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

D5.3: Application Development and Deployment

Due date of deliverable: 31 December 2013

Actual submission date: December 23, 2013

Start date of project	1 October 2010
Duration	36 months
Lead contractor for this deliverable	University Politehnica of Bucharest
Version	Final, December 23, 2013
Confidentiality status	Public

Abstract

This deliverable reports on the final updates to the CHANGE architecture and platform, as well the applications developed to run over them. Regarding the platform itself, we present two complementary technologies. First, we provide an update on the VALE high-performance switch which is used as a platform back-end to demux packets from NICs to the virtual machines carrying out the network processing. Second, we introduce new ClickOS results that show that it is able to process packets at 10Gb/s and higher while running middlebox processing. On the architecture side, we present three results: (1) Symnet, a static checker that verifies that CHANGE network configurations are safe to run, both at an architectural and platform levels; (2) tracebox, a tool that allows network operators to detect which middleboxes modify packets on any network path; and (3) CHANGE's inter-platform signaling framework. Finally, we discuss a number of applications developed for the CHANGE platform, including carrier-grade NATs, load balancers, IDSes and firewalls, to name a few. The appendix to this document lists information about open source releases of these technologies.

Target Audience

Experts in the field of computer networking, the European Commission.

Impressum

Full project title	CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions
Title of the workpackage	WP5: Deployment and Applications
Editor	Costin Raiciu, University Politehnica of Bucharest (PUB)
Project Co-ordinator	Adam Kapovits, Eurescom
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2013 Participants in project CHANGE

List of Authors

Authors	Felipe Huici, Michio Honda, Joao Martins, Mohamed Ahmed (NEC, editor), Olivier Bonaventure (UCL-BE), Georgios Smaragdakis (TUB), Costin Raiciu, Radu Stoenescu (PUB), Luigi Rizzo (UNIPI), Francesco Salvestrini, Gino Carrozzo, Nicola Ciulli (NXW), Vito Piserchia (DREAM).
Participants	NEC, ULANC, PUB, TUB, NXW, DREAM
Work-package	WP5: Deployment and Applications
Security	Public (PU)
Nature	R
Version	1.0
Total number of pages	126

Contents

List of Figures

List of Tables

1 Introduction

Software-based middlebox processing has been gaining considerable momentum in the past years. Recently, the concept of Network Function Virtualization has seen a lot of traction among major industry partners seeking to shift away from hardware based-middleboxes to virtualized, software-based ones running on commodity hardware. The reasons behind this push are several: conventional middleboxes are typical proprietary, expensive, and hard to upgrade, to name a few.

However, a simple shift to software-based processing is not a magical panacea. Part of the reason hardware boxes are expensive is that they provide excellent performance, a requirement for operator deployments. One of the big open questions in NFV then is: can software-based, virtualized middleboxes provide significant performance, or is the concept of NFV a pipe dream?

In this deliverable we present performance results from the CHANGE platform. These results show that the premise of NFV is feasible on current commodity hardware, as long as we are careful about how we configure the servers and design the software running on them.

More specifically, this document describes three complementary technologies developed in the CHANGE project that enable not only NFV but also the CHANGE architecture. We begin by presenting new results on mSwitch (section 2.2), an extended version of the VALE software switch. mSwitch is able to switch packets at rates of up to 283 Gbit/s while yielding low delay (5 microseconds). These properties make it ideal as the platform's switching back-end.

Second, we report on recent developments on ClickOS (section 2.3), showing that it is possible to run virtualized, middlebox processing on commodity hardware at rates of 10 Gbit/s and higher (using mSwitch as the switching back end running in the Xen control domain or dom0).

Third, we introduce recent results from Symnet (section 3.1), a static checker able to verify CHANGE platform network processing configurations and determine whether running them would result in incorrect networking (e.g., looping) or security problems. Unlike other checkers in the literature, Symnet is able to verify *stateful* middleboxes, and can check fairly large networks with hundreds of nodes in seconds. The CHANGE platform uses Symnet to decide whether to install a particular configuration or not, crucial to any entity running such a platform in their network.

At the CHANGE architecture level, Symnet can also be used to verify configurations involving several CHANGE platforms. In addition, this deliverable includes results from a novel tool developed within the project called tracebox (section 3.2). Tracebox is able to detect middleboxes that modify packets along network paths, and so can be used to detect problematic paths when instantiating CHANGE configurations. Further, we provide an update on the CHANGE platform's signaling framework (section 3.3).

In terms of applications, section 4 describes a number of middlebox applications developed within the CHANGE project, including firewalls, load balancers, carrier-grade NATs and IDSes. For some of these we present evaluation results of running them on ClickOS (section 4.2). These are encouraging: in many

cases we are able to process packets at rates of 2 to 6 millions packets per second on a single CPU core, high enough to show feasibility of NFV-type processing on commodity hardware. We also include implementations in stand-alone Click (section 4.3) which could be easily ported to ClickOS if desired. Further, we present results from work seeking to improve content delivery by having CDN providers and ISPs collaborate (section 4.4). The CHANGE work is relevant to this because the deployment of third-party software (i.e., the content caches) into an ISP's network can only be done if we can provide some level of guarantee as to how the introduced network processing will affect the operational network; the Symnet tool specifically, and the CHANGE platform more generally, are positive steps in this direction

Finally, this deliverable includes an appendix listing information about open source releases for these technologies.

2 The CHANGE Platform

2.1 Overview

In this section we present results from the final version of the CHANGE platform. In particular, we focus on two technologies. First, mSwitch, a high speed software switch that acts as the platform's back end, taking care of demuxing packets between the platform's NICs and the virtual machines doing the actual network processing (section 2.2).

Second, we describe, in great detail, the final version of ClickOS, which we use to implement the platform's virtualized, high performance data plane (section 2.3). It is worth pointing out that we do not revisit the Flowstream control software in this deliverable. While we make use of it, it is largely unchanged with respect to previous deliverables that reported on it, so it did not make much sense to mention it here. We do, however, cover updates to the platform's signaling framework in the next chapter.

In all, the results from both mSwitch (switching capacity in hundreds of Gigabits per second) and ClickOS (virtual machines able to process millions of packets per second on a single CPU core) are encouraging and show the feasibility of Network Function Virtualization on current, commodity hardware.

2.2 mSwitch: Software Switch Backend

2.2.1 Introduction

Software packet switching has been part of operating systems for almost 40 years, operating at different levels of the network protocol stack. Initially, software switching was mostly a prototyping tool or a low performance alternative to hardware-based devices. But several phenomena have changed the landscape in recent years: the widespread use of virtualization, which makes software switches necessary to interconnect virtual machines to physical interfaces; the growing interest in software-defined networking, which makes forwarding decisions more and more complex; and last but not least the increasing use of filters or middleboxes that require traffic analysis and manipulation before it reaches intended consumers.

