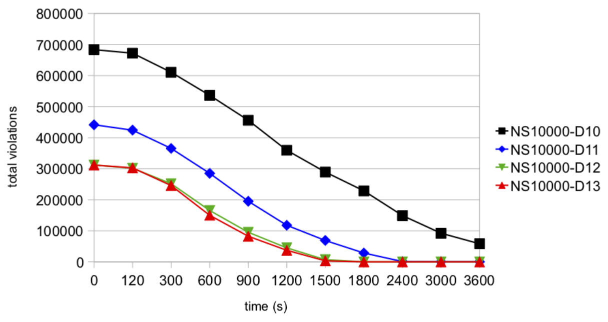


52 network services

Figure 3.3: Solution evolution for a basic CHANGE platform



a. 4000 network services



b. 10000 network services

Figure 3.4: Solution evolution for a large platform

each server is assumed to be 10Gbps. Each remaining link of the data center has the bandwidth of 10Gbps. The delay on each link is assumed to be 1. Each server has 8 cores and 2048MB of memory.

We take 4 network service samples including intrusion detection, load balancing, network access control, and tunneling which are described in Table 3.1 (see the topology in Figure 3.3). The size of each packet is assumed to be 1024B. From these network service samples, we generate $k \in \{1000, 1500, 2000, 2500, 3000\}$ instances for each type. In total, we have $\{4000, 6000, 8000, 10000, 12000\}$ network services. We try different values in $\{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$ for the delay bound.

Experimental results. Experimental results are presented in Tables 3.2, 3.3, 3.4, 3.5, 3.6 with the following structure. Column 1 presents the name of instances under the format NSx-Dy where x is the number of network services and y is the value of delay bound. Column 2 is the number of processing modules to be accommodated on servers. Columns 3-7 respectively present the number of violations of the combined constraint: constraints on cpu, memory, bandwidth, delay of the best solution found. Column 8 is the execution time. For each instance, we report the initial solution found by the greedy round robin search, and the best solution found by the tabu search with different time limits i.e., 120(s), 600(s), 3600(s). We find that, for every instance, the initial solutions found by the greedy round robin do not satisfy all the constraints. The algorithm cannot find feasible solutions in cases the delay bound is 10. These initial solution are then improved by the tabu search and The total violations improves overtime (see Figure 3.4). For instances with 4000 network services (see Table 3.2), and the delay bound is greater than 12, the tabu search can find feasible solutions in 600 seconds. For instances with 6000 network services (see Table 3.3), and the delay bound is greater than 12, the tabu search can find feasible solution in less than 1000 seconds. For instances with 8000 and 10000 network services (see Tables 3.4,3.5), when the delay bound is greater than 12, the tabu search can find feasible solution in less than 30 minutes. Finally, the tabu search cannot find feasible solutions for instances with 12000 network services (see Table 3.6). In this case, all the constraints (on cpu, memory, bandwidth, and delay) are violated.

3.2.5 Summary and Future work

In this report, we have specified the problem of locating processing modules of network services on Flowstream platform satisfying some performance criteria e.g., capacity constraints on links and servers and the constraint on the delay of network services. The model has been implemented in the Comet programming language.

The ongoing work in the short term consists of generating pertinent data for testing the model. In the long term, we will specify and solve the online version of the problem where network services are instantiated online.

Instance	♯PM	total	cpu	mem	bandwidth	delay	time (s)	search
NS4000-D10	18000	55788	16340	34122	0	5326	30.377	initial solution
NS4000-D10	18000	25304	14350	5616	0	5338	119.879	120s
NS4000-D10	18000	7552	1760	0	0	5792	599.737	600s
NS4000-D10	18000	4096	0	0	0	4096	3593.39	3600s
NS4000-D11	18000	98693	6670	91722	0	301	27.905	initial solution
NS4000-D11	18000	54151	3800	50046	0	305	119.719	120s
NS4000-D11	18000	197	0	0	0	197	545.514	600s
NS4000-D11	18000	0	0	0	0	0	1612.26	3600s
NS4000-D12	18000	108779	8540	100166	0	73	28.765	initial solution
NS4000-D12	18000	67377	4840	62464	0	73	119.235	120s
NS4000-D12	18000	17	0	0	0	17	589.976	600s
NS4000-D12	18000	0	0	0	0	0	798.077	3600s
NS4000-D13	18000	108857	8640	100166	0	51	31.797	initial solution
NS4000-D13	18000	77737	4990	72690	0	57	119.951	120s
NS4000-D13	18000	0	0	0	0	0	595.373	600s
NS4000-D13	18000	0	0	0	0	0	595.373	3600s
NS4000-D14	18000	108842	8640	100166	0	36	25.609	initial solution
NS4000-D14	18000	69996	4620	65340	0	36	119.599	120s
NS4000-D14	18000	0	0	0	0	0	515.3	600s
NS4000-D14	18000	0	0	0	0	0	515.3	3600s
NS4000-D15	18000	109700	8910	100790	0	0	32.038	initial solution
NS4000-D15	18000	74308	4720	69588	0	0	119.907	120s
NS4000-D15	18000	0	0	0	0	0	492.83	600s
NS4000-D15	18000	0	0	0	0	0	492.83	3600s
NS4000-D16	18000	109700	8910	100790	0	0	34.63	initial solution
NS4000-D16	18000	71480	4740	66740	0	0	118.391	120s
NS4000-D16	18000	0	0	0	0	0	456.124	600s
NS4000-D16	18000	0	0	0	0	0	456.124	3600s
NS4000-D17	18000	109700	8910	100790	0	0	34.962	initial solution
NS4000-D17	18000	76994	5080	71914	0	0	119.619	120s
NS4000-D17	18000	0	0	0	0	0	465.237	600s
NS4000-D17	18000	0	0	0	0	0	465.237	3600s
NS4000-D18	18000	109700	8910	100790	0	0	35.758	initial solution
NS4000-D18	18000	76714	4550	72164	0	0	119.311	120s
NS4000-D18	18000	0	0	0	0	0	461.492	600s
NS4000-D18	18000	0	0	0	0	0	461.492	3600s
NS4000-D19	18000	109700	8910	100790	0	0	35.818	initial solution
NS4000-D19	18000	84010	5020	78990	0	0	119.559	120s
NS4000-D19	18000	0	0	0	0	0	500.371	600s
NS4000-D19	18000	0	0	0	0	0	500.371	3600s
NS4000-D20	18000	109700	8910	100790	0	0	29.641	initial solution
NS4000-D20	18000	76844	4980	71864	0	0	119.835	120s
NS4000-D20	18000	0	0	0	0	0	465.817	600s
NS4000-D20	18000	0	0	0	0	0	465.817	3600s

Table 3.2: Experimental results with 4000 network services

Instance	♯PM	total	cpu	mem	bandwidth	delay	time (s)	search
NS6000-D10	27000	153691	36890	107826	0	8975	50.563	initial solution
NS6000-D10	27000	131289	33440	88876	0	8973	119.415	120s
NS6000-D10	27000	41411	24180	8298	0	8933	598.993	600s
NS6000-D10	27000	8300	0	0	0	8300	3597.55	3600s
NS6000-D11	27000	146245	8600	137340	0	305	48.419	initial solution
NS6000-D11	27000	121262	6690	114264	0	308	119.423	120s
NS6000-D11	27000	3527	1140	2060	0	327	599.701	600s
NS6000-D11	27000	0	0	0	0	0	1836.55	3600s
NS6000-D12	27000	181110	10770	170248	0	92	49.955	initial solution
NS6000-D12	27000	158040	8750	149198	0	92	119.547	120s
NS6000-D12	27000	20880	1380	19408	0	92	599.921	600s
NS6000-D12	27000	0	0	0	0	0	1231.27	3600s
NS6000-D13	27000	180481	10890	169524	0	67	53.031	initial solution
NS6000-D13	27000	157061	8970	148024	0	67	118.991	120s
NS6000-D13	27000	22331	1850	20404	0	77	599.321	600s
NS6000-D13	27000	0	0	0	0	0	906.168	3600s
NS6000-D14	27000	180463	10890	169524	0	49	54.123	initial solution
NS6000-D14	27000	159543	9270	150224	0	49	119.599	120s
NS6000-D14	27000	25667	1980	23634	0	53	599.989	600s
NS6000-D14	27000	0	0	0	0	0	891.783	3600s
NS6000-D15	27000	184436	10210	174224	0	2	48.707	initial solution
NS6000-D15	27000	159206	8180	151024	0	2	119.627	120s
NS6000-D15	27000	22014	1680	20330	0	4	598.913	600s
NS6000-D15	27000	0	0	0	0	0	832.452	3600s
NS6000-D16	27000	184434	10210	174224	0	0	46.846	initial solution
NS6000-D16	27000	160864	7840	153024	0	0	119.787	120s
NS6000-D16	27000	25184	1700	23484	0	0	599.121	600s
NS6000-D16	27000	0	0	0	0	0	832.052	3600s
NS6000-D17	27000	184434	10210	174224	0	0	45.93	initial solution
NS6000-D17	27000	157824	7900	149924	0	0	119.975	120s
NS6000-D17	27000	21678	1470	20208	0	0	599.721	600s
NS6000-D17	27000	0	0	0	0	0	788.465	3600s
NS6000-D18	27000	184434	10210	174224	0	0	53.887	initial solution
NS6000-D18	27000	158018	7670	150348	0	0	119.195	120s
NS6000-D18	27000	23402	1850	21552	0	0	599.277	600s
NS6000-D18	27000	0	0	0	0	0	806.89	3600s
NS6000-D19	27000	184434	10210	174224	0	0	51.799	initial solution
NS6000-D19	27000	160404	8230	152174	0	0	119.579	120s
NS6000-D19	27000	23810	1450	22360	0	0	599.909	600s
NS6000-D19	27000	0	0	0	0	0	806.638	3600s
NS6000-D20	27000	184434	10210	174224	0	0	53.287	initial solution
NS6000-D20	27000	162754	8030	154724	0	0	119.431	120s
NS6000-D20	27000	23368	1510	21858	0	0	599.317	600s
NS6000-D20	27000	0	0	0	0	0	810.462	3600s

Table 3.3: Experimental results with 6000 network services

Instance	♯PM	total	cpu	mem	bandwidth	delay	time (s)	search
NS8000-D10	36000	311455	70680	228116	0	12659	65.448	initial solution
NS8000-D10	36000	289791	66290	210852	0	12649	118.679	120s
NS8000-D10	36000	141748	55550	73570	0	12628	599.569	600s
NS8000-D10	36000	13332	0	0	0	13332	3599.67	3600s
NS8000-D11	36000	199552	13120	185970	0	462	58.179	initial solution
NS8000-D11	36000	176302	11570	164270	0	462	118.731	120s
NS8000-D11	36000	41209	3720	37008	0	481	599.425	600s
NS8000-D11	36000	7	0	0	0	7	3346.36	3600s
NS8000-D12	36000	237548	14780	222716	0	52	59.883	initial solution
NS8000-D12	36000	214268	12850	201366	0	52	119.963	120s
NS8000-D12	36000	69022	3260	65710	0	52	599.865	600s
NS8000-D12	36000	0	0	0	0	0	1475.04	3600s
NS8000-D13	36000	237888	15020	222840	0	28	64.632	initial solution
NS8000-D13	36000	223108	13590	209490	0	28	118.823	120s
NS8000-D13	36000	80352	3560	76762	0	30	599.557	600s
NS8000-D13	36000	0	0	0	0	0	1350.95	3600s
NS8000-D14	36000	238473	15020	223440	0	13	65.1	initial solution
NS8000-D14	36000	214337	13210	201114	0	13	119.619	120s
NS8000-D14	36000	69739	3640	66086	0	13	599.273	600s
NS8000-D14	36000	0	0	0	0	0	1135.08	3600s
NS8000-D15	36000	237515	14970	222540	0	5	61.723	initial solution
NS8000-D15	36000	216079	13210	202864	0	5	119.823	120s
NS8000-D15	36000	69251	3860	65384	0	7	599.489	600s
NS8000-D15	36000	0	0	0	0	0	1135.69	3600s
NS8000-D16	36000	237360	14920	222440	0	0	69.108	initial solution
NS8000-D16	36000	218190	13200	204990	0	0	119.843	120s
NS8000-D16	36000	76104	3740	72364	0	0	598.505	600s
NS8000-D16	36000	0	0	0	0	0	1137.46	3600s
NS8000-D17	36000	237360	14920	222440	0	0	69.156	initial solution
NS8000-D17	36000	224290	13350	210940	0	0	119.555	120s
NS8000-D17	36000	76710	3750	72960	0	0	599.841	600s
NS8000-D17	36000	0	0	0	0	0	1160.58	3600s
NS8000-D18	36000	237360	14920	222440	0	0	70.112	initial solution
NS8000-D18	36000	222604	13190	209414	0	0	119.899	120s
NS8000-D18	36000	70846	3560	67286	0	0	599.181	600s
NS8000-D18	36000	0	0	0	0	0	1150.55	3600s
NS8000-D19	36000	237360	14920	222440	0	0	69.272	initial solution
NS8000-D19	36000	219044	13330	205714	0	0	119.675	120s
NS8000-D19	36000	64218	3660	60558	0	0	599.713	600s
NS8000-D19	36000	0	0	0	0	0	1105.8	3600s
NS8000-D20	36000	237360	14920	222440	0	0	68.92	initial solution
NS8000-D20	36000	223930	13390	210540	0	0	119.707	120s
NS8000-D20	36000	80150	3790	76360	0	0	599.857	600s
NS8000-D20	36000	0	0	0	0	0	1155.98	3600s

Table 3.4: Experimental results with 8000 network services

Instance	♯PM	total	cpu	mem	bandwidth	delay	time (s)	search
NS10000-D10	45000	683660	132730	533802	0	17128	87.665	initial solution
NS10000-D10	45000	672139	130190	524834	0	17115	119.367	120s
NS10000-D10	45000	536373	101500	417932	0	16941	599.537	600s
NS10000-D10	45000	58581	32520	10066	0	15995	3598.89	3600s
NS10000-D11	45000	441466	34840	405716	0	910	80.233	initial solution
NS10000-D11	45000	424326	31640	391776	0	910	119.951	120s
NS10000-D11	45000	285205	17410	266852	0	943	599.425	600s
NS10000-D11	45000	497	0	0	0	497	3568.19	3600s
NS10000-D12	45000	311769	20470	291224	0	75	84.165	initial solution
NS10000-D12	45000	301555	17750	283724	0	81	119.511	120s
NS10000-D12	45000	165463	9460	155922	0	81	599.197	600s
NS10000-D12	45000	0	0	0	0	0	2087.29	3600s
NS10000-D13	45000	311948	20590	291324	0	34	83.805	initial solution
NS10000-D13	45000	302888	18030	284824	0	34	119.619	120s
NS10000-D13	45000	149980	8550	141396	0	34	599.189	600s
NS10000-D13	45000	0	0	0	0	0	1768.03	3600s
NS10000-D14	45000	311628	20570	291024	0	34	84.469	initial solution
NS10000-D14	45000	302462	18080	284348	0	34	119.667	120s
NS10000-D14	45000	165214	9230	155946	0	38	599.225	600s
NS10000-D14	45000	0	0	0	0	0	1666.04	3600s
NS10000-D15	45000	312240	20610	291624	0	6	122.451	initial solution
NS10000-D15	45000	312240	20610	291624	0	6	122.451	120s
NS10000-D15	45000	173326	10350	162970	0	6	599.529	600s
NS10000-D15	45000	0	0	0	0	0	1733.92	3600s
NS10000-D16	45000	311310	20810	290500	0	0	127.143	initial solution
NS10000-D16	45000	311310	20810	290500	0	0	127.143	120s
NS10000-D16	45000	170124	10350	159774	0	0	599.105	600s
NS10000-D16	45000	0	0	0	0	0	1701.01	3600s
NS10000-D17	45000	311310	20810	290500	0	0	125.663	initial solution
NS10000-D17	45000	311310	20810	290500	0	0	125.663	120s
NS10000-D17	45000	175190	9970	165220	0	0	599.837	600s
NS10000-D17	45000	0	0	0	0	0	1660.12	3600s
NS10000-D18	45000	311310	20810	290500	0	0	85.665	initial solution
NS10000-D18	45000	300190	17790	282400	0	0	119.275	120s
NS10000-D18	45000	154606	7960	146646	0	0	598.085	600s
NS10000-D18	45000	0	0	0	0	0	1564.42	3600s
NS10000-D19	45000	311310	20810	290500	0	0	80.825	initial solution
NS10000-D19	45000	297720	17370	280350	0	0	119.107	120s
NS10000-D19	45000	155966	7620	148346	0	0	599.981	600s
NS10000-D19	45000	0	0	0	0	0	1619.96	3600s
NS10000-D20	45000	311310	20810	290500	0	0	79.228	initial solution
NS10000-D20	45000	299670	17870	281800	0	0	119.819	120s
NS10000-D20	45000	172026	8380	163646	0	0	598.973	600s
NS10000-D20	45000	0	0	0	0	0	1655.03	3600s