Clearly, software switches play an important role in today's network systems. Mostly because of their evolution, however, those currently available in modern operating systems or virtualization solutions do not deliver both the performance and flexibility required by modern applications. Many OSes only offer basic Layer-3 (routing) or Layer-2 forwarding, so have limited flexibility and often not even reasonable performance. As we will show in section 2.2.2, many recent proposals have addressed the performance issue, but for specific tasks (L2 or L3 forwarding); or they offer relatively flexible solutions (Openflow, Open vSwitch) but without sufficient performance.

Ideally, we would like to decouple the switching *logic*, i.e., the mechanism to decide where packets should go, from the switching *fabric*, the underlying system in charge of quickly delivering packets from source to destination ports.

In this paper we introduce mSwitch, a software packet switching platform which is extensible, feature-rich

and can perform at high speeds while supporting a large number of ports. Our system redesigns the mSwitch switch [102] providing the following main contributions:

- We show how the system can be made to scale to hundreds of virtual ports, and make use of multiple cores to speed up forwarding.
- We enhance the scalability of the system, achieving an almost four-fold speedup over mSwitch. mSwitch reaches forwarding speeds of 72 Mpps (small packets) or 283 Gbit/s (large segments).
- We fully decouple the switching fabric from the switching logic, designing an API that allows the development of extremely compact forwarding modules that can be loaded at runtime, without sacrificing performance.
- We validate the generality of this API by implementing or porting three different types of mSwitch forwarding modules: a learning bridge consisting of 45 lines of code which outperforms the FreeBSD one by 6 times; an Open vSwitch module comprising small code changes and resulting in a 2.6 times boost; and a module used to support user-level network stacks.

In the rest of the paper, section 2.2.2 provides background material and covers related work; Section 2.2.3 gives a high level description of mSwitch, whose internal design is discussed in Section 2.2.4. A detailed performance evaluation is presented in Section 2.2.5, while Section 2.2.6 shows some use cases, implementing different types of switching functions as mSwitch modules.
mSwitch is open source, available for FreeBSD and Linux, and can be found at www.anonymizedurl.com.

2.2.2 Background and Related Work

In this section we provide a brief history of software switching and background regarding the basic operations that take place when switching packets on commodity hardware platforms.

2.2.2.1 Background

Early implementations of software packet switches/routers were mostly focused on the interconnection of physical ports, and on implementing the required functionality without much concern for performance or flexibility. The implicit (and reasonable) assumption was that truly high performance operation required custom hardware to supply the necessary I/O capacity and port fanout.
As for flexibility of operation, it was similarly assumed that forwarding decisions were based on a small and relatively immutable set of algorithms (longest prefix match for routing; table-based lookups for Ethernet switching or MPLS) that could be hard-wired in the system.
As a result, the switches/routers that one can find in legacy operating systems build on standard OS abstractions regarding device drivers, packet representation, and layering of functions.
Performance is conditioned by at least three factors: the access to physical I/O devices; the ability to optimize the packet processing code for packet forwarding rather than generic protocol processing; and the compu-

tational cost of the packet processing functions (flow lookup, destination selections, etc.). We will address these factors separately before discussing full systems that implement software switches.

2.2.2.2 Device I/O and Packet Representation

Physical port access has often been (and still is in many cases) a performance bottleneck due to hardware limitations. Many NICs are unable to handle small packets at line rate due to insufficient speed on the part of the on-board logic (a problem that affects a large number of 1- and 10 Gbit/s NICs), and/or limited bus bandwidth (PCI was unable to handle 1 Gbit/s ports); depending on the configuration, PCIe can be borderline for 10 Gbit/s and 40 Gbit/s ports).

Packet representation is another stumbling block. Operating systems typically store packets as a list of buffers, potentially shared by multiple consumers, and supplemented by a large amount of meta data (references to interface, sockets, pointers to protocol-specific headers, flags to support hardware offloading, and so on). Just initializing or managing these fields can take way longer than the actual transmission time of small packets, preventing line rate operation.

To address these problem, many recent proposals have moved to custom packet representations and device drivers. There are two dominant approaches. One puts the NIC under complete control of application programs, including custom device drivers sometimes running in userspace; Intel's DPDK [79] and Deri's PF_RING-DNA [66] are the two most representative examples of this kind. A second approach leaves the device driver within the kernel, but modifies it to support leaner and more efficient packet representations. Packetshader [70] used this technique to move traffic quickly to/from GPUs. The netmap framework [138] takes this further, providing an extremely simple packet representation that is also suitable for batch processing.

2.2.2.3 Moving Packets Around

Moving packets between ports (physical or virtual) of a switch is simple, at least in principle. Once a packet's destination is known, one just has to push the packet onto the relevant queue, and trigger subsequent processing in the NIC or in another process. However, these apparently simple operations are made expensive by several factors: the cost of the "trigger" action (writes to a NIC register may need to be preceded by a write barrier, with a cost in the order of 100 ns; wakeup signals for other processes may be even more expensive); contention between multiple senders to the same destination (generally requiring some form of locking); and the lifetime of the buffers used to store packets.

These problems are amortized by processing packets in batches, as done in many recent works [138, 135] but that is not a mechanism commonly found in OSes, which tend to rely on per-packet semantics. Queue contention can also be eliminated if each source port owns a private queue on each destination port. This approach, followed as an example in [54], is made possible by multi-queue NICs which can combine traffic from the various queues in hardware.

2.2.2.4 Generic Packet Processing

More generic packet matching and processing functions can be found within firewalls and packet capture systems. The Berkeley Packet Filter, or BPF, translates high-level packet matching operations into microinstructions that are either interpreted, or compiled into native code; compilation into native code is also used in DPF [59].

The BPF or DPF instruction set is not sufficiently expressive for efficient, general purpose processing. Firewalls, as an example, often implement data structures such as hash tables, lookup tables, radix trees and so on to support advanced filtering and matching functions. There are almost no reports on actual performance of software firewalls, but some data can be found in [40, 41], and also in [126].

Note that these numbers are highly conditioned by other factors mentioned before (locking, poor locality, slow I/O). The same code (ipfw) running on top of netmap has been shown to handle much higher packet rates [65].

Generic packet matching functions are also implemented by OpenFlow, which provides a standard set of match fields and rulesets similar to those of typical firewalls. In order to reduce per-packet costs, software implementations of OpenFlow (Open vSwitch) cache results of previous decisions into an exact-match cache (flow table) which is implemented as a hash table. The relatively large (44 bytes for IPv4) size of the match fields means that the hash computation and lookup is slower than for a simple Ethernet switch [141].

The use of exact match entries can potentially cause a flow explosion so recent Open vSwitch implementations introduce the ability to cache wildcarded match entries – called megaflows. This reduces the number of entries in a table, but the implementation requires looking up entries in multiple hash tables to find the best match.

2.2.2.5 Software Switches

Native bridging (and routing) in Linux, BSD and Windows is not particularly fast as it inherits all the performance problems detailed so far. We measured throughput in the order of 1-2 Mpps in the best case (see Figure 2.1), which can give excellent throughput with large segments (1500-byte frames, or possibly 64 KB TSO frames) but is largely insufficient for small frames on a 10 Gbps interface. Similar packet rates are achieved by proprietary solutions such as vSphere [161], which mostly focus on optimizing bulk TCP transfers.

It has also been shown in [141] that the forwarding performance of Open vSwitch can be greatly improved by replacing the I/O routines with more performant ones; we achieve similar results in this paper.

The Click modular router [114] has often been used to build custom packet processing systems. In-kernel Click can use modified device drivers with slightly better performance than the native OS drivers. Recent versions of Click also support the netmap API, boosting performance even further. Despite its flexibility, it is not trivial to use Click to create fast virtual ports that can be used for the interconnection of virtual machines, one of mSwitch's main use cases.

RouteBricks [54] uses Click to achieve flexibility. Performance is achieved by careful parallelization of

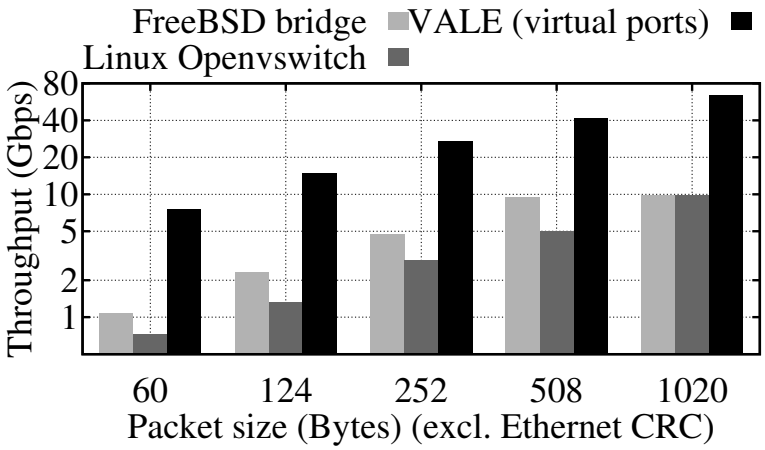


Figure 2.1: Packet forwarding throughput for the major software switches. Numbers are for forwarding between two 10 gigabit NICs except for mSwitch, which only supports virtual ports.

the data path in a cluster of multi-core servers. Within each server, multi-queue NICs are used to create contention-free paths between each pair of ports.

Hyper-Switch [135] is a recent development that improves the throughput and latency of Open vSwitch in virtualization environments by placing it directly in the hypervisor. It also implements a number optimizations aimed at reducing expensive context switches between the guests and the hypervisor.

Finally, CuckooSwitch [178] uses DPDK [79] to build a high performance switch to interconnect physical ports with large forwarding tables. While the system is tailored to a specific use case (Ethernet switching between physical ports), the solutions used to implement a high performance, concurrent hash table can be effectively used to implement one of the forwarding modules in mSwitch.

2.2.3 Our Contribution

The discussion in the previous section shows that existing systems do not simultaneously fulfill the demand for performance and flexibility of operation that today's virtualization and SDN architectures pose. To quantify their performance limitations, we carried out a simple experiment that forwards packets between two 10 Gbit/s ports using different major software switches (Figure 2.1). We see that the native FreeBSD switch cannot saturate a 10 Gbit/s link at short packet sizes even though it uses a hardwired forwarding function (the Linux bridge has similar performance). Open vSwitch loses another 20-30% in performance in exchange for more flexible operation. The mSwitch switch achieves much higher packet rates and throughput, but has a fixed forwarding function and supports virtual ports only.

Nevertheless, we have seen how recent research has produced useful insights into how to address the various bottlenecks that hinder performance. We adopted some of these design ideas and software components to build mSwitch, our software switch that delivers extremely high performance while being highly configurable and thus adaptable to different use cases.

We base our design on the netmap API and the mSwitch software switch, as these components provide a reasonable starting point in terms of performance, are already used in actual systems, and are also well decoupled

from specific operating systems or device constraints. mSwitch integrates and extends these components in several ways:

- Significant scalability enhancements, including (1) support for large number of virtual ports which may be needed when running many VMs on a single host and (2) support for multiple, per-port packet ring buffers to boost performance with additional CPU cores.
- Improved parallelism, e.g., achieving a 4-fold speed-up over the original mSwitch in the presence of multiple concurrent senders and one destination.
- Flexibility, by fully decoupling the packet processing from the switching fabric, and allowing dynamic reconfiguration of the forwarding function. As a proof of concept, we demonstrate how easily we can replace the default learning bridge functionality with Open vSwitch or a port demultiplexer.
- Better integration with the OS and with VMs, by supporting the large MTUs used by virtual machines and NIC acceleration features, as well as the ability to directly connect physical ports and the host stack to the switch.

These improvements are not achieved by simply gluing existing pieces together, but by changing core parts of the design of our base components. Crucially, our system is not just a prototype, but production quality software that runs smoothly within Linux and FreeBSD, and can be used to implement high performance services. Our code is available at www.anonymizedurl.com.

2.2.4 mSwitch Design

In this section we provide an in-depth description of mSwitch's architecture, including the algorithms and mechanisms that allow it to achieve high performance. Further, we discuss the modular mechanism that allows different modules to be seamlessly plugged into the switch.

2.2.4.1 Architecture

Figure 2.2 shows mSwitch's software architecture. mSwitch can attach virtual or physical interfaces as well as the protocol stack of the system's operating system. From the switch's point of view, all of these are abstracted as *ports*, each of which is given a unique index in the switch. When a packet arrives at a port, mSwitch *switches* it by relying on the (pluggable) packet processing stage to tell it which port(s) to send the packet to[1]. In addition, multiple instances of mSwitch can be run simultaneously on the same system, and ports can be created and connected to a switch instance dynamically.