Table 3.5: Experimental results with 10000 network services

Instance	♯PM	total	cpu	mem	bandwidth	delay	time (s)	search
NS12000-D10	54000	1695749	310120	1348282	14240	23107	99.458	initial solution
NS12000-D10	54000	1689815	309070	1346382	11240	23123	117.827	120s
NS12000-D10	54000	1664382	304950	1330930	5240	23262	535.921	600s
NS12000-D10	54000	1523967	273290	1225326	2240	23111	3386.23	3600s
NS12000-D11	54000	1704638	210340	1476754	14240	3304	94.261	initial solution
NS12000-D11	54000	1692391	209190	1474654	5240	3307	118.203	120s
NS12000-D11	54000	1611535	195540	1410430	2240	3325	553.71	600s
NS12000-D11	54000	1547215	185320	1356328	2240	3327	3598.88	3600s
NS12000-D12	54000	1662884	196640	1450674	14240	1330	93.677	initial solution
NS12000-D12	54000	1650684	195740	1448374	5240	1330	119.387	120s
NS12000-D12	54000	1582098	183860	1394670	2240	1328	581.724	600s
NS12000-D12	54000	1543141	177300	1362252	2240	1349	3596.8	3600s
NS12000-D13	54000	1656946	195890	1445950	14240	866	93.257	initial solution
NS12000-D13	54000	1642096	195340	1443650	2240	866	119.803	120s
NS12000-D13	54000	1570226	183460	1383648	2240	878	540.849	600s
NS12000-D13	54000	1533968	177180	1353678	2240	870	3544.45	3600s
NS12000-D14	54000	1657901	193480	1449480	14240	701	92.581	initial solution
NS12000-D14	54000	1641907	192560	1446406	2240	701	118.727	120s
NS12000-D14	54000	1569118	179810	1386384	2240	684	530.745	600s
NS12000-D14	54000	1518459	172410	1343124	2240	685	3584.66	3600s
NS12000-D15	54000	1658688	196300	1447938	14240	210	85.409	initial solution
NS12000-D15	54000	1640742	194950	1443342	2240	210	119.427	120s
NS12000-D15	54000	1569793	182580	1384764	2240	209	568.367	600s
NS12000-D15	54000	1533058	176570	1354042	2240	206	3525.62	3600s
NS12000-D16	54000	1631378	185200	1431846	14240	92	93.953	initial solution
NS12000-D16	54000	1615530	184200	1428998	2240	92	118.679	120s
NS12000-D16	54000	1559214	174300	1382586	2240	88	452.124	600s
NS12000-D16	54000	1529050	169390	1357332	2240	88	3495.07	3600s
NS12000-D17	54000	1631352	185200	1431846	14240	66	92.969	initial solution
NS12000-D17	54000	1616552	184300	1429946	2240	66	119.643	120s
NS12000-D17	54000	1562684	174820	1385558	2240	66	452.824	600s
NS12000-D17	54000	1519322	167610	1349410	2240	62	3471.18	3600s
NS12000-D18	54000	1638102	185120	1438718	14240	24	92.289	initial solution
NS12000-D18	54000	1623202	184220	1436718	2240	24	119.167	120s
NS12000-D18	54000	1556406	172180	1381962	2240	24	552.786	600s
NS12000-D18	54000	1530706	167190	1361252	2240	24	3460.55	3600s
NS12000-D19	54000	1638090	185120	1438718	14240	12	89.281	initial solution
NS12000-D19	54000	1620552	183930	1434370	2240	12	119.855	120s
NS12000-D19	54000	1565202	173170	1389780	2240	12	581.104	600s
NS12000-D19	54000	1531438	167860	1361326	2240	12	3590.09	3600s
NS12000-D20	54000	1617558	182570	1420748	14240	0	92.137	initial solution
NS12000-D20	54000	1604958	181470	1418248	5240	0	118.911	120s
NS12000-D20	54000	1549318	171820	1375258	2240	0	542.369	600s
NS12000-D20	54000	1511524	165970	1343314	2240	0	3508.26	3600s

Table 3.6: Experimental results with 12000 network services

4 Processing Modules Design and Implementation

CHANGE uses comoddity hardware to perform flow processing. The Flowstream platform leverages a programmable switch to direct traffic to the x86 servers called *module hosts*. Here, traffic must be processed by user-defined processing functionality running inside *processing modules*.

CHANGE envisions that remote users can instantiate processing on CHANGE platforms. For efficiency, it is wise to run the processing modules belonging to different users on the same module host. Thus, the first requirement is to **multiplex processing belonging to different users on the same hardware**.

One user's processing should not impact another user's processing running on the same hardware, and should not be able to read other user's traffic or data. Hence, processing modules belonging to different users should be **isolated** from each other.

Finally, flow processing should offer **high performance**, making flow processing attractive to network operators and users alike.

Virtual machines are the obvious solution to achieve great isolation, and to offer good multiplexing of processing modules on the same hardware. The problem with using one virtual machine for each user is the high overhead this introduces: the CHANGE platform would run a full operating system for each user which limits both the performance measured in packets processed per second, packet delay as well as the number of users that can be accomodated on each physical machine.

An important part of the CHANGE project is to create new technologies that allow better performance, isolation and multiplexing than the state-of-the-art. This section is an overview of the work to date on these topics. We have been working three different but complementary approaches to solving the various problems of existing techniques.

netmap is a high-speed framework for user-level packet processing that was born and has already matured in the CHANGE project. *netmap* does away with the unnecessary fat accumulated in regular packet stacks by allowing fast access to (something resembling) NIC ring buffers directly from userspace, while also ensuring safety. *netmap* achieves impressive performance, being able to source 14 million packets per second using a single core running at 900Mhz; this is an order of magnitude better than existing approaches. We describe *netmap* in Section 4.1.3.

The second strand of work is called ClickOS and it sets out to build lightweight virtual machines that can run in existing hypervisors (such as XEN). ClickOS raises Click to the rank of an OS and it achieves the same isolation properties as Xen, but allows many more users to be multiplexed on the same hardware, and has the potential to achieve better performance. ClickOS is described in Section 4.2.4

Our third strand of work reasseses OS design in light of the requirements for flow processing. The result is FlowOS, which we are buidling from ground up to support flow processing. We review our work to date on

Flow OS in section 4.3.4.

These three approaches are being explored in parallel, and they complement each other. Both *netmap* and FlowOS coud be run within ClickOS if needed to ensure better isolation, while FlowOS could be implemented on top of *netmap* to leverage its very fast data path.

4.1 Netmap

General purpose operating systems provide a rich and flexible environment for running all sorts of applications, including those related to network operation, monitoring and testing. A peculiarity of these applications is the need to send or receive raw network packets at high speed and possibly without interference from the host stack. This is not the intended target of general purpose Operating Systems, which usually limit the access to raw packet access only to the kernel, to protect the system from untrusted user programs. The kernel itself has to access the network adapter (NIC) through an API (based on mbufs/skbufs/NdisPacket) which is extremely flexible, but heavyweight, as it requires every packet to be accompanied by a large amount of metadata, expensive to generate and manipulate.

Over time, several mechanisms have been proposed to address the application requirements for direct access to the network. Raw sockets, the Berkeley Packet Filter [38] (BPF), the AF_SOCKET family, and equivalent APIs have been proposed and used to build all sorts of network monitors, traffic generators, and generic routing systems. Performance, however, is not adequate the millions of packets per second (*pps*) that can be present on 1..10 Gbit/s links.

It is a common belief that the cost of crossing the kernel-userspace boundary is the main reason for this lack of performance. However, past experience with existing systems, and even our measurements in Section 4.1.9 show that even remaining in the kernel does not permit reaching the peak pps rates at 10 Gbit/s.

In search of better performance, some systems try to bypass the device driver and the entire network stack by exposing the network adapter's data structures to user space applications. We will provide a survey of these proposals in Section 4.1.2. Efficient as they may be, many of these systems have drawbacks. Most of them rely on specific devices and hardware features, and bypass operating system protections, putting system stability at risk. Equally often, the new APIs have a poor integration with the OS, so that their use from applications is cumbersome or inefficient.

As part of the CHANGE project we have designed *netmap* , a novel framework that permits very fast packet I/O even for userspace applications on commodity operating systems. In designing *netmap* , we identified and successfully removed three main cost factors in the path traversed by network packets. In order of importance: per-packet dynamic memory allocations and deallocations are removed by preallocating resources; system call overheads are reduced thanks to an API that handles multiple packets per call; and memory copies are eliminated by sharing metadata and memory buffers between kernel and userspace, while still protecting access to device registers and other kernel memory areas. Separately, some of these techniques have been used in the past. The novelty in our proposal is not only that we exceed the performance of most of previous work, but we also provide an architecture that is tightly integrated with existing operating system primitives, not tied to specific hardware, and easy to use and maintain.

Compared to other systems, some features are almost unique in *netmap* , and address the most significant shortcomings of research proposals or commercial products addressing fast packet processing. In particular, our API is extremely simple and supports existing applications, unmodified, through a libpcap compatibility library; and we focused on making the framework maintainable and easy to port to new hardware.

One metric to evaluate our framework is performance: in our implementation, moving one packet between the wire and the userspace application takes about 70 CPU clock cycles, which is at least one order of magnitude faster than standard APIs. In other words, a single core running at 900 MHz can source or sink the 14.88 Mpps achievable on a 10 Gbit/s link. The same core running at 150 MHz is well above the speed of a 1 Gbit/s link.

Other, equally important, metrics are safety of operation and ease of use. *netmap* clients cannot possibly crash the system, because device registers and critical kernel memory regions are not exposed to clients, and they cannot inject bogus memory pointers in the kernel. At the same time, *netmap* uses an extremely simple data model well suited to zero-copy packet forwarding; supports multi-queue adapters; and uses standard system calls (such as `select()/poll()`) to achieve synchronization between the hardware and the software. All this makes it very easy to port existing applications to the new mechanism, and to write new ones that make effective use of the *netmap* API.

4.1.1 Motivations and background

There has always been interest in using general purpose hardware and operating systems to run applications such as software switches [39], routers [23, 15, 22], firewalls, traffic monitors, intrusion detection systems, or traffic generators.

There is no doubt that a general purpose OS offers a convenient, flexible, rich software environment for running such applications. The software provides all sorts of functionalities so that applications requiring memory, databases, presentation capabilities find all they need at no cost. Additionally, the hardware where these OSs run tends to have the CPUs with the best price/performance ratios. I/O capabilities, often a weakness of general purpose hardware, are becoming less and less of a problem with modern I/O buses (e.g., PCI-Express), especially for applications that require a modest number of network ports.

It is unfortunate that general purpose OS normally do not offer efficient mechanisms to access raw packet data at high packet rates. Thus, the focus of this framework is to address this limitation.

As a matter of fact, the problem of coping with high packet rates is not limited to applications running in user space. There has always been a tension between the nominal speed of communication links, and the ability of systems to cope with their maximum packet rates. Link speeds grow over time in steps of one order of magnitude, with 10 Gbit/s and 14.88 Mpps being increasingly available nowadays. The speed of I/O and memory buses (other potential bottlenecks) also increases in large steps, but not necessarily synchronized with link speeds. As of this writing, a PCI-Express bus reaches 4 GBytes/s, whereas the memory bandwidth

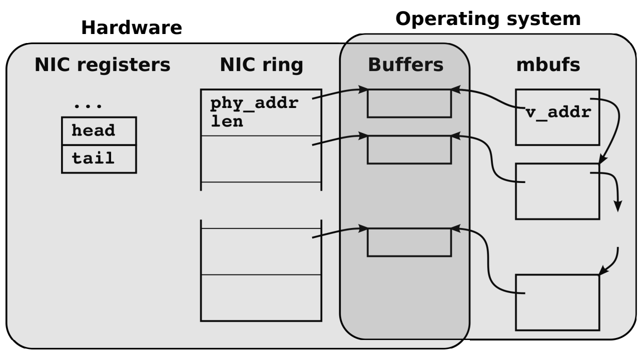


Figure 4.1: Typical NIC's data structures and their relation with the OS data structures.

is between 6 and 10 Gbytes/s on modern systems. CPU speeds increase much more gradually, with clock speeds topping around 3.5..4 GHz while the number of cores grows, sometimes to two digit figures.

As a result of all these factors, the ability to move packets between the network and applications changes over time, and depends critically on the relative speeds of the subsystems involved. This said, hardware is not the only performance bottleneck, and actually in many cases the blame for poor network performance goes to the software. The design of the software interface between network device drivers and the OS is in many cases over 20 years old, and the next subsections will describe it so that we can understand its limitations.

NIC data structures and operation. Network adapters (NICs) normally manage incoming and outgoing packets through circular queues (*rings*) of buffer descriptors, or slots, as in Figure 4.1. Each slot contains the length and physical address of the buffer, and some flags to control the buffer's handling by the NIC. CPU-accessible registers in the NIC indicate the portion of the ring containing buffers available for transmission or reception.

On reception, incoming packets are stored in the next available buffer (possibly split in multiple fragments), and length/status information is written back to the slot to indicate the availability of new data. Interrupts notify the CPU of these events. On the transmit side, the NIC expects the OS to fill buffers with data to be sent. The request to send new packets is issued by writing into one of the NIC's registers, which in turn starts sending packets marked as available in the TX ring.

The above arrangement is almost universal among modern NICs; variations are mostly on ring size, format of the slots, and restrictions on buffer sizes and alignments.

Some high speed NICs support multiple transmit and receive rings. This helps spreading the load on multiple CPU cores, eases on-NIC traffic filtering, and also improves decoupling virtual machines sharing the same hardware.

At high packet rates, the cost of interrupt processing may be significant. Early hardware generated interrupts at high rates [40], which caused a number of undesirable phenomena ranging from high per-packet processing costs to the so-called "receive livelock", or inability to do any useful work under extreme load. To overcome this problem, some systems use polling device drivers [30, 40, 44], and modern NICs implement interrupt mitigation directly in the hardware.

Interface between the NIC and the OS. The OS maintains shadow copies of the NIC's data structures. Buffers are linked to OS-specific, but device-independent containers (`mbufs` [49] or equivalent structures such as `skbufs` and `NdisPackets`). These containers include large amounts of metadata about each packet: size, source or destination interface, and attributes and command/status bits to indicate how the buffers should be processed by the NIC and the OS.

The software interface (API) between device drivers and the OS usually assumes that packets, in both directions, can be split into an arbitrary number of fragments, so both the device drivers and the host stack must be prepared to handle the fragmentation. The same API also expects that subsystems may retain packets for deferred processing, which means that buffers and metadata cannot be simply passed by reference during function calls, but they need to be copied or reference-counted. This flexibility is paid with a significant (and most of the times, unnecessary) overhead at runtime.

These API contracts, perhaps appropriate 20-30 years ago when they were designed, are extremely inadequate for today's systems. The cost of allocating, managing and navigating through buffer chains often exceeds that of linearizing their content, even when producers do indeed generate fragmented packets (e.g. TCP when prepending headers to data from the socket buffers).

Access from user programs. The standard OS APIs to read/write raw packets require at least one memory copy to move data and metadata between kernel and user space, and one system call per packet (or, in the best cases, per batch of packets). Typical approaches involve opening a socket, or a Berkeley Packet Filter [38] device, and doing regular I/O through it using `read()/write()` or specialized `ioctl()` functions. System calls are the dominant source of overhead in these APIs.

In summary, the result of these hardware and software architectures is that most systems barely reach 0.5..1 Mpps per core from userspace, and even remaining in the kernel yields only modest speed improvements, usually within a factor of 2.