Virtual ports are accessed from user space using the netmap API: they can be virtual machines (e.g., QEMU instances), or generic netmap-enabled applications. The netmap API uses shared memory between the application and the kernel, but each virtual port uses a separate address space to prevent interference between

[1]The processing function is completely arbitrary, so it can also apply transformations to the packet such as encap/decap, NAT, etc.

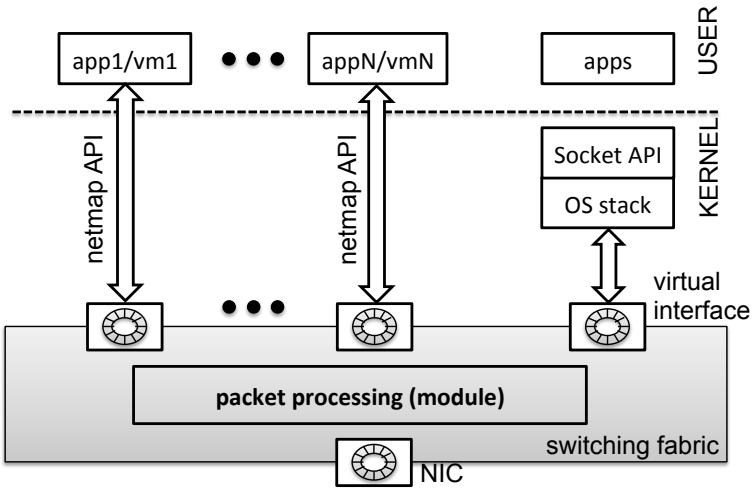


Figure 2.2: mSwitch architecture. A switch supports a large number of ports that can attach to virtual machines or processes, physical ports, or to the host's network stack. mSwitch handles efficient packet delivery between ports, while packet processing (forwarding decisions, filtering etc.) are implemented by loadable kernel modules.

clients. Physical ports are also connected to the switch using a modified version of the netmap API, providing much higher performance than the default device drivers.

Each port connected to the host network stack is linked to a physical (NIC) port. Traffic that the host stack would have sent to the NIC is diverted to mSwitch instead, and from here can be passed to the NIC, or to one of the virtual ports depending on the packet processing logic running on the switch. Likewise, traffic from other ports can reach the host stack if the switching logic decides so.

Packet forwarding is always performed within the kernel and in the context of the thread that generated the packet. This is a user application thread (for virtual ports), a kernel thread (for packets arriving on a physical port), or either for packets coming from a host port, depending on the state of the host stack protocols. Several sending threads may thus contend for access to destination ports.

mSwitch always copies data from the source to the destination netmap buffer. In principle, zero-copy operation could be possible (and cheap) between physical ports and/or the host stack, but this seems an unnecessary optimization: for small packet sizes, once packet headers are read (to make a forwarding decision), the copy comes almost for free; for large packets, the packet rate is much lower so the copy cost (in terms of CPU cycles and memory bandwidth) is not a bottleneck. Cache pollution might be significant, so we may revise this decision in the future.

Copies are instead the best option when switching packets from/to virtual ports: buffers cannot be shared between virtual ports for security reasons, and altering page mappings to transfer ownership of the buffers is immensely more expensive.

2.2.4.2 Packet Forwarding Algorithm

The original mSwitch switch operates on batches of packets to improve efficiency, and groups packets to the same destination before forwarding so that locking costs can be amortized. The grouping algorithm uses a

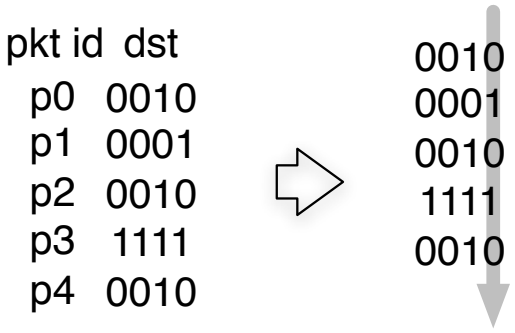


Figure 2.3: Bitmap-based packet forwarding algorithm: packets are labeled from p0 to p4; for each packet, destination(s) are identified and represented in a bitmap (a bit for each possible destination port). The forwarder considers each destination port in turn, scanning the corresponding column of the bitmap to identify the packets bound to the current destination port.

bitmap (see Figure 2.3) to indicate packets' destinations; this is an efficient way to support multicast, but has a forwarding complexity that is linear in the number of connected ports (even for unicast traffic and in the presence of idle ports). As such, this does not scale well for configurations with a large number of ports.

To reduce this complexity, in mSwitch we only allow unicast or broadcast packets (multicast is mapped to broadcast). This lets us replace the bitmap with N+1 lists, one per destination port plus one for broadcast traffic (Figure 2.4). In the forwarding phase, we only need to scan two lists (one for the current port, one for broadcast), which makes this a constant time step, irrespective of the number of ports. Figure 2.8 and 2.9 show the performance gains.

2.2.4.3 Improving Output Parallelism

Once the list of packets destined to a given port has been identified, the packets must copied to the destination port and made available on the ring. In the original mSwitch implementation, the sending thread locks the destination port for the duration of the entire operation; this is done for a batch of packets, not a single one, and so can take relatively long.

Instead, mSwitch improves parallelism by operating in two phases: a sender reserves (under lock[2]) a sufficiently large contiguous range of slots in the output queue, then releases the lock during the copy, allowing concurrent operation on the queue, and finally acquires the lock again to advance queue pointers. This latter phase also handles out-of-order completions of the copy phases, which may occur for many reasons (different batch or packet sizes, cache misses, page faults). With this new strategy, the queue is locked only for the short intervals needed to reserve slots and update queue pointers. As an additional side benefit, we can now tolerate page faults during the copy phase, which allows using userspace buffers as data sources. We provide evaluation results for this mechanism in Section 2.2.5.3.

2.2.4.4 Pluggable Packet Processing

mSwitch uses a hard-wired packet processing function that implements a learning Ethernet bridge. In mSwitch, each switch instance can configure its own packet processing function by loading a suitable kernel

[2]The spinlock could be replaced by a lock-free scheme, but we doubt this would provide any measurable performance gain.