4.1.2 Related work

It is useful at this point to present some techniques that have been proposed in the literature, or used in commercial systems, to improve packet processing speeds. This will be instrumental in understanding their advantages and limitations, and how our *netmap* framework can make use of them.

The Berkeley Packet Filter, or BPF [38], is one of the most popular systems to support direct access to raw packet data. It works by tapping into the data path of a network device driver, and dispatching a copy of each received or transmitted packet to a special device, from which userspace processes can read or write. Linux has a similar mechanism through the AF_PACKET socket family. BPF can coexist with regular traffic from/to the system, but usually BPF clients need to put the card in promiscuous mode, causing large amounts of traffic to be delivered to the host stack (and immediately dropped).

Packet filter hooks. BPF was designed to capture and generate packets without altering regular traffic from/to the system, but this is not always a requirement or even a desire. As an example, monitoring tools might want to keep the monitoring interface well separated from the rest of the system; firewalls and NATs *must* filter or modify traffic before it gets to the host stack; and in software routers or switches, most of the traffic does not need to be passed to the host stack at all. In these cases, a more efficient mechanisms is to implement packet hooks such as Netgraph (FreeBSD), Netfilter (Linux), Ndis Miniport drivers (Windows). These hooks permit to intercept traffic from/to the driver and pass it to processing modules without additional data copies. Note however that even the packet filter hooks rely on the standard mbuf/skbuf based packet representation, so the cost of metadata management still remains. Netslice [37] is an example of a system that uses the netfilter hooks to export traffic to userspace processes through a suitable kernel module.

Direct buffer access As mentioned, transferring information between kernel and user space involves system calls and data copies. One easy way to get rid of these operations is to run the application code directly within the kernel. Systems based on kernel-mode Click [30, 15] follow this approach. Click permits an easy construction of packet processing chains through the composition of modules, some of which support fast access to the NIC bypassing the OS (even though they retain an skbuf-based packet representation).

The kernel environment is however much more limited than the one available in user space, so this approach is not always convenient. As a consequence, a number of recent proposals try a different approach: instead of running the application in the kernel, they remove the system call overhead by exposing NIC registers and data structures to user space. This approach generally requires modified device drivers, and its use poses some stability risks at runtime, because the NIC has a DMA engine that can write to arbitrary memory addresses, and potentially trash data anywhere in the system if the wrong values are written to the NIC's data structures.

UIO-IXGBE [32] implements exactly what we have described above: buffers, hardware rings and NIC registers (see Figure 4.1) are directly accessible to user programs, with obvious risks for the stability of the system.

PF_RING [13] is a packet-capture solution somewhat related to direct buffer access. In PF_RING, user processes have access, through memory mapping, to a ring of pre-allocated buffers which contain packet data. The buffers are filled in software by copying the content of the mbufs/skbufs returned by the device driver. No individual driver modifications are needed. Compared to BPF, the fact that buffers are preallocated saves the per-packet allocation costs.

An evolution of PF_RING called DNA [14] avoids the copy because the memory mapped ring buffers are directly accessed by the NIC. Same as UIO-IXGBE, DNA clients have direct access to the registers and rings of the NIC.

Ringmap [19] is a packet capture solution with some similarities with the work presented here. It exports packet buffers and a shadow ring to userspace programs via memory mapping. Unlike our work, ringmap does not support the transmit side or communication with the host stack.

Finally, we would like to mention the PacketShader [22] I/O engine (PSIOE), which is one of the closest relatives to our proposals. PSIOE uses a custom device driver that replaces the skbuf-based API with a simpler one, using preallocated buffers. Custom `ioctl()`s are used to synchronize the kernel with userspace applications, and multiple packets are passed up and down through a memory area shared between the kernel and the application. The kernel is in charge of copying packet data between the shared memory and packet buffers. Unlike *netmap* , PSIOE only supports one specific network card, and does not support `select()`/`poll()`, requiring modifications to applications in order to let them use the PSIOE API.

Hardware solutions. We conclude our survey mentioning a number of hardware cards designed specifically to support high speed packet capture, or possibly generation, together with special features such as timestamping. Usually these cards come with custom device drivers and user libraries to access the hardware. As an example, DAG [6, 24] cards are FPGA-based devices designed to achieve wire-rate packet capture and precise timestamping. These cards use on-board memory to avoid depending on the performance of the bus (of course, access from the CPU will still have to go through the system bus). NetFPGA [36] is another example of an FPGA-based card where the firmware of the card can be programmed to implement specific functions directly in the NIC, offloading the main CPU from some work.

4.1.3 Solution

Our framework, called *netmap* , is a system to give user space applications very fast access to network packets, both on the receive and the transmit side, and including those from/to the host stack. Efficiency does not come at the expense of safety of operation: potentially dangerous actions such as programming the NIC are validated by the OS, which also enforces memory protection. Also, a distinctive feature of *netmap* is the attempt to design and implement an API that is simple to use, tightly integrated with existing OS mechanisms, and not tied to a specific device or hardware features.

netmap achieves its high performance through several techniques:

- a lightweight metadata representation which is compact, easy to use, and hides device-specific features. Also, the representation supports processing of large number of packets in each system call, thus amortizing its cost;
- linear, fixed size packet buffers that are preallocated when the device is opened, thus saving the cost of per-packet allocations and deallocations;
- removal of data-copy costs by granting applications direct, protected access to the packet buffers. The same mechanism also supports zero-copy transfer of packets between interfaces;
- support of useful hardware features (such as multiple hardware queues).

Overall, we use each part of the system for the task it is best suited to: the NIC to move data quickly between the network and memory, and the OS to enforce protection and provide support for synchronization.

At a very high level, when a program requests to put an interface in *netmap* mode, the NIC is partially disconnected (see Figure 4.2) from the host protocol stack. The program gains the ability to exchange packets with

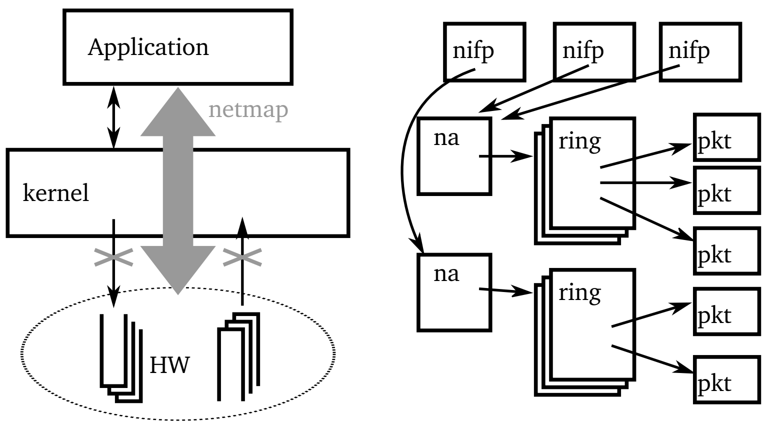


Figure 4.2: In netmap mode, the NIC rings are not driven by the host network stack anymore, and instead exchange packets through the netmap API. An additional pair of netmap rings is used to make the application talk to the host stack.

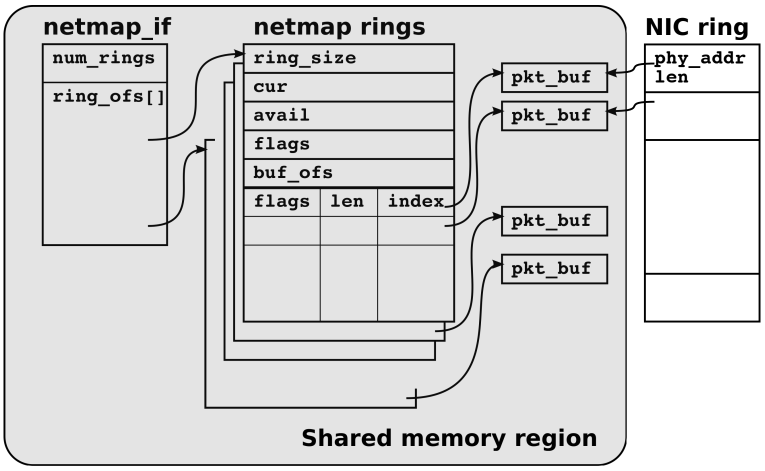


Figure 4.3: User view of the shared memory area exported by netmap.

the NIC and (separately) with the host stack, through circular queues of buffers (*netmap rings*) implemented in shared memory. Traditional OS primitives such as `select()/poll()` are used for synchronization. Apart from the disconnection in the data path, the operating system is unaware of the change so it still continues to use and manage the interface as during regular operation.

4.1.3.1 Basic operation

An application that wants to use *netmap* first calls `open("/dev/netmap")`. Subject to the permission checks performed by the OS, the system call returns a file descriptor used for subsequent operation. An interface can be bound to the file descriptor and switched to *netmap* mode by calling `ioctl(..., NIOCREG, ...)`. On a successful execution, the NIC's data path is disconnected from the host stack, and becomes accessible through data structures described in Figure 4.3 and Section 4.1.3.2.

Buffers in the TX and RX rings are made available to user processes in a shared memory region, accessible through the `mmap()` system call. Within the same region, the application can access a trimmed down, shadow version of the NIC's ring (we call this shadow structure *netmap ring*) addressing the buffers.

The netmap ring mimics the behaviour of the NIC's ring, in the sense that the application uses it to communicate with the NIC. However, the actual transfer of information occurs under control of a system call, either an `ioctl()` if the application wants to read/write without blocking, or a `select()/poll()` if we want to wait for certain events. Closing the file descriptor, or issuing another ioctl(), returns the NIC to its normal operation.

4.1.3.2 Data structures

The key component in the *netmap* architecture is the set of data structures represented in Figure 4.3. These are designed to support efficiently the various functions that we are interested in, namely:

- reduce/amortize per-packet overheads;
- efficient forwarding between interfaces;
- efficient communication between the NIC and the host stack;
- support for multi-queue adapters and multi core systems.

netmap supports these features by associating, to each interface, three types of user-visible objects, shown in Figure 4.3: *netmap_if* descriptors, *netmap rings*, and *packet buffers*. All objects for all netmap-enabled interfaces in the system reside in one large memory region, allocated by the kernel in a non-pageable area, and shared by all user processes. The use of a single buffer pool is just for convenience and not a strict requirement of the architecture. With little effort and almost negligible runtime overhead we can easily modify the API to have separate memory regions for different interfaces, or even introduce shadow copies of the data structures to reduce data sharing.

Since the memory region is mapped by processes and kernel threads in different virtual address spaces, any memory reference contained in that region must use *relative* addresses, so that pointers can be calculated in a position-independent way. The solution to this problem is to implement references as offsets between the parent and child data structures.

Packet buffers are just fixed-size buffers (2 Kbytes in the current implementation) used by the NICs and user processes to exchange packet data. Each buffer is identified by a unique *index*, which can be easily translated into a virtual address by user processes or by the kernel, and into a physical address used by the NIC's DMA engines. Buffers for all netmap rings are preallocated when the interface is put into netmap mode, so that during network I/O there is never the need to allocate memory, or a risk to run out of resources. The metadata describing the buffer (index, data length, some flags) are stored into *slots* which are part of the netmap rings described next. Each buffer is referenced by a netmap ring and by the corresponding hardware ring.

A *netmap ring* is a device-independent representation of the circular queue implemented by the hardware. This data structure contains the following fields:

- `ring_size`, the number of slots in the ring;
- `cur`, the index of the current read or write position in the ring;

- `avail`, the number of available buffers, representing received packets in RX rings, or empty slots in TX rings;
- `flags`, to indicate special conditions such as empty TX ring or errors;
- `buf_ofs`, indicating the memory offset between the ring and the beginning of the memory area containing buffers. From this and the ring address, processes can locate any packet buffer i in the system as:

  `(char *)ring + ring->buf_ofs + i * buf_len`
- `slots[]`, an array with `ring_size` entries. Each slot contains the index of the corresponding packet buffer, the length of the packet, and some flags used to request special operations on the buffer.

Finally, a *netmap_if* contains read-only information describing the interface, such as the number of rings and an array with the memory offsets between the `netmap_if` and each netmap ring associated to the interface (once again, offsets are used to make addressing position-independent).

4.1.3.3 Data ownership and access rules

The *netmap* data structures are shared between the kernel and userspace, but the ownership of the various data areas is well defined, so that there are no races. In particular, the *netmap_ring* is always owned by the userspace application except during the execution of a system call, when it is updated by the upper half of the kernel. The lower half of the kernel, which may be called by an interrupt, never touches a netmap ring.
Packet buffers between `cur` and `cur+avail-1` are owned by the userspace application, whereas the remaining buffers are owned by the kernel. The boundaries between these two regions are updated during system calls.

4.1.3.4 The netmap API

Programs put an interface in *netmap* mode by opening the special device `/dev/netmap` and issuing an

`ioctl(.., NIOCREG, arg)`

on the file descriptor. The argument contains the interface name, and optionally the indication of which rings we want to control through this file descriptor (see Section 4.1.3.6). On success, the function returns the size of the shared memory region where all data structures are located, and the offset of the `netmap_if` within the region. The shared memory region can be mapped in the process' address space using the `mmap()` system call.
Once the file descriptor is associated to an interface, two more `ioctl()`s support the transmission and reception of packets. In particular, transmissions require the program to fill up to `avail` buffers in the TX ring, starting from slot `cur` (packet lengths are written to the `len` field of the slot), and then issue an

`ioctl(.., NIOCTXSYNC)`

to tell the system that there are new packets to transmit. This system call passes the information to the kernel, and on return it updates the `avail` field in the netmap ring, reporting slots that have become available due to the completion of previous transmissions.

On the receive side, programs should first issue an

`ioctl(.., NIOCRXSYNC)`

to ask the system how many packets are available for reading; then their lengths and payload are immediately available through the content of the slots (starting from `cur`) in the netmap ring.

Both NIOC*SYNC `ioctl()`s are non blocking, involve no data copying (except from the synchronization of the slots in the netmap and hardware rings), and can deal with multiple packets at once. These features are essential to reduce the per-packet overhead to very small values. The in-kernel part of these system calls does the following:

- locates the interface and rings involved. This is done through a private kernel copy of the `netmap_if` which cannot be modified by user processes;
- validates the `cur` field and the content of the slots involved (lengths and buffer indexes, both in the netmap and hardware rings);
- synchronizes the content of the slots between the netmap and the hardware rings, and issues commands to the NIC to advertise new packets to send or newly receive buffers;
- updates the `avail` field in the netmap ring.

The amount of work in the kernel is kept to a minimum, and the checks performed make sure that any value written to the shared data structure cannot cause system crashes.

4.1.3.5 Blocking primitives

Support for blocking operation is provided through the well known `select()` and `poll()` system calls. Netmap file descriptors can be passed to such functions, and are reported as ready (causing the process to be woken up) when `avail` > 0. Before returning from a potentially blocking `select()/poll()`, the system executes the kernel part of the NIOC*SYNC functions, updating the status of the rings. This way, applications spinning on an event loop require only one system call per iteration.

4.1.3.6 Multi-queue interfaces

On cards with a single TX/RX ring, a file descriptor operates on the only ring pair available. For cards with multiple ring pairs, file descriptors (and the related `ioctl()` and `poll()`) can be configured in one of two modes, chosen through the `ring_id` field in the argument of the NIOCREG `ioctl()`. In the default mode, the file descriptor controls all rings, causing the kernel to check for available buffers on any of the rings. In the alternate mode, a file descriptor is associated to a single TX/RX ring pair. This allows multiple threads/processes to create separate file descriptors, bind them to different ring pairs, and to operate independently on the card without interference or need for synchronization. Binding a thread to a specific core just requires a standard OS system call, `setaffinity()`, without the need of any new mechanism.

4.1.3.7 Talking to the host stack

In addition to the netmap rings associated to the hardware ring pairs, each interface in *netmap* mode exports an additional ring pair, used to handle packets to/from the host stack. A file descriptor can in fact be associated to the "host stack" netmap rings using a special value for the `ring_id` field in the NIOCREG call. In this mode, NIOCTXSYNC encapsulates buffers into mbufs and then passes them to the host stack. Packets coming from the host stack are instead copied to the buffers in the netmap rings so that subsequent NIOCRXSYNC will report them as "received" packets.

Note that the network stack in the OS believes it has full control and access to a network interface even when this operates in netmap mode. As a consequence, it will happily try to communicate over that interface with external systems. It is then a responsibility of the netmap client to make sure that packets are properly passed between the rings connected to the host stack, and those connected to the NIC. Implementing this feature is straightforward, possibly even using the zero-copy technique shown in Section 4.1.6. This is also an ideal opportunity to implement functions such as firewalls, traffic shapers and NAT boxes, which are normally attached to packet filter hooks.

4.1.4 Safety considerations

The sharing of memory areas between the kernel and the multiple user processes who can open `/dev/netmap` poses the question of what safety implications exist in the usage of *netmap* .

Processes using *netmap* , even if misbehaving, cannot cause the kernel to crash, unlike many other high-performance packet I/O systems (e.g. UIO-IXGBE, PF_RING-DNA, in-kernel Click). In fact, the shared memory areas do not contain critical kernel memory areas, and buffer indexes and lengths are always validated by the kernel before being used.

There is a possibility that a misbehaving process corrupts someone else's netmap rings or packet buffers, and this is a direct consequence of the shared memory approach. There are two easy cures for this problem:

- we can restrict the shared memory regions to contain only the rings associated to file descriptors owned by a process. This protects a process from interference from other misbehaving netmap clients, and has no impact on operation or performance: a process can still own multiple descriptors/interfaces and pass packets among them without data copies.
- additionally, we can remove the sharing of data buffers between kernel and userland, forcing a copy instead (the content of netmap rings is already validated and copied). Users processes would still see buffers in a memory mapped area, but once they are passed to the kernel, any modification to the buffer content will not affect packets in transit. In this case there is a modest performance penalty coming from the data copy (in one of our experiments, copying 64-byte packets increased the processing time by less than 30 ns per packet).

We will explore the usefulness and performance of these two modifications in future work.

4.1.5 Example of use

To give an idea of the simplicity of use of the *netmap* API, below we show a real-life example – the core of the packet generator used in Section 4.1.9. Apart from a few macros used to navigate through the data structures in the shared memory region, netmap clients do not need any library to use the system, and the code is extremely compact and readable.

```
fds.fd = open("/dev/netmap", O_RDWR);
strcpy(nmr.nm_name, "ix0");
ioctl(fds.fd, NIOCREG, &nmr);
p = mmap(0, nmr.memsize, fds.fd);
nifp = NETMAP_IF(p, nmr.offset);
fds.events = POLLOUT;
for (;;) {
  poll(fds, 1, -1);
  for (r = 0; r < nmr.num_queues; r++) {
    ring = NETMAP_TXRING(nifp, r);
    while (ring->avail-- > 0) {
      i = ring->cur;
      buf = NETMAP_BUF(ring, ring->slot[i].buf_index);
      ... store the payload into buf ...
      ring->slot[i].len =  ... // set packet length
      ring->cur = NETMAP_NEXT(ring, i);
    }
  }
}
```

4.1.6 Zero-copy packet forwarding

The fact that all buffers for all interfaces reside in the same memory area permits zero-copy packet forwarding between interfaces in a very simple way. It suffices to swap the buffers between the receive slot on the incoming interface, and the transmit slot on the outgoing interface, and update the length and flags fields accordingly. The swap at the same time enqueues the packet on the output interface, and replenishes the input ring with an empty buffer without the need of resorting to a memory allocator. The relevant code is below:

```
  ...
  src = &src_nifp->slot[i]; /* locate src and dst slots */
  dst = &dst_nifp->slot[j];
  /* swap the buffers */
  tmp = dst->buf_index;
  dst->buf_index = src->buf_index;
  src->buf_index = tmp;
  /* update length and flags */
  dst->len = src->len;
  /* tell kernel to update addresses in the NIC rings */
  dst->flags = src->flags = BUF_CHANGED;
  ...
```

4.1.7 libpcap compatibility

An API is worth little if there are no applications that use it, and a significant obstacle to the deployment of new APIs is the need to adapt existing code to them.

Following a common approach to address compatibility problems, one of the first application we built for netmap was a small library that maps `libpcap` calls into *netmap* calls. The task was heavily simplified by the fact that *netmap* uses standard synchronization primitives, so we just needed to map the read/write functions (`pcap_dispatch()`/`pcap_inject()` into equivalent netmap functions. The whole implementation for these functions is below:

```
int pcap_inject(pcap_t *p, void *buf, size_t size)
{
  int si, idx;

  for (si = p->begin; si < p->end; si++) {
    struct netmap_ring *ring = NETMAP_TXRING(p->nifp, si);

    if (ring->avail == 0)
      continue;
    idx = ring->slot[ring->cur].buf_idx;
    bcopy(buf, NETMAP_BUF(ring, idx), size);
    ring->cur = NETMAP_RING_NEXT(ring, ring->cur);
    ring->avail--;
    return size;
  }
  return -1;
}

int pcap_dispatch(pcap_t *p, int cnt,
    pcap_handler callback, u_char *user)
{
  int si, ret = 0;
  for (si =p->begin; si < p->end; si++) {
    struct netmap_ring *ring = NETMAP_RXRING(p->nifp, si);

    while ((cnt == -1 || ret != cnt) && ring->avail > 0) {
      int i = ring->cur;
      int idx = ring->slot[i].buf_idx;

      p->hdr.len = p->hdr.caplen = ring->slot[i].len;
      callback(user, &p->hdr, NETMAP_BUF(ring, idx));
      ring->cur = NETMAP_RING_NEXT(ring, i);
      ring->avail--;
      ret++;
    }
  }
  return ret;
}
```

4.1.8 Implementation

In the design and development of *netmap* , a fair amount of work has been put into making the system maintainable and performant, so it is useful to discuss some of the implementation details.

The current prototype of *netmap* , developed on FreeBSD, consists of about 2000 lines of code for system call (ioctl, select/poll) and driver support. As mentioned, there is no need for a userspace library, and a small C header (200 lines) contains all the necessary definitions for the structures, prototypes and macros used by *netmap* clients.

To reduce the amount of modifications to existing device drivers (a must, if we want the API to be implemented on new hardware), most of the functionalities are implemented in common code, and each device driver only needs to implement two functions roughly corresponding to the core of the NIOC*SYNC routines, one function to reinitialize the rings in netmap mode, and one function to export device driver locks to the common code. This reduces individual driver modifications, mostly mechanical, to about 500 lines each, (a typical device driver has 4k .. 10k lines of code).

In the *netmap* architecture, device drivers are expected to perform most of their work in the context of the userspace process. This simplifies resource management (e.g. binding processes to specific cores), and generally makes the system more controllable and robust, as we do not need to worry of executing too much work in non-interruptible contexts.

We generally modify drivers so that the interrupt service routine does almost no work except waking up any sleeping process. This means that interrupt mitigation delays are directly passed to user processes. Synchronization between the hardware and netmap rings is done in the upper half of the system calls, and in a way that avoids expensive operations. As an example, we don't reclaim transmitted buffers or look for more incoming packets if a system call is invoked with `avail` > 0. This provides huge speedups for applications that unnecessarily invoke system calls on every packet.

Two more optimizations (implicitly pushing out any packets queued for transmission even if POLLOUT is not specified; and optionally updating a timestamp within the netmap ring before `poll()` returns) help reducing from 3 to 1 the number of system calls in each iteration of the typical event loop – once again a significant performance enhancement for certain applications.

To date we have not tried to add special performance optimizations to the code, such as aggressive use of `prefetch` instructions, or data placements to improve cache 0. *netmap* support is currently available for the Intel 10 Gbit/s adapters (ixgbe driver), and for various 1 Gbit/s adapters (Intel, RealTek, Nvidia). Support for slower interfaces (e.g. 100 Mbit) is only interesting for embedded platforms, where CPU cycles tend to be scarce even for relatively low data rates.

4.1.9 Performance analysis

In this Section we discuss the performance of our framework by first 0 its 0 for simple I/O functions, and then looking at the 0 of more complex applications running on top of *netmap* . Before presenting our results, it is important to define the test conditions in detail.

4.1.9.1 Performance metrics

Processing a packet involves the activity of multiple subsystems, including CPU pipelines, caches, memory and I/O buses. In our measurements we will assume that the bottleneck is the CPU, so we will focus primarily on the CPU cycles consumed for processing a packet. Part of these cycles will be spent within the actual application (a traffic generator, router, firewall, etc.) and obviously depend heavily on the task performed. Another part, which we call *system costs*, is instead independent of the particular application, and covers the work performed to move packets between the application and the network card. This is precisely the task that *netmap* or other packet-I/O APIs are in charge of. As a consequence, our evaluation will try to isolate and evaluate the CPU cycles that we classify as *system costs*.

We can further split these cycles in two components:

i) Per-byte costs are the CPU cycles consumed to move data from/to the NIC's buffers (for reading or writing a packet). This component can be equal to zero in some cases: as an example, *netmap* exports NIC buffers to the application, so it has no per-byte system costs. Other APIs, such as the socket API, impose a data copy to move traffic from/to userspace, and this has a per-byte CPU cost that taking into account the width of memory buses and the ratio between CPU and memory bus clocks, can be in the range of 1/4 to 2 clock cycles per byte.

ii) Per-packet costs have multiple origins. At the very least, the CPU must update a slot in the NIC ring for each packet. Additionally, depending on the software architecture, each packet might require additional work, such as memory allocations, system calls, programming the NIC's registers, updating statistics and the like. In some cases, part of the operations in the second set can be removed or amortized over multiple packets.

Given that in most cases (and certainly this is true for *netmap*) *per-packet* costs are the dominating component, the most challenging situation in terms of system load is when the link is traversed by the smallest possible packets. This explains why we run most of our tests with 64 byte packets (60 + 4 for the Ethernet CRC).

Of course, in order to exercise the system and measure its performance we need to run some application, but we want it to be as simple as possible in order to reduce the interference on the measurement. Our initial tests then use two very simple programs that make application costs almost negligible: a packet generator which streams pre-generated packets, similar to the one presented in Section 4.1.5, and a packet receiver which just counts incoming packets.

As a final remark on testing, we should note that there are other system components (I/O and memory buses and related interface logic) which sometimes can act as bottlenecks. An example will be given in Section 4.1.9.5.

We should also remember that at the speeds at which we are operating, sometimes even minor issues such as memory alignment and data layout can easily cause costly cache misses that impact the throughput significantly, especially at the highest packet rates.

4.1.9.2 Test equipment (hardware and software)

We have run most of our experiments on systems equipped with an i7-870 4-core CPU running at a top speed of 2.93GHz (reaching 3.2 GHz with turbo-boost), memory running at 1.33 GHz, and a dual port 10 Gbit/s card based on the Intel 82599 chip mounted on a x16 PCI-Express slot (only 8 lanes are used by the card)[1]. The operating system is FreeBSD/i386 as of summer 2011 (both HEAD and RELENG_8 exhibit similar performance). Experiments have been run using directly connected cards on two similar systems. For a given version of the *netmap* software, the experimental results are highly repeatable (within 2% or less) so we do not report confidence intervals in the tables and graphs.

netmap is extremely efficient so the CPU is mostly idle even when the interface is running at the maximum packet rates. Running the system at reduced clock speeds also helps determining the effect of small performance improvements. Our system can be clocked at different frequencies, taken from a discrete set of values. Nominally, most of them are multiples of 150 MHz, but we don't know how precise are the clock speeds, nor the relation between CPU and memory/bus clock speeds.

The transmit speed (in packets per second) has been measured with a packet generator similar to the one in Section 4.1.5. The packet size can be configured at runtime, as well as the number of queues and threads/cores used to send traffic. Packets are prepared in advance so that we can run the tests with close to zero per-byte costs. The test program loops around a poll(), sending at most B packets (*batch size*) per ring at each round. On the receive side we use a similar program, except that this time we poll for read events and only count

[1]We have also run the tests with some 1 Gbit/s cards, but the results are not particularly interesting because the peak speed is well below the capability of the CPU. We also found that all of our 1 Gbit/s cards, presumably due to internal limitations, were unable to reach line rate irrespective of CPU speed and operating system.

Clock (GHz)	1 core 4 rings	4 cores 4 rings
0.150	2.17	5.35
0.300	4.57	10.13
0.450	7.46	14.88
0.600	10.87	14.88
0.750	13.64	14.88
0.900	14.88	14.88
1.050	14.88	14.88

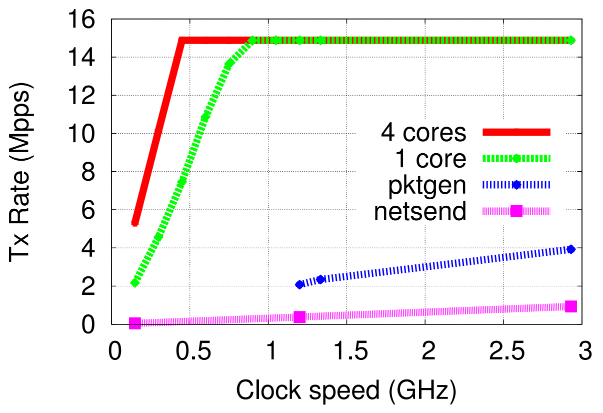


Figure 4.4: Transmit performance with 64-byte packets, variable clock rates and number of cores. The system reaches line rate near 900 MHz. The two lines at the bottom represent a 0, in-kernel generator available on linux (pktgen, peaking at about 4 Mpps at 2.93 GHz) and a FreeBSD userspace generators (netsend, peaking at 790 Kpps).

packets.

4.1.9.3 Transmit speed versus clock rate

As a first experiment we ran the generator with variable clock speeds and number of cores, using a large batch size so that the system call cost is almost negligible, and throughput is maximized. *netmap* is able to saturate a link even with just one core. By lowering the clock frequency we can determine the point where the CPU actually becomes the bottleneck for performance, and estimate an (amortized) number of cycles spent for each packet.

Figure 4.4 shows a subset of our results, using 4 rings (the NIC we used, as shown in Section 4.1.9.6, cannot send at line rate with minimum packet sizes using only 1 ring). In one of the experiments, a single thread is sending to all rings, while in another we assign each ring to a separate thread.

Our experimental results show the throughput scales quite well with clock speed, reaching the maximum line rate near 900 MHz[2] Dividing the clock rate by the packet rate we find an average of 70-80 cycles/packet with 1 core, a value which is reasonably in line with our expectations. In fact, in this particular test, the per-packet work is limited to validating the content of the slot in the netmap ring and updating the corresponding slot in the NIC ring. The cost of cache misses (which do exist, especially on the NIC ring) is amortized among all descriptors that fit into a cache line, and other costs (such as reading/writing the NIC's registers) are amortized

[2]previous versions of netmap were slower, reaching line rate slightly above 1.2 GHz.

over the entire batch.

The measurements exhibit a slightly superlinear 0 (e.g. when doubling the clock speed between 150 and 300 MHz), though in absolute terms the deviation is very small (3-4 clock cycles per packet), and could be easily explained by small mismatches between the reported and actual frequencies, or non-uniform changes in CPU, memory and I/O bus speeds as the clock speed changes. Not shown by the graphs, but measured in the experiments, is the fact that once the system reaches line rate, increasing the clock speed reduces the CPU usage. The generator in fact sleeps until an interrupt from the NIC reports the availability of new buffers, and wakes up the process.

Experiments using 4 cores show a speedup of about 2.5 times over the 1-core case. The speedup is modest, but might be explained by the sharing of several system components (caches, memory and I/O buses) among the four competing threads.

It is useful to compare the performance of our *netmap* -based generator with similar applications using conventional APIs. As a reference, Figure 4.4 also reports the maximum throughput of two packet generators representative of the performance achievable using standard APIs. The line at the bottom represents `netsend`, a FreeBSD userspace application running on top of a raw socket. `netsend` peaks at 930 Kpps at the highest clock speed (2.93 GHz plus turbo boost), and does 390 Kpps at 1.2 GHz. The 40x difference in speed can be explained with the many additional operations that the raw socket API requires: data copies, one system call per packet, in-kernel buffer allocations, and no chance to amortize the cost of accessing the NIC's registers.

The other line in the graph is `pktgen`, an in-kernel packet generator available in Linux, which reaches almost 4 Mpps at maximum clock speed, and 2 Mpps at 1.2 GHz (the minimum speed we could set in Linux). In this case the system call and memory copy costs are removed, but there are still device driver overheads (skbuf allocations and deallocations, NIC programming) that make the system at least 7 times slower than *netmap* .