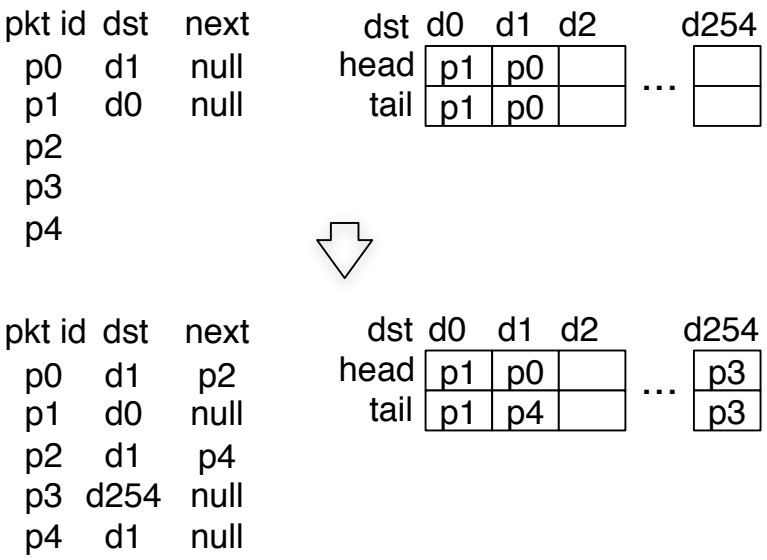


Figure 2.4: List-based packet forwarding: packets are labeled from p0 to p4, destination port indices are labeled from d1 to d254 (254 means broadcast). The sender's "next" field denotes the next packet in a batch, the receiver's list has "head" and "tail" pointers. When the destinations for all packets have been identified (top part of graph), the forwarder serves each destination by scanning the destination's private list and the broadcast one (bottom of graph).

module which implements it. The packet processing function is called once for each packet in a batch, before the actual forwarding takes place, and the response is used to add the packet to the unicast or broadcast lists discussed in Section 2.2.4.2. The function receives a pointer to the packet's buffer and its length, and should return the destination port (with special values indicating "broadcast" or "drop"). The function can perform arbitrary actions on the packet, including modifying its content or length, within the size of the buffer. Speed is not a concern since it will only block the current sending port, not other output ports.

The mSwitch code will take care of performance optimizations (such as prefetching the payload if necessary), as well as validating the return values (e.g., making sure that packets are not bounced back to their source port).

By default, any newly created mSwitch instance uses a packet processing function that implements a basic level-2 learning bridge. At any time during the bridge's lifetime, however, the packet processing function may be replaced. Note that this mechanism is only available to other kernel modules, and is not directly accessible from user space.

2.2.4.5 Other Extensions

The netmap API only supported fixed-size packet buffers (2 Kbytes by default) allocated by the kernel. Many virtualization solutions achieve huge performance improvements by transferring larger frames or even entire 64 Kbyte segments across the ports of the virtual switch. As a consequence, we extended the netmap API to support scatter-gather I/O, allowing up to 64 segments per packet. This has an observable but relatively modest impact on the throughput, see Figure 2.6 and 2.7. More importantly, it permits significant speedups when connected to virtual machines, because the (emulated) network interfaces on the guest can spare the

segmentation of TSO messages.

Additionally, forcing clients to use netmap-supplied buffers to send packets might cause an unnecessary data copy if the client assembles outgoing packets in its own buffers, as is the case for virtual machines. For this reason, we implemented another extension to the netmap API so that senders can now store output packets in their own buffers. Accessing those buffers (to copy the content to the destination port) within the forwarding loop may cause a page fault, but mSwitch can handle this safely because copies now are done without holding a lock as described in Section 2.2.4.3.

2.2.5 Performance Evaluation

In this section we provide a performance evaluation of mSwitch's switching fabric, including scalability with increasing number of ports and NICs; the next section will provide a similar evaluation when plugging in different packet processing functions.

We run all mSwitch experiments on a 2U rack-mount server with an Intel Xeon E5-1650@3.2GHz 6-core CPU (3.8 GHz with Turbo Boost [81]), 16GB of DDR3-ECC 1333MHz RAM (quad channel) and a dual-port, 10 Gbit/s Intel x520-T2 NIC with the 82599 chipset. The two ports of this card are connected via direct cable to another server (Intel Xeon E3-1220@3.1GHz, 16GB of RAM) with a similar card. Unless otherwise stated, we run the CPU of the mSwitch server at 3.8 GHz. In terms of operating system, we rely on FreeBSD 10 [3] for most experiments, and Linux 3.9 for the Open vSwitch experiments in the next section. To generate and count packets we use `pkt-gen`, a fast generator that uses the netmap API and so can be plugged into mSwitch's virtual ports. Throughout, we use Gbps to mean Gigabits per second, Mpps for millions of packets per second. Unless otherwise stated we use a batch size of 1024 packets. Finally, packet sizes in the text and graphs do not include the 4-byte Ethernet checksum; this is to be consistent when using virtual ports, for which that field does not apply.

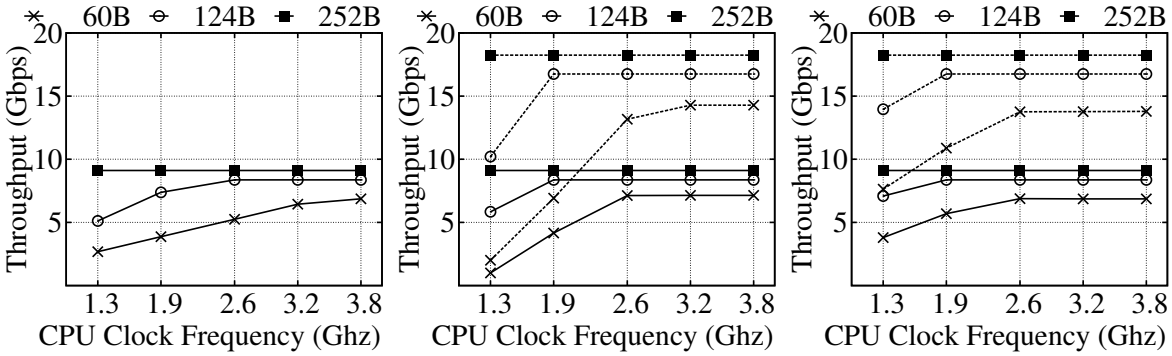
2.2.5.1 Basic Performance

For the first experiments, and to derive a set of baseline performance figures, we implemented a dummy packet processing function. The idea is that the source port (in the case for NICs) or `pkt-gen` (in the case for virtual ports) routes packets to an arbitrary destination port. From there, the packet goes to the dummy module, which returns immediately (thus giving us a base figure for how much it costs for packets to go through the entire switching fabric), and then the switch sends the packet to the destination port.

We evaluate mSwitch's throughput for different packet sizes and combinations of NICs and virtual ports. We further vary our CPU's frequency by either using Turbo Boost to increase it or a `sysctl` to decrease it; this lets us shed light on CPU-bound bottlenecks.