4.1.9.4 Receive speed

The speed vs. clock rate experiments on the receive path give results similar to the transmit section. The system can do line rate, at least for packet sizes multiple of 64 (see Section 4.1.9.5) even below 1 GHz. In this case even using just one queue suffices to reach line rate.

4.1.9.5 Speed versus packet size

The experiments reported so far used minimum-size packets, which is the most critical situation in terms of per-packet overhead. 64-byte packets match very well the bus widths along the various path in the system and this helps the performance of the system. We then checked whether varying the packet size has an impact on the performance of the system, both on the transmit and the receive side.

Transmit speeds with variable packet sizes exhibit the expected $1/size$ 0, as shown by the upper curve in Figure 4.5.

The receive side, instead, shows some surprises as indicated by the bottom curve in Figure 4.5. The maximum rate, irrespective of CPU speed and number of rings, is achieved only for certain packet sizes such as 64+4,

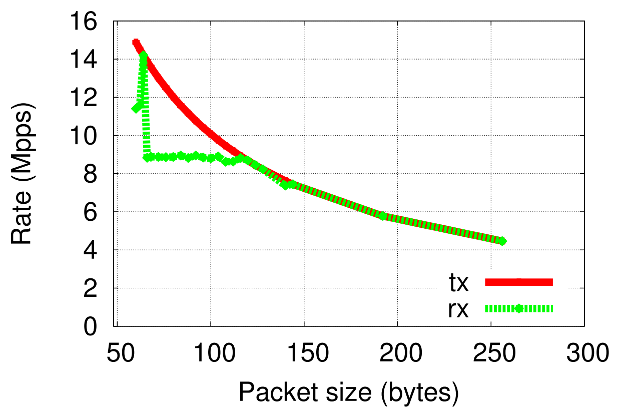


Figure 4.5: Actual transmit and receive speed with variable packet sizes (excluding Ethernet CRC). The top curve is the transmit rate, the bottom curve is the receive rate. See Section 4.1.9.5 for explanations.

128+4 and above 144+4 bytes (in all cases, the extra 4 bytes corresponding to the Ethernet CRC is not written to memory). At other packet sizes, performance drops (e.g. we see 11.6 Mpps at 60 bytes, and a plateau around 8.5 Mpps between 65 and 127 bytes). We briefly investigated this 0 looking at the NIC's statistic counters. As confirmed by other sources, it appears that this particular NIC is starving on the PCI-Express bus, presumably because it tries to do a read-modify-write sequence when receiving packets that are not a multiple of 64 bytes.

We should note that this 0 has no architectural relation with the way *netmap* works, but it depends purely on the implementation of bus transactions on the NIC. Yet, in a very high speed packet processing system, we should be aware that these phenomena may occur and not immediately blame applications for unexpected performance numbers.

4.1.9.6 Transmit speed versus batch size

Operating with large batches enhances the throughput of the system as it amortizes the cost of system calls and other potentially expensive operations. But not all applications have this luxury, and in some cases they are forced to operate in regimes where a system call is issued on each/every few packets.

To test performance in these conditions, we ran another set of experiments using different batch sizes, this time running the system at minimum-size packets (64 bytes, including the Ethernet CRC). The goal of this experiment is to determine the system call overhead, and, as a related result, what kind of batch sizes permit reaching line rate. In this particular test we only use one core, and variable number of queues. Results are summarized in Figure 4.6 (the receive side exhibits similar numbers, except that it can do line rate even when using a single ring).

As we see from the data, performance grows almost linearly with the batch size, up to the maximum value, which is approximately 12.5 Mpps with one queue, and 14.88 Mpps with 2 or more queues. The low speed achieved with a batch size of 1 shows that the cost of the `poll()` system call is much larger than the per-packet cost measured in Section 4.1.9.3, and so it is important to amortize it over large batches.

With some surprise, we found out that using a single queue (an experiment which was not done previously)

Burst size	1 ring	2 rings	4 rings
1	1.41	1.91	2.15
2	2.77	3.65	4.32
4	5.44	7.15	7.73
8	10.03	12.84	13.43
16	12.48	14.88	14.88
32	12.51	14.88	14.88

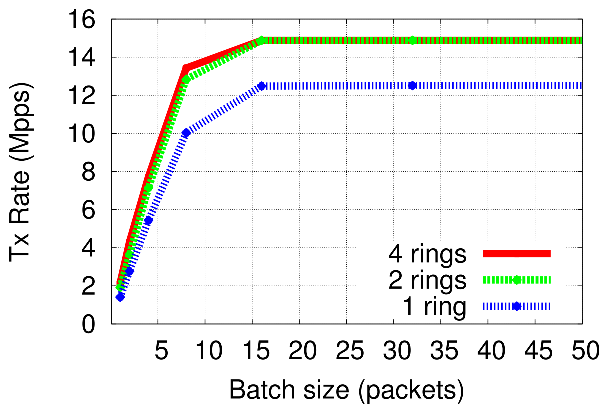


Figure 4.6: Transmit performance with 1 core at 2.93 GHz, 64-byte packets, and variable number of rings and burst size.

we could not reach the line rate but saturated near 12.5 Mpps. Further tests with lower clock speeds reached the same maximum rate, suggesting that the bottleneck is not the CPU but must be searched in the NIC and its interface to the PCI-Express bus. This throughput limitation is not present with 2 and 4 queues, which allow the generator to reach the maximum packet rate achievable on the link.

4.1.9.7 Packet forwarding performance

The experiments described so far measure the system costs involved in moving packets between the wire and memory. This includes the operating systems overhead, but excludes any significant application cost, as well as any data touching operation.

It is then interesting to measure the benefit of the *netmap* API when used by more CPU-intensive tasks. Packet forwarding is one of the main applications of packet processing systems, and a good test case for our framework. In fact it involves simultaneous reception and transmission (thus potentially causing memory and bus contention), and may involve some packet processing that consumes CPU cycles, and causes pipeline stalls and cache conflicts. All these phenomena will likely reduce the benefits of using a fast packet I/O mechanism, compared to the simple packet generators and receivers used so far.

We have then explored how a few packet forwarding applications behave when using the new API, either directly or through the `libpcap` compatibility library described in Section 4.1.7. The applications we used are the following:

- `netmap-fwd`, a simple application that forwards packets between interfaces using the zero-copy technique shown in Section 4.1.6;

Configuration	Mpps
netmap-fwd	10.66
netmap-fwd + ts	9.42
netmap-fwd + pcap	7.50
click-fwd + netmap	3.95
click-etherswitch + netmap	3.10
click-fwd + native pcap	0.49
openvswitch + netmap	3.00
openvswitch + native pcap	0.78
bsd-bridge	0.75

Figure 4.7: Forwarding performance in various configurations.

- `netmap-fwd + ts`, as above, but requesting the API to update a timestamp on each `poll()` (timestamps are frequently used by packet processing applications, and embedding their generation in an existing system call remove the need for additional expensive system calls);
- `netmap-fwd + pcap`, as above but using the libpcap emulation (Section 4.1.7) instead of the zero-copy code;
- `click-fwd`, a simple Click [30] configuration shown below, that passes packets between interfaces:

  ```
  FromDevice(ix0) -> Queue -> ToDevice(ix1)
  FromDevice(ix1) -> Queue -> ToDevice(ix0)
  ```

  The experiment has been run using Click userspace with the system's libpcap, and on top of netmap with the libpcap emulation library;
- `click-etherswitch`, as above but replacing the two queues with an EtherSwitch element;
- `bsd-bridge`, in-kernel FreeBSD bridging, using the mbuf-based device driver;
- `openvswitch`, the OpenvSwitch software with userspace forwarding, both with the system's libpcap and on top of netmap.

Figure 4.7 reports the measured performance. All experiments have been run on a single core with maximum clock rate and two 10 Gbit/s interfaces.

The various `netmap-fwd` entries show that moving packets between interfaces can be almost 10 times faster than using the same task using native APIs (in-kernel bridge, openvswitch). Comparison of individual values also gives some useful indications. As an example, the entries "netmap-fwd + ts" and "netmap-fwd + pcap" essentially differ for the presence of a data copy step on the transmit side (performed in `pcap_inject()`. Computing the per-packet times we see that the difference between the two measurements is less than 30 ns, which is an upper bound for the packet copy cost.

The `click-fwd` entries show that replacing the native libpcap with the netmap-based version gives a speedup of 8 times, and similar results also hold for slightly more complex tasks such as click-etherswitch. OpenvSwitch also shows an improvement, though more limited (about four times, but see the discussion in Section 4.1.9.9).

4.1.9.8 Discussion

In presence of huge performance improvements such as those presented in Figure 4.4, and Figure 4.7, which show that *netmap* is 4 to 40 times faster than similar applications using the standard APIs, one might wonder about two things: i) are we doing a fair comparison, and ii) what is the contribution of the various mechanisms to the performance improvement.

The answer to the first question is that the comparison is indeed fair. All the various generators do exactly the same thing and try to do it in the most efficient way, constrained only by the underlying APIs they use. The answer is even more obvious for Figure 4.7, where in many cases we just use the same unmodified binary on top of two different `libpcap` implementations.

The results measured in different configurations also let us answer the second question – evaluate the impact of different optimizations on the *netmap* 's performance.

Data copies, as shown in the previous Section, are relatively inexpensive (though their massive use can cause cache pollution). This is an interesting result, in light of the fact that we may want to reintroduce data copies to reduce the interference among netmap clients (see Section 4.1.4)

Per-packet system calls certainly play a major role, as witnessed by the difference between netsend and pktgen (albeit on different platforms), or by the low performance of the packet generator when using small batch sizes.

Finally, an interesting information on the cost of the skbuf/mbuf-based API comes from the comparison of pktgen (taking about 250 ns/pkt, of which perhaps 25-30 can be attributed to memory copies) and the netmap-based packet generator, which only takes 20-30 ns per packet which are spent in programming the NIC).

4.1.9.9 Running applications on top of netmap

We conclude our analysis with a brief discussion of the issues encountered in adapting existing applications to netmap. In principle, running an existing application on top of our libpcap library should not require any modification to the code, and just replacing the underlying shared library should suffice. However, it is possible that the application itself has performance bottlenecks that prevent the exploitation of a faster I/O subsystem, and this is exactly the case we encountered with two applications, OpenvSwitch and Click (full details are described in [45]).

In the case of OpenvSwitch, the original code (with the userspace forwarding module) was running a very expensive event loop, and could only do less than 70 Kpps. Replacing the native libpcap with the netmap-based version gave almost no measurable improvement. After restructuring the event loop and splitting the

	Total power (W, +/- 2%)				
Clock	Idle	1 core	4 cores	tx(1)	tx(4)
150	70	71	72	72	75
300	70	72	73	73	76
600	70	72	77	75	81
1050	70	72	79	79	84
1200	71	74	82	79	84
1467	71	76	87	84	86
2133	71	83	102	97	98
2400	71	87	110	101	102
2800	71	92	124	108	110
2934* (Turbo)	71	108	155	128	129

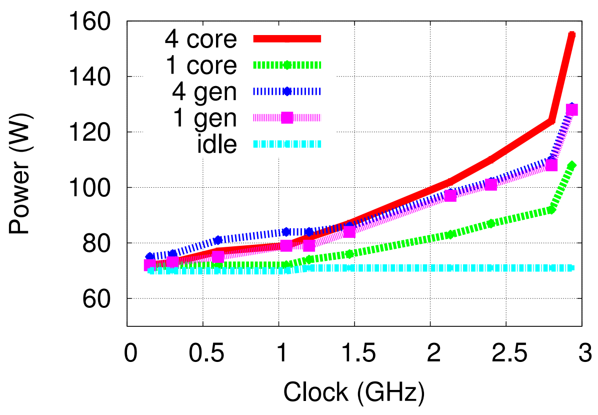


Figure 4.8: Total system power in various conditions.

system in two processes, the native performance went up to 780 Kpps and the netmap based libpcap further raised the forwarding performance to almost 3 Mpps.

In the case of Click, we encountered similar issues, this time for different reasons. First of all, several Click elements add a timestamp to packets, and depending on the platform this may require a costly system call. Second, the buffers used to store packets' data and metadata were allocated using the C++ allocator, which is significantly more expensive than managing a private pool of fixed-size blocks. These two operations have little impact when the I/O subsystem is slow, but a large one when Click runs on top of netmap. As an example, replacing the memory allocator brought the forwarding performance from 1.3 Mpps to 3.95 Mpps when run over netmap, and from 0.40 to 0.495 Mpps when run on the standard libpcap. As for the effect of timestamps, a simple configuration `InfiniteSource -> ToDevice` on top of netmap can reach 9 Mpps disabling timestamps, but only 1 Mpps with timestamps.

4.1.10 Future work

We are still exploring some aspects of the performance of *netmap* . As an example, communication latency is important for certain applications which require many RPCs, or fast response times. Usually latency and throughput are contrasting requirements – optimizing the latter normally requires aggressive use of batching, which in turn penalizes latency. However, *netmap* introduces significant simplifications in the device drivers and system calls, so we expect the latency for netmap clients to be similar to that of in-kernel applications using conventional APIs. As an example, between two hosts connected by a 10 Gbit/s interface, we measured a ping response time (median) of 17 μs using the native host stack, and 20 μs when the response is generated by a userspace netmap client, despite the extra cost for scheduling the process and running two system calls

We also plan to experiment with userspace version of certain kernel subsystems (firewalls, forwarding engines, even protocol stacks). The goal is twofold: first, we want to evaluate how the netmap APIs can simplify these applications; second, having them in userspace makes it a lot easier to experiment alternatives to the algorithms currently implemented in the OS.

4.2 ClickOS

As can be seen from the previous sections, the module hosts and their processing elements need to fulfill certain requirements: The elements should provide good performance and be flexible in terms of processing capabilities, and - as a variety of applications might be deployed by different users - isolation mechanisms and support for potentially many domains are also a major concern. Furthermore, solutions for flow migration and management of per-flow state are needed to provide a unified framework for applications.

To this end there are several viable approaches that support these requirements, including minimalistic OSes (e.g. tiny Linux distributions, minix, Exokernel, L4, etc.), container-based virtualization (KVM, openVZ, etc.) and full virtualization (Xen, Vmware, etc.).

This section will give an overview about *ClickOS*, a small and light-weight operating system targeted mainly at network packet processing. ClickOS combines the *Click* software router framework [31] with *MiniOS*, a minimalistic example OS implementation [5] that is provided as part of the XEN source tree. The OS runs as a XEN stub guest domain - this low-overhead domain type was originally intended to run (untrusted) drivers in an isolated environment. Newer versions of MiniOS support the newlib C library and the lwIP stack, thus providing a basic POSIX enviroment, including TCP/IP networking. In addition, ClickOS control plane functionality is provided via Xenstore/Xenbus (see fig. 4.9).

In general, ClickOS is intended to provide a low-overhead, massively scalable solution based on standard Xen hypervisor technology. In contrast to existing solutions like tiny Linux-based OSes, it will be possible to support 100s or even 1000s of running ClickOS instances on a single consumer PC.

ClickOS takes advantage of features provided by XEN, like domain isolation, driver support and certain performance features (e.g. device passthrough) and combines them with the processing flexibility provided by the Click framework.

By supporting recently introduced multi-queue NIC hardware (like the Intel 10G cards based on the 82598 chipset) and leveraging Xen's enhanced para-virtualization features, the full 10 Gbit/s network bandwidth provided by modern consumer equipment can be utilized.

However, this approach currently focuses on providing a Click-based network processing environment, which means that support for legacy applications is only possible through dedicating a non-ClickOS domain to them (without efficient flow migration). It also needs to be further evaluated whether Click is flexible enough to support all desired kinds of processing needed.

Building ClickOS required major efforts in merging the source trees of XEN, MiniOS, and the Click framework and setting up a unified build system. As ClickOS is not based on Linux, the user-level Click codepath was used.

Currently, 57 out of the original 93 Click elements compile under ClickOS, the remaining elements make use of specific Linux (kernel) features which are not available in ClickOS at the moment (e.g. kernel data structures like `sk_buff` or raw sockets) or rely on filesystem support (which is work-in-progress). Some of these elements may be ported to ClickOS in the future as well, either by adding additional functionality to the ClickOS kernel itself or by adapting the original element implementation.

Even with all elements compiled into the binary the ClickOS image right now is only 1MB in size. This small memory footprint potentially allows to runs 100's of instances on a single machine - in fact, initial tests have shown that a modern 64bit machine is indeed able to run over 1000 ClickOS instances at the same time. Its fast boot-up times (in the order of 10 seconds) make ClickOS especially suited for just-in-time live migration scenarios, clearly outperforming standard desktop OSes which usually require boot times in the order of minutes and have much bigger image sizes (typically around 0.5 to 1GB).

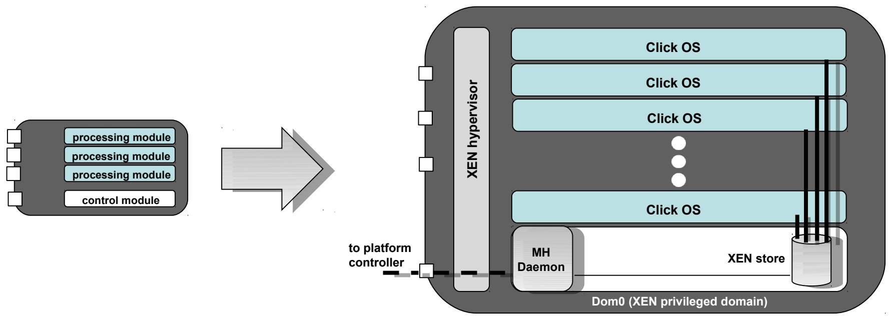


Figure 4.9: Xen and ClickOS

4.2.1 The Click Modular Router

As outlined above, ClickOS allows applications to be built on top of the Click infrastructure. This section gives a short introduction of the basic concepts behind Click and high-level background information on how Click works internally.

Click is a modular router software framework [31][2] that allows to flexibly compose packet processing functionality using provided building blocks. The provided elements cover a wide range of functionality like NIC I/O, header parsing, filtering, queueing, etc. and are implemented as C++ classes.

By specifying a graph of elements in a configuration file users can setup complex packet processing pipelines without writing C++ code. The mechanisms for dynamic instantiation of elements, the communication and forwarding of data between them, and all scheduling-related issues are taken care of by the framework.

For example, the following configuration (see http://read.cs.ucla.edu/click/examples/test.click) will print 5 sample packets and then stop:

```
InfiniteSource(DATA \<00 00 c0 ae 67 ef  00 00 00 00 00 00  08 00
45 00 00 28  00 00 00 00  40 11 77 c3  01 00 00 01
02 00 00 02  13 69 13 69  00 14 d6 41  55 44 50 20
70 61 63 6b  65 74 21 0a>, LIMIT 5, STOP true)
      -> Strip(14)
```

```
    -> Align(4, 0)     // in case we're not on x86
    -> CheckIPHeader(BADSRC 18.26.4.255 2.255.255.255 1.255.255.255)
    -> Print(ok)
    -> Discard;
```

When this configuration is passed to the Click runtime engine, it will dynamically instantiate all required elements (like `Strip`, `Align`, `CheckIPHeader`, etc.),wire them up accordingly (internally all Click elements have input and output ports to connect/communicate with each other) and then start its scheduling and processing mainloop.
In general, Click elements pass data among each other using a well-defined packet structure (defined in `include/click/packet.hh`). This is the object that most element execution methods expect to receive and/or output.
Internally, Click schedules the routers CPU with a task queue. Each router thread runs a loop that processes the task queue one element at a time. The task queue is scheduled with the flexible and lightweight stride scheduling algorithm. Tasks are simply elements that would like special access to CPU time. Thus, elements are Clicks unit of CPU scheduling as well as its unit of packet processing. An element should be on the task queue if it frequently initiates push or pull requests without receiving a corresponding request. For example, an element that polls a device driver should be placed on the task queue; when run, it would remove packets from the driver and push them into the configuration. However, most elements are never placed on the task queue. They are implicitly scheduled when their push or pull methods are called. Once an element is scheduled, either explicitly or implicitly, it can initiate an arbitrary sequence of push and pull requests, thus implicitly scheduling other elements.

4.2.2 ClickOS Network I/O

In general, paravirtualized network I/O in Xen is achieved through a pair of interlinked drivers; `netfront` the frontend driver in the guest, and `netback` the backend driver in the host domain (fig. 4.10)
The frontend and backend communicate through a region of shared memory provided by Xen's grant table API. Virtual interrupts based on Xen event channels coordinate the shared access to memory when transfering packets between host and guest domain.
The upper edge of the frontend driver presents the interface of a standard network device driver, allowing it to interface to the bottom of the guest's network stack. The backend appears likewise and is usually configured to connect to a software bridge in the host OS. This allows it to communicate with the host's network stack, other virtual machines' backend drivers, and physical network interfaces and so the network beyond.
Packets that arrive from a physical network interface are routed by the bridge to the appropriate backend drivers, which in turn forward them to the corresponding guests' frontend drivers. These then pass them on to the network stack as if they had arrived directly at the guests. Packets sent by guests follow the same path in reverse. Packets sent from one guest to another are routed between the corresponding backend drivers by the bridge.

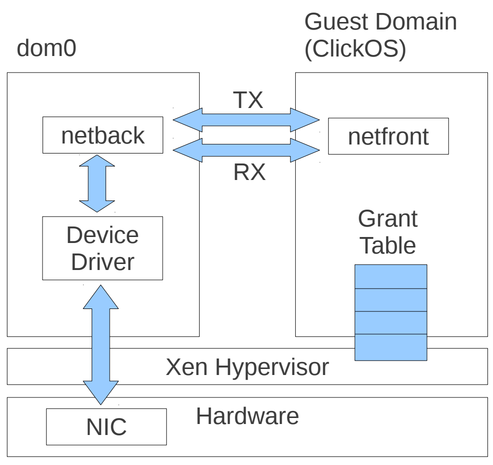


Figure 4.10: Xen netfront and netback infrastructure

Under Linux, the original Click relies on the `PollDevice`/`FromDevice` and `ToDevice` elements that use Unix sockets in order to interact with network interfaces.

However, as ClickOS network I/O is based on directly accessing the netfront API, new Click networking elements were implemented to talk to XENs netfront driver directly:

The `FromClickOS` element is responsible for receiving packets from the netfront driver. It implements a callback function which stores incoming netfront packets into a queue which will be processed the next time the element is scheduled (i.e. `FromClickOS::run_task()` is called).

```
// from mini-os/lwip-net.c
extern "C" {
struct netfront_dev* get_net_dev();
}

// NOTE: Global queue only works when using only
//       a single FromClickOS element instance.
// Solution would be to pass FromClickOS this ptr as
// additional param to netfront_rx_handler
// but this will have implications to mini-os/netfront..
static DEQueue<Packet*> netfront_packet_queue;

static void netfront_rx_handler(unsigned char* data, int len)
{
    ...
    Packet *p = Packet::make(_headroom, data, len, 0);
    netfront_packet_queue.push_back(p);
}
```

```
int FromClickOS::configure(Vector<String> &conf, ErrorHandler *errh)
{
    ...
    _dev = get_net_dev();

    if (netfront_add_rx_handler(_dev, netfront_rx_handler, "FromClickOS netfront
        handler") != 0) {
        printf("FromClickOS::configure() - netfront_add_rx_handler failed.\n");
        return -1;
    }
    ...
}

bool FromClickOS::run_task(Task *)
{
    _task.fast_reschedule();

    // Read and push() at most one packet.

    // bail out if no packets left
    if (netfront_packet_queue.size() == 0) return true;

    // process packet queue
    Packet* p = netfront_packet_queue.front(); // get from queue front
    netfront_packet_queue.pop_front();         // remove from queue

    // output packet
    if (p) {
        output(0).push(p);
    }

    _count++;

    return true;
}
```

The `ToClickOS` element sends packets to the netfront driver. It pulls waiting packets from its input port and then uses the `netfront_xmit` function to forward them to the backend driver inside dom0. From there the packet data will finally be send to the physical network interface.

```
bool ToClickOS::run_task(Task *)
{
    _task.fast_reschedule();
    Packet *p = _q;
    _q = 0;
    if (!p) {
        p = input(0).pull();
        _pulls++;
    }

    if (p) {
        // send packet data via netfront_xmit()
        netfront_xmit(_dev, (unsigned char*) p->data(), p->length());
        checked_output_push(0, p);
    }
    return true;
}
```

4.2.2.1 Netfront driver callbacks

The original miniOS netfront driver only supports one single (hard-coded) RX callback/handler function that gets called when packets arrive. Therefore the `netfront_add_rx_handler()` function was added, which allows for registering (further) callbacks from external code, keeping them in a list of `netfront_rx_nodes`.

The following part describes the modifications made to `clickos/src/mini-os/netfront.c`:

First we define a node structure for a linked list of callback functions. Each callback can be tagged with a string for debugging purposes:

```
#define NETFRONT_TAG_LEN 32

struct netfront_rx_node {
  void (*netif_rx_callback)(unsigned char* data, int len);
  char tag[NETFRONT_TAG_LEN];
  struct netfront_rx_node *next;
};
```

Then the original single callback pointer inside the main `netfront_dev` device structure was replaced by a list using the node structure above:

```
struct netfront_dev {
        ...
-    void (*netif_rx)(unsigned char* data, int len);
+    struct netfront_rx_node *netif_rx_list;
};
```

The actual functions to register a callback function pointer (i.e. adding it to the list) and to clean up the list were implemented as follows:

```
int netfront_add_rx_handler(
  struct netfront_dev *dev,
  void (*netif_rx_func)(unsigned char* data, int len),
  const char* tag)
{
  struct netfront_rx_node *node_next = NULL;
  struct netfront_rx_node *node_new  = NULL;

  /* init RX callback list head */
  if (!dev->netif_rx_list) {
    dev->netif_rx_list =
      (struct netfront_rx_node*) malloc(sizeof(struct netfront_rx_node));

    if (dev->netif_rx_list) {
      dev->netif_rx_list->netif_rx_callback = netif_rx_func;
      dev->netif_rx_list->next = NULL;
      memcpy(dev->netif_rx_list->tag, tag, NETFRONT_TAG_LEN);
      printk("netfront_add_rx_handler(): list head initialized ('%s').\n", tag);
      return 0;
    }
    printk("netfront_add_rx_handler(): Setting list head failed ('%s').\n", tag);
    return -1;
  }

  /* append new handler behind head -- head should not be touched
     and always remain as set in init_netfront
     because of NETIF_SELECT_RX / NULL special case *?
  */
  /* save pointer to list elements following the head */
  node_next = dev->netif_rx_list->next;

  node_new = (struct netfront_rx_node*)
      malloc(sizeof(struct netfront_rx_node));

  if (!node_new) return -1; /* malloc failed */
  node_new->netif_rx_callback = netif_rx_func;
  memcpy(node_new->tag, tag, NETFRONT_TAG_LEN);
  node_new->next = node_next; /* chain remaining list elements */

  dev->netif_rx_list->next = node_new;  /* append to head */

  printk("netfront_add_rx_handler(): Added handler '%s'.\n", tag);

  return 0;
}
```

```
void netfront_release_rx_handler_list(struct netfront_dev *dev)
{
    struct netfront_rx_node *node = dev->netif_rx_list;
    struct netfront_rx_node *node_next = NULL;

    while (node) {
        node_next = node->next;
        printk("Releasing netfront RX handler '%s'\n", node->tag);
        free(node);
        node = node_next;
    }
    dev->netif_rx_list = NULL;
}
```

The miniOS `network_rx` function was adapted so that it will iterate over all registered handlers and call their callback functions when a new packet arrives. However, it has to be noted that too many registered handlers can potentially impact packet processing performance under higher load conditions when these handlers perform significant processing.

4.2.3 ClickOS Control Plane (Xenbus)

As described earlier, Click provides mechanisms to install configurations (through the command line) and furthermore to interact with elements by means of read and write handlers. In addition, the original kernel-level Click actually writes the configuration after initialization to a `/proc` entry, but it is not possible to use kernel-level Click for ClickOS.

ClickOS must provide equivalent mechanisms, even though the commands will come from dom0 (In the CHANGE platform architecture Dom0 has the module hosts outward-facing interface and runs the module host daemon which talks to the platform controller).

The solution implemented in ClickOS uses the *XenStore* mechanism for control plane messaging. The XenStore is an information storage space shared between domains, similar to `procfs` in Linux. In this tree-based database each domain gets its own path in the store (e.g., `/local/domain/<domID>/clickos/config`) so it is possible to communicate configuration settings between the dom0 and domUs. Generally, the dom0 is able to see the full tree of entries for all domains, including itself. In contrast each domU is only able to see information relevant to itself.

ClickOS utilizes the *Xenbus* API to listen to read and write events on the XenStore: Xenbus is a Xen-internal mechanism to transfer data back and forth between the dom0 and the domU. This way ClickOS is able to dynamically reconfigure the current Click setup at run-time as soon as a new configuration is available.

In context of the CHANGE platform architecture, ClickOS configuration installation then works as follows:

- The Platform Controller tells the Module Host daemon to install a configuration

- The Module Host daemon writes the configuration to the relevant entry in the XenStore
- ClickOS receives a notification about the new data via Xenbus events and passes the new settings to the Click engine.

4.2.4 Future work

Current work concentrates on testing ClickOS packet processing performance with the help of packet generator tools like `netperf` and `PF_RING` and its scalability on the Xen hypervisor. Based on these results further improvements to the ClickOS packet handling code and scalability will be implemented. To do so, we will leverage the multi-queue hardware features of recent network interface cards; so manage how ClickOS instance get access to packets. For example, for very high packet processing performance (approaching 10G) may achieved by tying individual queues to individual instances of ClickOS.

It order to fully realise the module host target for ClickOS, we are also working on memory management to support methods to support the creating and migration of flow primitive instances. The goal is to support the management and live migration of flow processing instances, with minimal overhead and without unduly affecting their performance and timeliness. Finally we intend to test the performance of ClickOS realistic applications.

Achieving these goals opens up the possibility of exploiting ClickOS as a high performance and low footprint module host capable of realising the requirements of the CHANGE architecture.

4.3 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. As a complementary work for ClickOS (Section 4.2.4), and in order to go further in the CHANGE project vision, we rethink this concept inside our CHANGE platforms and handle each input traffic as a whole block instead of packets.

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. This section will give an overview of *FlowOS*, which 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 the socket interface.

Each input packet matching some criteria is placed in a specific structure which is shared between all processing modules that interact in a parallel manner with this flow. In fact, within a single processing pipeline, different processing modules can see the same flow as multiple independent flows (as in, “sub-flows”), without having to bother about even the existence of other bytes.

Thus, the potential performance benefits are twofold. First, it provides an easier parallelism through a better adapted programming model (remember that each processing module in our execution pipes have each a "window of flow data" within which they do what they want, 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).

As ClickOS, *FlowOS* proposes an API to users to manage flows (create, delete, modify, etc.), and to write processing modules to process these flows.

FlowOS is a kernel module that captures incoming IP packets from NIC upon configuration. It creates then a shared virtual queue for each flow and enqueues IP packets that belong to the flow. A flow maintains multiple virtual queues of data stream, one queue for each protocol (e.g., IP, TCP, UDP, etc.).

A Processing Module (PM) is also a kernel module that is attached to a flow for processing data in this flow. We assume that each PM can interact on one specific layer of a flow (e.g., an IP checksum processing module will interact with a flow only on its IP layer). For each data processing, we create a kernel thread of the corresponding PM that maintains a data window, defined by two pointers (head and tail). This thread goes to sleep when there is no data to process in its window.

A CHANGE platform administrator can attach multiple PMs to a single flow creating then a pipeline of PMs. She gives a certain order and inter-dependency between PMs in the pipeline that define which PMs wake up first and which PMs have to wait for other PMs to release their data. Thus, when input data matched a specific flow definition, one or more PMs, attached to this flow, wake up first, and these PMs release data to the next PMs in the pipeline. The last PM of the processing pipeline is responsible for forwarding traffic to the output NIC.

To simplify writing PMs for programers, FlowOS provides data structures, macros, and APIs that hide the complexity of the internal details. For instance, the same function used to pass data to the next PM can be used to push data back to the output NIC. It also provides an abstract pointer type to point a specific byte of data in a flow.