To begin, we connect the two 10 Gbit/s ports of the NIC to the switch, and have it forward packets from one to the other using a single CPU core (Figure 2.5(a)). With this setup, we obtain line rate for all CPU frequencies for 252-byte packets or larger, and line rate for 124-byte ones starting at 2.6 GHz. Minimum-sized packets

[3] mSwitch can also run in Linux.



(a) NIC to NIC (b) NIC to virtual interface (c) Virtual interface to NIC

Figure 2.5: Throughput between 10 Gbit/s NICs and virtual ports for different packet sizes and CPU frequencies. Solid lines are for a single NIC, dashed ones for two.

are more CPU-intensive, requiring us to enable Turbo Boost to reach 96% of line rate (6.9 out of 7.1 Gbps or 14.3 out of 14.88 Mpps). In a separate experiment we confirmed that this small deviation from line rate was caused by our receiver server.

For the next experiment we attach the two NIC ports and two virtual ports to the switch, and run tests with one or two NIC-to-virtual port connections, assigning one CPU core per port pair. The results in Figure 2.5(b) and Figure 2.5(c) are similar to those for the NIC-to-NIC case: mSwitch achieves line rate forwarding at all packet sizes, in both directions, and with either one or two ports. Our system could not fit more than two 10 Gbit/s NIC ports, but as we will see next with virtual port tests, mSwitch supports much higher packet rates, and scalability for these simple traffic patterns is going to be limited by I/O and memory bus capacity rather than by CPU constraints.

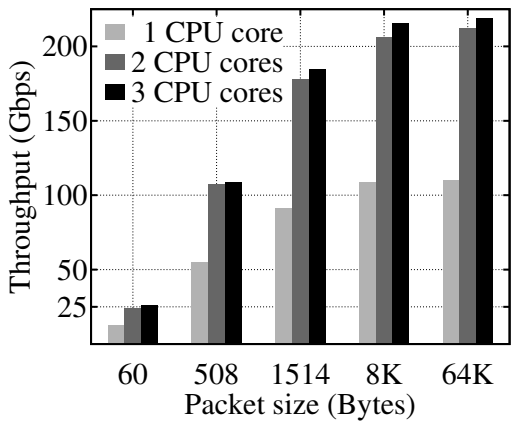


Figure 2.6: Forwarding performance between two virtual ports and different number of CPU cores/rings per port.

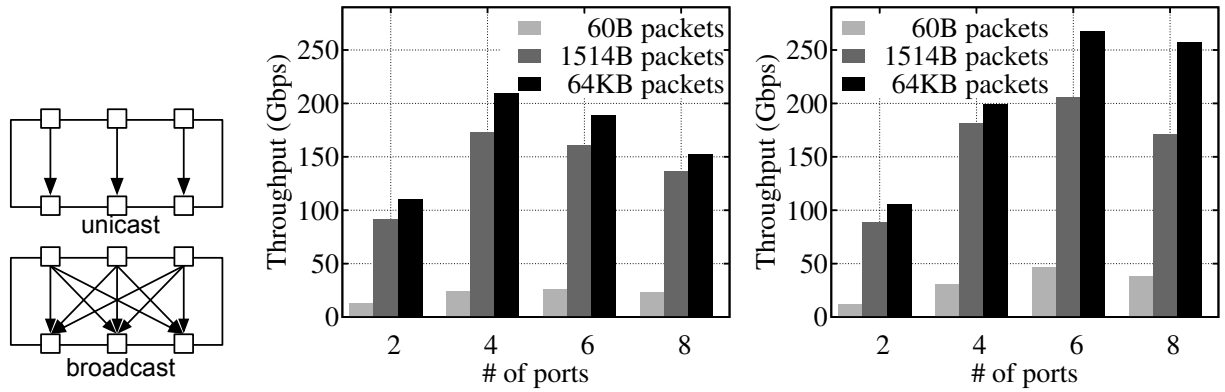
To get an idea of mSwitch's raw switching performance, we attach a pair of virtual ports to the switch and forward packets between them. We also leverage mSwitch's ability to assign multiple packet rings to each port (and a CPU core to each of the rings) to further scale performance. Figure 2.6 shows throughput when assigning an increasing number of CPU cores to each port (up to 3 per port, at which point all 6

cores in our system are in use). We see a rather high rate of about 185.0 Gbps for 1514-byte packets and 25.6 Gbps (53.3 Mpps) for minimum-sized ones. We also see a peak of 215.0 Gbps for 64 KByte "packets"; given that we are not limited by memory bandwidth (our 1333MHz quad channel memory has a maximum theoretical rate of 10.6 GByte/s per channel, so roughly 339 Gbps in total), we suspect the limitation to be CPU frequency.

2.2.5.2 Switching Scalability

Having tested mSwitch's performance with NICs attached, as well as between a pair of virtual ports, we now investigate how its switching capacity scales with additional numbers of virtual ports.

We begin by testing unicast traffic, as shown in Figure 2.7(a) (top). We use an increasing number of port pairs, each pair consisting of a sender and a receiver. Each pair is handled by a single thread which we pin to a separate CPU core (as long as there are more cores than port pairs; when that is not the case, we pin more than one pair of ports to a CPU core in a round-robin fashion).



(a) Experiment topologies. (b) Unicast throughput. (c) Broadcast throughput.

Figure 2.7: Switching capacity with an increasing number of virtual ports. For unicast, each src/dst port pair is assigned a single CPU core, for broadcast each port is given a core. For setups with more than 6 ports (our system has 6 cores) we assign cores in a round-robin fashion.

Figure 2.7(b) shows the results. We observe high switching capacity. For minimum-sized packets, mSwitch achieves rates of 26.2 Gbps (54.5 Mpps) with 6 ports, a rate that decreases slightly down to 22.5 Gbps when we start sharing cores among ports (8 ports). For 1514-byte packets we see a maximum rate of 172 Gbps with 4 ports.

For the second test, we perform the same experiment but this time each sender transmits broadcast packets (Figure 2.7(a), bottom). An mSwitch sender is slower than a receiver because packet processing and forwarding happen in the sender's context. This experiment is thus akin to having more senders, meaning that the receivers should be less idle than in the previous experiment and thus cumulative throughput should be higher (assuming no other bottlenecks).

This is, in fact, what we observe (Figure 2.7(c)): in this case, mSwitch yields a rate of 46.0 Gbps (95.9 Mpps) for 60-byte packets when using all 6 cores, going down to 38.0 Gbps when sharing cores among 8 ports.

1514-byte packets result in a maximum rate of 205.4 Gbps for 6 ports and 170.5 Gbps for 8 ports. As expected, all of these figures are higher than in the unicast case.