4.3.1 FlowOS Architecture

FlowOS defines a *classifier* on the input side which is compatible with OpenFlow [1] rules and is responsible for filtering and forwarding IP packets into appropriate flows. On the output side, FlowOS defines the *merger* which is responsible for reassembling IP packets and sending them to the output interfaces. It also defines a flow *controller* which is responsible for managing flows by creating a queue for each protocol on the flow; providing functionalities to insert, delete, or modify the definition of the flow; and dynamically attaching or detaching processing modules from a flow. It is also responsible for communicating with other entities in the network such as the source or the destination of a flow or a peer flow processing platform. Figure 4.11 shows a simplified view of FlowOS.

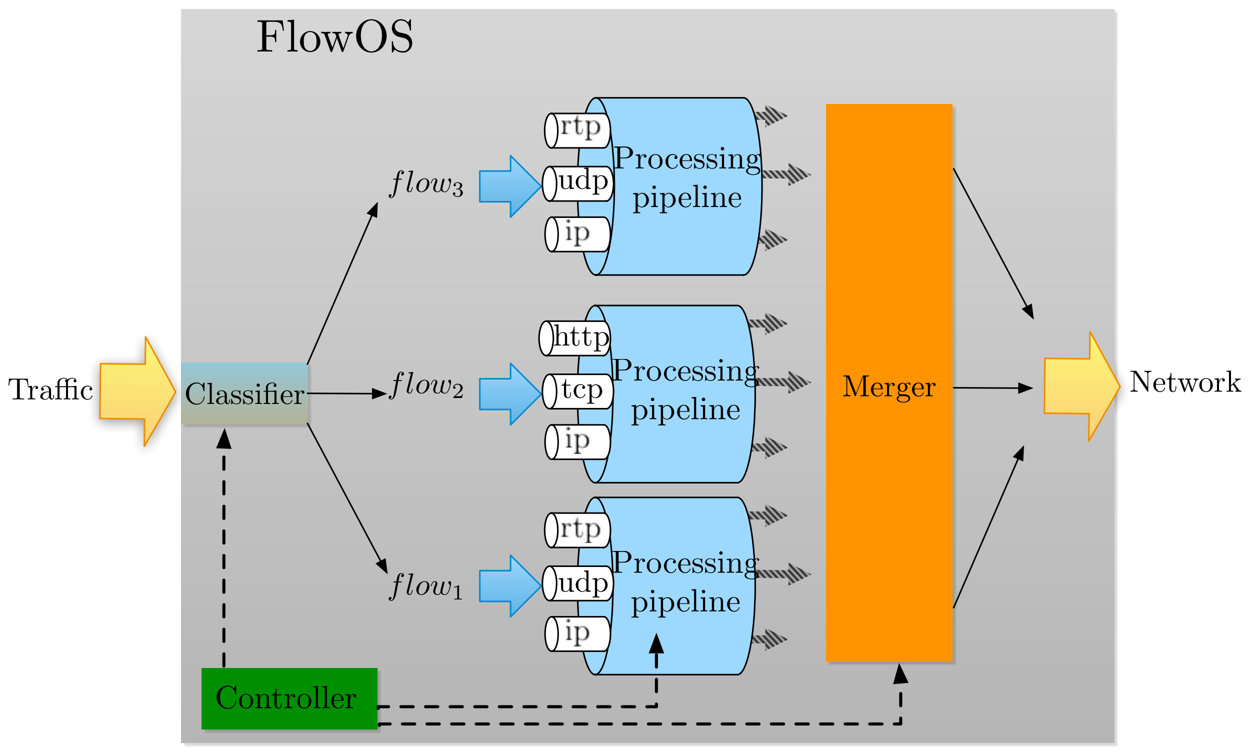


Figure 4.11: FlowOS Architecture.

FlowOS builds flows in the shared kernel space in order to avoid the overhead of copying IP packets from kernel to user space. Each flow contains one or more layers, where each layer represents a protocol. For the sake of clarity, we define a layer as a *stream* of data in the rest of this report. A stream then is illustrated as a sequence of virtual buffers defined by the *head* and *tail* pointers in the `sk_buff`. Each stream pointer (an abstract data type) consists of a pointer to the protocol data unit, a pointer to the payload buffer, and a character pointer to the actual data byte. The stream pointer along with associated APIs offer to the programmers a uniform view of flows as a stream of bytes. Figure 4.12 shows the internal structure of a flow.

As highlighted above, a Processing Module (PM), which is also a kernel module in FlowOS, can interact with a flow on one specific stream at a time. Since that more than one PM can process a flow at the same time and at the same stream, a PM can access a particular section (*window*) of a flow at a time. This window is delimited by *head* and *tail* pointers associated with that PM. Note that two or more PMs may process overlapping windows of a flow simultaneously. Therefore, the head and tail pointers management is crucial for safety and performance.

Suppose that module 2 is situated after module 1 on a pipeline, and module 2 depends on module 1 (i.e. module 2 needs module 1 outputs), then module 2 should not have access to data beyond module 1's head. Similarly, if either of these modules is in read/write mode (i.e. it will change the content of the flow), they cannot overlap. Figure 4.13 illustrates the head and tail pointers of three processing modules interacting with the same flow. Module1 interacts with this flow at the *http* stream, where it removes all occurrences of a

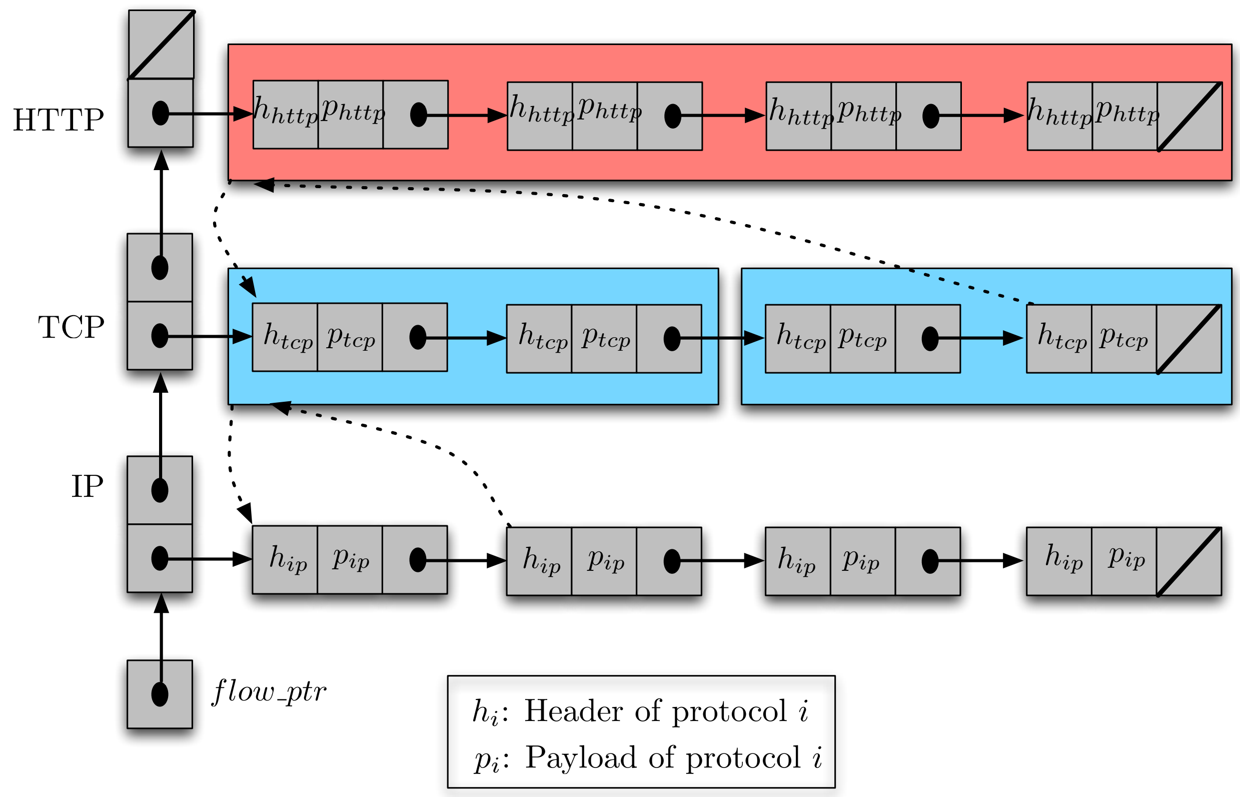


Figure 4.12: Internal structure of a flow.

given suite of bytes from the stream. Module2 and Module3 interact with the same flow at the *ip* stream, where Module2 counts the number of bytes, and Module3 counts the number of occurrences of a given suite of bytes. Given that Module1 is in write mode (i.e. it changes the content of the flow) in interval $[h_1, t_1]$, Module2's interval $[h_2, t_2]$ cannot overlap Module1's interval. Thus, the maximal value of t_2 is h_1. However, given that Module2 and Module3 are in read mode, they can overlap and have access to the same bytes without any risk.

The invariants are as follows:

(i) A PM's head and tail must be within the system head and tail.

(ii) The head of a PM cannot go beyond its tail.

(iii) The tail of a PM cannot go beyond the tail of the previous PM.

4.3.2 Data Structures

FlowOS defines a number of data structures, associated APIs, and macros to write flow processing modules.

4.3.2.1 FlowID

Each flow is defined with a number of parameters, for example source IP address, destination IP address, source port, destination port, etc. FlowOS borrowed the FlowID structure from the OpenFlow definition of flows.

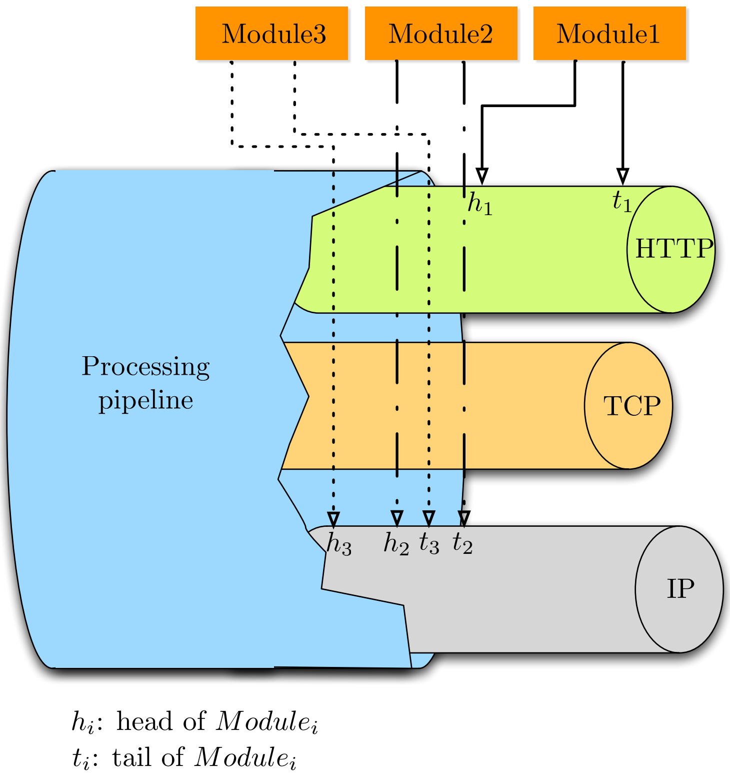


Figure 4.13: Processing modules attached to a flow. Traffic crosses the pipeline from right-to-left.

4.3.2.2 DataUnit

Each communication protocol encapsulates data in its own Protocol Data Unit (PDU) which has some header information and protocol payload. FlowOS constructs a virtual data stream for each protocol which is a sequence of PDUs of that protocol. FlowOS defines a generic structure to hold any PDU of a flow.

```
struct DataUnit {
  /* next packet of this protocol of a flow */
  struct hlist_node list;
  /* higher level PDU it belongs to */
  struct DataUnit *parent;
  /* lower level PDU it contains */
  struct DataUnit *children;
  /* number of children */
  u8 num_child;
  /* protocol of this level */
  uint32_t protocol;
  /* pointer to header of protocol */
  void *header;
  /* pointer to payload -- linked list of payload */
  struct list_head payload;
  /* status of this node */
  uint32_t status;
};
```

The “protocol” field tells which protocol data unit is pointed to by this DataUnit. A DataUnit in a flow is linked to other DataUnits of the same protocol (stream). It is also linked to the higher layer protocol DataUnit called “parent” and a number of the lower layer protocol DataUnits called “children”. The “num_child” tells that how many lower layer PDUs a higher layer PDU is fragmented into. Each DataUnit also maintains the “header” and “payload” fields where the “header” points the actual protocol header and the “payload” is a list of fragmented data buffers. Finally, the “status” field describes the current state of this DataUnit (PM_CLEAN, PM_DIRTY, etc.) for each processing module.

4.3.2.3 Payload

Payload is defined in flowOS as a virtual buffer containing payload of each packet contained in the same DataUnit (maybe fragmented into several lower layer DataUnits). Each Payload object maintains a pointer to the `sk_buff`, a char pointer to the beginning of the payload, and the number of payload bytes.

```
struct Payload{
  /* pointer next segment */
  struct list_head list;
  /* pointer to SK_BUFF */
  struct sk_buff *skb;
  /* beginning of data */
  u_char *buffer;
  /* data in this segment */
  uint16_t length;
};
```

4.3.2.4 Stream

A sequence of DataUnits of a specific protocol form a virtual stream for that protocol. It is a doubly linked list (not circular to allow concurrent head and tail processing) with head pointed to by the “head” member variable. Since a stream is not a circular linked list, the “tail” pointer holds the tail of the stream. The “protocol” field defines the protocol of a stream.

```
struct Stream{
 /* protocol list */
  struct list_head list;
  /* this protocol */
  uint32_t protocol;
  /* list head of protocol PDUs (DataUnit) */
  struct hlist_head head;
  /* last node of the queue */
  struct Pointer tail;
};
```

4.3.2.5 Flow

A Flow consists of a number of Streams connected together in a doubly linked-list pointed to by the "streams" field. A processing module works on a specific Stream of a flow. The list of processing threads attached to this flow are kept on a doubly linked-list pointed to by "threads". Each flow is given a human readable name and is defined by a set of flow parameters in FlowID. The "protocols" field maintains the list of protocols this flow encodes into streams.

```
struct Flow {
  /* next to put it in the list of flows */
  struct list_head list;
  /* a readable name */
  char name[25];
  /* what defines this flow */
  struct FlowID *id;
  /* number of PMs attached to this flow */
  u8 num_threads;
  /* list of modules processing this flow */
  struct list_head threads;
  /* number of protocols streams */
  u8 num_protos;
  /* protocols this flow considers */
  uint32_t protocols[MAX_LEVELS];
  /* list of protocols and corresponding queues */
  struct list_head streams;
};
```

4.3.2.6 Module

A wrapper for kernel module structure with the thread function, protocol, and type. When a new processing module is loaded into the system, a Module object is created and populated with its "type" (MODULE_RONLY or MODULE_RDWR), "protocol" stream that it processes, the thread function "process". The kernel module pointer is stored in "mod" and all the processing modules are listed in the module list of the system.

```
struct Module {
  /* to put in the list of PMs */
  struct list_head list;
  /* pointer to the kernel module */
  struct module *mod;
  /* read only/read-write */
  u8 type;
  /* protocol this module process */
  uint32_t protocol;
  /* thread function */
  int (*process)(void *);
};
```

4.3.2.7 Thread

When a processing module is attached to a flow, a new thread of the processing module is created which maintains the "head" and "tail" information of the "flow" being processed. The kernel thread pointer is stored in "thread" and all the processing threads attached to a flow are listed in threads list of the flow. A thread also keeps the Module pointer "mod" from which it is instantiated. The "order" or priority of a PM in the pipeline represents the dependency of PMs. Note that a pointer to the thread is passed to the thread function of a module when a thread is created.

```
struct Thread {
  /* next to put in a list of PM threads */
  struct list_head list;
  /* the flow this thread is attached to */
  struct Flow *flow;
  /* the kernel module */
  struct Module *mod;
  /* kernel thread associated with the kernel module */
  struct task_struct *thread;
  /* position on the processing pipeline */
  int order;
  /* head of the flow for this thread */
  struct Pointer head;
  /* tail of the flow for this thread */
  struct Pointer tail;
};
```

4.3.2.8 Pointer

A Pointer can point a single byte in the flow or can be used to iterate over DataUnits. The "unit" keeps the current DataUnit pointer, "payload" stores the current payload buffer, and "ptr" is the actual byte pointer in the payload buffer.

```
struct Pointer{
  /* current DataUnit */
  struct DataUnit *unit;
  /* current Payload buffer within DataUnit */
  struct Payload *payload;
  /* current byte pointer inside current payload buffer */
  u_char *ptr;
};
```

4.3.3 FlowOS APIs

FlowOS provides a number of module programming APIs to help programmers to write processing modules. In this section, we briefly describe the functionalities of these APIs.