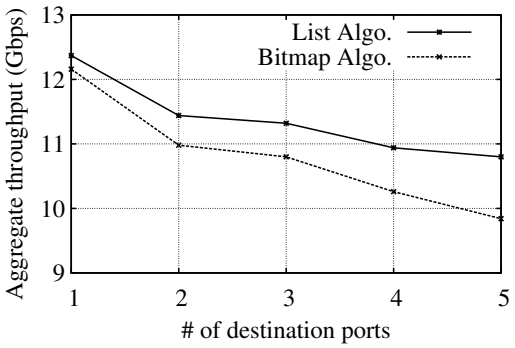


Figure 2.8: Packet forwarding throughput from a single source port to multiple destinations using minimum-sized packets. The graph compares mSwitch's forwarding algorithm (list) to mSwitch's (bitmap).

Next, we evaluate the cost of having a single source send packets to an increasing number of destination ports, up to 5 of them, at which point all 6 cores in our system are in use. Figure 2.8 shows aggregate throughput from all destination ports for minimum-sized packets. For comparison, it includes mSwitch's forwarding algorithm (list) and the one in the mSwitch switch (bitmap). The test results in high rates going from 12.4 Gbps (25.8 Mpps) for one port down to 10.8 Gbps (22.5 Mpps) for five; these rates are improvements over the bitmap algorithm, which yields throughputs of 12.1 Gbps and 9.8 Gbps respectively.

In the final baseline test we compare mSwitch's algorithm to that of mSwitch in the presence of large numbers of ports. In order to remove effects arising from context switches and the number of cores in our system from the comparison, we emulate the destination ports as receive packet queues that automatically empty once packets are copied to them. This mechanism incurs most of the costs of forwarding packets to the destination ports without being bottlenecked by having to wait for the destination ports' context/CPU core to run. We further use minimum-sized packets and a single CPU core.

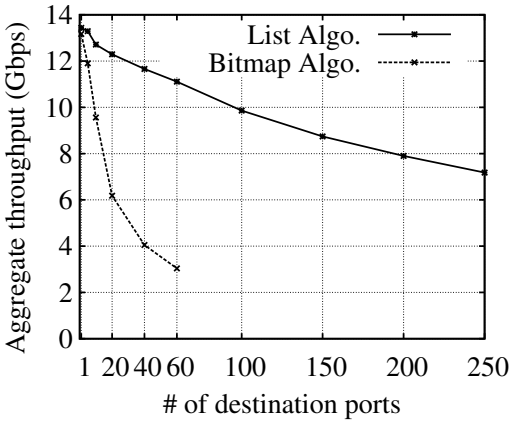


Figure 2.9: Comparison of mSwitch's forwarding algorithm (list) to that of mSwitch (bitmap) in the presence of a large number of active destination ports (single sender, minimum-sized packets). As ports increase, each destination receives a smaller batch of packets in each round hence reducing the efficiency of batch processing.

Figure 2.9 clearly shows the advantage of the new algorithm: we see a linear decrease with increasing number of ports as opposed to an exponential one with the bitmap algorithm. As a point of comparison, for 60 ports the list algorithm yields a rate of 11.1 Gbps (23.2 Mpps) versus 3.0 Gbps (6.4 Mpps) for the bitmap one. This is rather a large improvement, especially if we consider that the list algorithm supports much larger numbers of ports (the bitmap algorithm supports up to 64 ports only; the graph shows only up to 60 for simplicity of illustration). In the presence of 250 destination ports mSwitch is still able to produce a rather respectable 7.2 Gbps forwarding rate.

2.2.5.3 Output Queue Management

We now evaluate the performance gains derived from using mSwitch's parallelization in handling output queues. Recall from Section 2.2.4.3 that mSwitch allows concurrent copies to the output buffers, whereas the mechanism in mSwitch serializes copies.

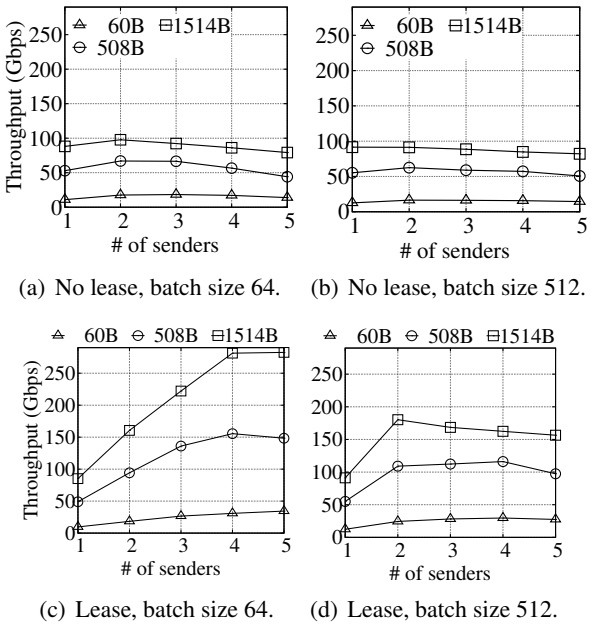


Figure 2.10: Comparison of mSwitch's receive queue mechanism (top graphs, no lease) and mSwitch's lease one (bottom graphs) for different packet and batch sizes. The graphs are done using a single receiver and up to 5 senders, each assigned its own CPU core.

To conduct the experiment we use one receiver and up to five senders in order to create contention on the receiver's queue. Each of these is assigned one of our server's six CPU cores. We further run the tests comparing mSwitch's and mSwitch's queue management schemes with varying packet and batch sizes.

The results in Figure 10 exactly match our expectations. With small batch sizes (Figure 10 a, c) the output queue is large enough to support concurrent operation for all senders. However, mSwitch has approximately constant throughput because of the serialization, while mSwitch scales almost linearly, up to the point where memory bandwidth starts becoming a bottleneck (for large packets). With a larger batch (512 packets versus a queue size of 1024) only a couple of senders at a time can effectively work in parallel, and so mSwitch saturates with just 2 senders. Again, mSwitch is unaffected by the number of senders.

2.2.5.4 Switch Latency

For the final basic performance test we measure the latency of the switch. The experiment uses a request-response transaction (similar to `ping`) between a client and a server connected through mSwitch operating as a learning bridge. In this configuration, the round trip time (RTT) includes twice the latency of the switch, plus two thread wakeups (which consume at least $0.5 - 1$ μs each). When running between two mSwitch virtual ports, the latency is extremely low (an RTT of 5 μs corresponds to a latency of much less than 2 μs). For reference we also include results when using physical 10 Gbit/s ports, both for mSwitch and for the native FreeBSD bridge. These two curves practically overlap, because the bulk of the delay in each RTT is the same in both cases and it is due to interrupt processing (4 times), thread wakeups (4 times) and traversals of low bandwidth buses (PCIe and link).