CreateFlow(struct FlowID *id, char *name) Creates an empty flow based on the parameters found in flow ID. An optional flow name can be specified using the "name" parameter. If the name is NULL, the system gives a name like `flow<n>` to the flow. The function puts the newly created flow on the flow table and returns a Flow pointer. It returns NULL on failure.

DeleteFlow(struct Flow *flow) Deletes the specified flow. If the flow is not empty, it removes all the processing modules attached to this flow, then deletes all the streams in it, and finally removes the flow from the flow table.

DeleteFlowByName(char *name) Retrieves the flow pointer in the flow table by using the "name" and then invokes the DeleteFlow with the flow pointer.

FindFlowByName(char *name) Retrieves the flow named "name" from the flow table. It returns a Flow pointer on success otherwise it returns a NULL.

CreateStream(struct Flow *flow, uint32_t protocol) Creates a new stream of protocol "protocol" for the flow "flow" and returns a Stream pointer. It returns NULL upon failure.

DeleteStream(struct Stream *st) Removes the stream "st" from a flow. It just deletes DataUnits from the stream and then remove the stream from the flow. Note that it does not handle PMs attached to a flow. Therefore, before calling this function users should remove any PM attached to stream "st".

GetStream(struct Flow *flow, uint32_t protocol) Retrieves the stream pointer for "protocol" from "flow". It returns NULL if the stream does not exist.

GetStreamHead(struct Flow *flow, uint32_t protocol) Returns a Pointer to the head of the stream for "protocol" in "flow". If stream is empty or does not exist, the Pointer value is null. Note that IsNull() API is used to check if a Pointer is null or not.

GetStreamTail(struct Flow *flow, uint32_t protocol) Returns a Pointer to the tail of the stream for "protocol" in "flow". If stream is empty or does not exist, the Pointer value is null.

FindModule(struct module *mod) Searches for the kernel module "module" in FlowOS's PM list. It returns a Module pointer if the module is in the list or NULL otherwise.

FindModuleByName(char *module) Searches for the processing module with name "module" and returns a Module pointer if the module is in the list or NULL otherwise.

LoadModule(char *name) Tries to load a PM into FlowOS. First, it locates the kernel module and loads into memory, then probes for necessary functions in it, and finally puts into the FlowOS PM list. It returns the Module pointer upon success otherwise it returns NULL upon failure.

UnloadModule(struct Module *mod) Removes the module from FlowOS module list, unloads associated kernel module from memory, and frees memory.

AttachModule(char *flow, char *module, int order) Attaches the processing module "module" to the flow "flow" at position "order". It creates a new thread of the module for this flow, puts it in the processing pipeline, and starts the thread. It returns 0 on success and -1 upon failure.

DetachModule(char *flow, char *module) Removes the module from the processing pipeline of the flow. If the module currently processing data it finishes processing and releases data to the next PM. If the module is not used by any other flows, it unloads the module by calling UnloadModule API.

IsNull(struct Pointer p) Returns 1 if "p" is a null stream pointer or 0 otherwise.

IsEqual(struct Pointer p1, struct Pointer p2) Returns 1 if "p1" and "p2" point the same byte in a stream or 0 otherwise.

IncPointer(struct Pointer p) Moves pointer "p" to right by one byte and returns the new pointer.

MovePointerNext(struct Pointer p) Moves pointer "p" to the next DataUnit and returns the new pointer.

MovePointerUp(struct Pointer p) Maps pointer "p" to corresponding position at the immediate higher level protocol stream and returns the new pointer.

MovePointerDown(struct Pointer p) Maps pointer "p" to corresponding position at the immediate lower level protocol stream and returns the new pointer.

MapPointerUp(struct Pointer p, uint32_t protocol) Maps pointer "p" to corresponding position at higher "protocol" level stream and returns the new pointer.

MapPointerDown(struct Pointer p, uint32_t protocol) Maps pointer "p" to corresponding position at lower "protocol" level stream and returns the new pointer.

MapPointer(struct Pointer p, uint32_t protocol) Maps pointer "p" to corresponding position at "protocol" level stream and returns the new pointer.

PacketToFlow(struct sk_buff *skb) Matches received IP packets against currently defined flows. If the packet matches a flow it returns the Flow pointer otherwise it returns NULL.

InsertIntoFlow(struct Flow *flow, struct sk_buff *skb) Goes through the list of protocols defined for this flow sequentially and invokes that protocol's decoder to construct a DataUnit for that protocol and insert it into the corresponding stream. It returns a pointer to DataUnit on success and NULL otherwise.

DeleteFromFlow(struct Flow *flow, struct DataUnit *unit) Removes a DataUnit “unit” from the “flow”. It is necessary to pass `sk_buff` level DataUnit to this function. It deletes corresponding DataUnits from every layer of the “flow”.

DeleteDataUnit(struct Thread *thread, struct DataUnit *unit) Removes a DataUnit from the flow being processed by the “thread”. DataUnit can be specified from any level. It updates the head and tail pointers of “thread” if necessary.

InsertIntoStream(struct Stream *stream, struct DataUnit *unit) Inserts a DataUnit returned by a protocol decoder into a Stream. It updates stream head and tail pointers if necessary.

ReleaseData(struct Thread *thread, struct Pointer head, struct Pointer tail) Is used by the “thread” to release data between the “head” and “tail” pointers to the next PM (could be the controller) and wakes the thread up. If the next thread is a PM, it maps head and tail pointers to corresponding head and tail pointers of the next PMs level. If the “thread” is the last thread of the processing pipeline, it hands data to the system controller. It updates its head pointer and the tail pointer of the next thread.

Protocol_Decoder(struct Flow *flow, struct DataUnit *unit) Constructs a Protocol DataUnit from lower level protocol DataUnit “unit”. It handles fragmentation reassembly and builds complete PDUs from multiple lower level DataUnits. It invokes InsertIntoStream to put current DataUnit into appropriate stream of the flow. It returns a DataUnit pointer on success and returns NULL upon failure.

RegisterDecoder(DECODER decoder, uint32_t protocol) Dynamically registers a new “protocol” decoder with the system.

UnregisterDecoder(uint32_t protocol) Removes the “protocol” decoder from the system.

GetDecoder(uint32_t protocol) Retrieves the “protocol” decoder from the list of registered decoders. It returns a function pointer to “protocol” decoder otherwise returns a NULL.

4.3.4 Writing Processing Modules

A flow processing module is a kernel module which runs as a kernel thread. Besides module init and exit functions, a PM must provide the following functions:

(i) `int <modname>_process(void *)` It is the main processing function that works as a kernel thread. It takes one generic pointer as the parameter which is used to pass data to be processed.

(ii) `uint32_t <modname>_protocol()` It returns the protocol that this module processes.

(iii) `int <modname>_type()` It returns the type of the module (`MODULE_RONLY` or `MODULE_RDWR`).

In the main function `<modname>_process()`, the parameter `(void *data)` is converted to a `Thread *thread` as:

```
struct Thread *thread = (struct Thread *) data;
```

Then, all the processing is done between the following macros:

```
BEGIN_PROCESS(thread, sleep);

/* this PM process data */

END_PROCESS;
```

The BEGIN_PROCESS/END_PROCESS macros are defined in flowos.h which implements the infinite loop of the thread and make it sleep if there is no data to process. BEGIN_PROCESS macro also handles the thread kill signal. The second parameter “sleep” is used to make the PM sleep even if there are data to process. Note that the thread sleeps at the beginning, so when it wakes up, it processes the entire code between BEGIN_PROCESS and END_PROCESS.

Note also that Thread has two pointers “head” and “tail” which represent the window of the flow being processed by this PM. One can iterate over the DataUnits between “head” and “tail” using the macro:

```
for_each_dataunit(thread, unit, next)
```

where “unit” is a DataUnit pointer that holds each DataUnit and “next” is a temporary DataUnit pointer and should not be used inside the loop.

A PM has to pass data to the next PM in the pipeline when it is ready for the next PM. Note that a PM can release data as soon as it starts processing or it has to wait until it has finished the processing. A PM should use the FlowOS API ReleaseDataByUnit() which takes the “thread” window and passes every DataUnit between “head” and “tail” pointers. A PM can also release any amount of data to the next PM by using ReleaseData() API. A PM can delete/drop a DataUnit by using DeleteDataUnit() API.

5 Conclusions

In this deliverable we have reviewed our work on building the CHANGE flow-processing platform, which is key to enabling the vision of the CHANGE project. The platform has many components including the platform controller, the processing modules, and the API that any CHANGE platform must implement (our current platform prototype does).

We have pushed through all these components, moving closer to creating an efficient, fast processing platform.

The work presented here is at different stages of maturity, which is unavoidable for this stage in the project.

The highlights of our work so far include *netmap* , a framework that gives userspace applications a fast channel to exchange raw packets with the network adapter. As shown by our measurements, *netmap* can give huge performance improvements to a wide range of applications. Thanks to a libpcap compatibility library, existing applications may be able to run much faster even with no source or binary modifications. *netmap* is ideal to build packet capture and generation tools, but also useful for for all packet processing tasks, including high performance software routers or switches, firewalls. Although it requires device driver modifications, *netmap* is not bound to any specific piece of hardware, and its design makes very reasonable assumptions on the capabilities of the hardware. The *netmap* code has been released publicly and can be downloaded from *http://info.iet.unipi.it/ luigi/netmap/*

Another highlight is our work on offloading routing to the Openflow switch. Internet traffic follows Zipf's law: a small fraction of flows capture most of the traffic. We have propose to exploit this property to offload most of the traffic from a complex controller to a simple forwarder. We have showed that the main challenge lies in achieving a high traffic offloading gain while limiting the churn, and propose a heavy hitter selection strategy called Traffic-aware Flow Offloading (TFO). We perform simulations of TFO on real traffic traces. The results highlight the effectiveness of the strategy used in TFO and suggest that traffic offloading should be considered to build feasible alternatives to current router designs.

We have also started work on two different but promising approaches to support network processing. ClickOS is a minimalistic OS running in Xen that gives great isolation properties for very little overhead. FlowOS is another novel OS that we have started to build from ground up with the purpose of supporting support flow processing.

Finally, our work on efficient state migration holds promise if we can apply it many types of network processing and we can embed this support in the operating system.

Bibliography

[1]

[2] The click modular router project.

[3] Openflow switch specification, version 1.0.0.

[4] Openflow switch specification, version 1.1.0.

[5] A tour of the mini-os kernel.

[6] The dag project. Technical report, University of Waikato, 2001.

[7] S. Agarwal, C. Chuah, S. Bhattacharyya, and C. Diot. The Impact of BGP Dynamics on Intra-Domain Traffic. In *ACM SIGMETRICS*, 2004.

[8] AdvancedTCA Specifications for Next Generation Telecommunications Equipment. www.advancedtca.org.

[9] 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.

[10] Kimberly C. Claffy and Nevil Brownlee. Understanding Internet traffic streams: Dragonflies and Tortoises. *IEEE Comm. Mag.*, 2002.

[11] Edith Cohen, Nick Duffield, Haim Kaplan, Carsten Lund, and Mikkel Thorup. Algorithms and estimators for accurate summarization of internet traffic. In *ACM IMC*, 2007.

[12] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB*, 2003.

[13] L. Deri. Improving passive packet capture:beyond device polling. In *SANE 2004, Amsterdam.*

[14] L. Deri. ncap: Wire-speed packet capture and transmission. In *Workshop on End-to-End Monitoring Techniques and Services*, pages 47–55. IEEE, 2005.

[15] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *ACM SOSP*, 2009.

[16] 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.

[17] Cristian Estan, Ken Keys, David Moore, and George Varghese. Building a better netflow. In *ACM SIGCOMM*, 2004.

[18] Wenjia Fang and Larry Peterson. Inter-as traffic patterns and their implications. In *IEEE Global Internet*, 1999.

[19] Alexander Fiveg. Ringmap capturing stack for high performance packt capturing in freebsd. *http://ringmap.googlecode.com/ files/ringmap_slides.pdf*, 2010.

[20] 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.

[21] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT Behavioral Requirements for TCP. RFC 5382 (Best Current Practice), October 2008.

[22] 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.

[23] M. Handley, O. Hodson, and E. Kohler. Xorp: An open platform for network research. *ACM SIGCOMM Computer Communication Review*, 33(1):53–57, 2003.

[24] A. Heyde and L. Stewart. Using the endace dag 3.7 gf card with freebsd 7.0. 2008.

[25] Z. Xiao J. Rexford, J. Wang and Y. Zhang. BGP routing stability of popular destinations. In *ACM IMC*, 2002.

[26] R. Jain. Characteristics of destination address locality in computer networks: A comparison of caching schemes. *Computer Networks and ISDN*, 1989/90.

[27] K. Mitsuya K. Cho and A. Kato. Traffic data repository at the wide project. In *USENIX ATC, Freenix track*, 2000.

[28] Christopher Clark Keir, Christopher Clark, Keir Fraser, Steven H, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI*, pages 273–286, 2005.

[29] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting route caching: The world should be flat. In *PAM*, 2009.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[31] 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.

[32] Max Krasnyansky. Uio-ixgbe. *https://opensource.qualcomm.com/wiki/UIO-IXGBE*.

[33] Nate Kushman, Srikanth Kandula, and Dina Katabi. Can you hear me now?!: it must be BGP. *ACM CCR*, 2007.

[34] C. Labovitz, G. Malan, and F. Jahanian. Internet Routing Instability. In *ACM SIGCOMM*, September 1997.

[35] Ramon Lawrence. A survey of process migration mechanisms.

[36] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Proc. of IEEE MSE*, 2007.

[37] Tudor Marian. Operating systems abstractions for software packet processing in datacenters. *PhD Dissertation, Cornell University*, 2010.

[38] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Winter Conference*. USENIX Association, 1993.

[39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.

[40] J.C. Mogul and K.K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems (TOCS)*, 15(3):217–252, 1997.

[41] OECD. Broadband Portal. www.oecd.org/sti/ict/broadband, 2008.

[42] K. Papagiannaki, N. Taft, and C. Diot. Impact of flow dynamics on traffic engineering design principles. In *IEEE INFOCOM*, 2004.

[43] Larry Peterson. Inter-as traffic patterns and their implications. In *in Proc. IEEE GLOBECOM*, 1999.

[44] Luigi Rizzo. Polling versus interrupts in network device drivers. *BSDConEurope 2001*, 2001.

[45] Luigi Rizzo, Marta Carbone, and Gaetano Catalli. Transparent acceleration of software packet forwarding using netmap. *http://info.iet.unipi.it/∼luigi/netmap/*.

[46] Nadi Sarrar, Anja Feldmann, Steve Uhlig, Rob Sherwood, and Xin Huang. Towards hardware accelerated software routers. In *ACM CoNEXT Student Workshop*, 2010.

[47] Rob Sherwood, Glen Gibb, Kok-Kiong Yapa, Martin Cassado, Guido Appenzeller, Nick McKeown, and Guru Parulkar. Can the production network be the test-bed? In *OSDI*, 2010.

[48] W. Shyu, C. Wu, and T. Hou. Efficiency analyses on routing cache replacement algorithms. In *IEEE ICC*, 2002.

[49] W.R. Stevens and G.R. Wright. *TCP/IP illustrated (vol. 2): the implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[50] J. Wallerich, H. Dreger, A. Feldmann, B. Krishnamurthy, and W. Willinger. A methodology for studying persistency aspects of Internet flows. *ACM CCR*, 2005.

[51] Jrg Wallerich and Anja Feldmann. Capturing the variability of internet flows across time. In *IEEE Global Internet*, 2006.

[52] L. Wang, X. Zhao, D. Pei, R. Bush, D. Massey, A. Mankin, S. Wu, and L. Zhang. Observation and Analysis of BGP Behavior under Stress. In *ACM IMW*, 2002.

[53] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[54] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. In *In ACM SIGCOMM*, pages 309–322, 2002.

[55] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *ACM IMC*, 2004.

[56] Zhiruo Cao Zheng, Zheng Wang, and Ellen Zegura. Performance of hashing-based schemes for internet load balancing, 1999.