Note in particular the difference between 60- and 1514-byte packets, which seems high but has a clear explanation. When traversing each link, the packet must be transferred, in sequence, through the PCIe bus (16 Gbit/s) on the transmit side, then the link (10 Gbit/s) and then the PCIe bus on the receive side (memory copies in mSwitch are negligible because of the much larger memory bandwidth). The difference in size between small and large packets is over 11 Kbits, and adding the transmission times for the 8 PCIe bus and 4 link traversals gives a figure in the order of $12 - 15$ μs which matches exactly the measurement results.

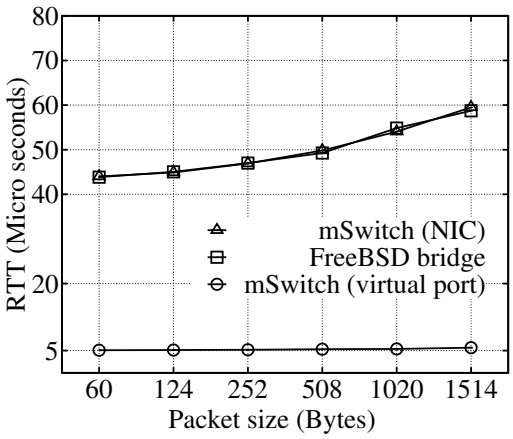


Figure 2.11: RTT of a request-response transaction (going through the switch twice). The actual latency of the switch is smaller than half the RTT.

2.2.6 Use Cases

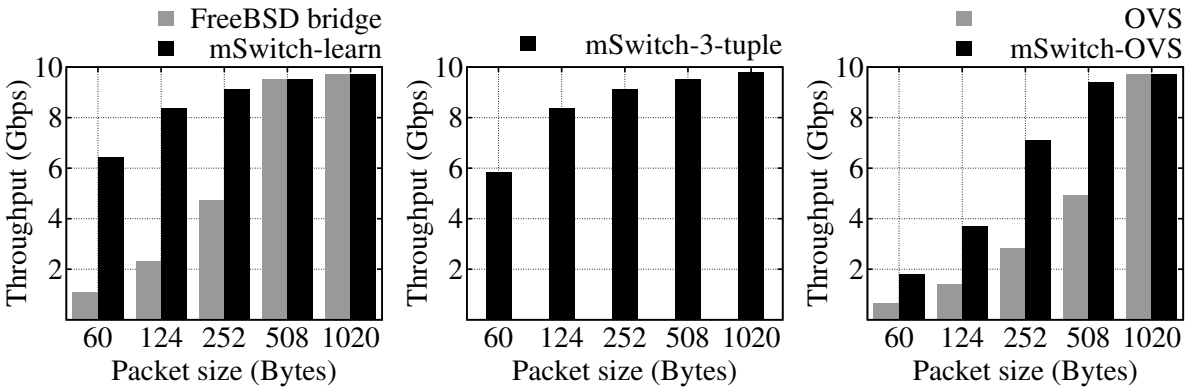
Having tested various aspects of mSwitch's performance using a dummy packet processing function, we now turn to evaluating it while using modules that implement more realistic functionality. In particular, we implement a basic layer-2 learning bridge module; we program a 3-tuple filter module in support of user-level network stacks; and we port/modify Open vSwitch to turn it into an mSwitch module (only about 480 lines of code changed out of 5635 lines of code).

All tests in this section are done using a single CPU core which forwards packets between two 10 Gbit/s ports and through each of the modules.

2.2.6.1 Layer 2 Learning Bridge

We begin by implementing a standard layer 2 learning bridge mSwitch module. The module relies on a forwarding table containing three fields: one to store MAC address hashes (we use SpookyHash [36] for the hash calculation), another to store MAC addresses and a final one to store switch port numbers.

When a packet arrives at the module, we hash its source MAC address, and if no entry exists in the table's first field, we insert a new `<hash(srcmac),srcmac,srcport>` entry. Once finished, we hash the packet's destination port and match the result with the table's first field. If there is a match, we send the packet to the port number found in the third field; otherwise, we broadcast the packet to all ports in the switch (except the one the packet came in on, which mSwitch automatically takes care of doing).



(a) Layer 2 learning bridge.

(b) 3-tuple filter (user-space nw stack support).

(c) Open vSwitch.

Figure 2.12: Packet forwarding performance for different mSwitch modules and packet sizes. For the learning bridge and Open vSwitch module we compare them to their non-mSwitch versions (FreeBSD bridge and standard Open vSwitch, respectively).

Figure 2.12(a) plots the results, along with rates for the FreeBSD bridge for comparison. The graph shows that the FreeBSD bridge can only achieve line rate for 508-byte packets and larger, yielding just 15% of line rate for 60-byte packets. The learning bridge mSwitch module largely outperforms it, reaching 90% of line rate for minimum-sized packets (a six times improvement) and line rate for all other packet sizes.

Finally, it is worth pointing out that the entire learning bridge module (not including the hash function) consists of only 45 lines of code, demonstrating that mSwitch can provide important performance boosts without too much development time.

2.2.6.2 Support for User-Space Protocol Stacks

Recent studies [74] show that more than 86% of Internet paths allow well-designed TCP extensions, meaning that it is still possible to deploy transport layer improvements despite the existence of middleboxes in the network. Hence, the blame for the slow evolution of protocols (with extensions taking many years to become widely used) should be placed on end systems, and especially on the fact that it takes a long time to deploy changes to kernels, the place where protocols are typically implemented.

Towards solving this impasse it would be ideal if we could move protocol stacks up into user space in order

to ease the deployment of new protocols, extensions, or performance optimizations. To get there, we would need at least (1) fast packet I/O to the user-level stacks and (2) a safe way for different stacks to share common network devices as well as mechanisms to ensure that the stacks are not malicious (e.g., that they do not send spoofed packets).

Close inspection reveals that these requirements can be met with an mSwitch module. In greater detail, we implement a filter module able to direct packets between mSwitch ports based on the `<dst IP, protocol type,dst port>` 3-tuple. User-level processes (i.e., the stacks) connect to mSwitch's virtual ports, request the use of a certain 3-tuple, and if the request does not collide with previous ones (including ones that the regular applications `bind()` in the host stack), the module inserts the 3-tuple entry, along with a hash of it and the switch port number that the process is connected to, into its routing table. On packet arrival, the module ensures that packets are IPv4 and that their checksum is correct. If so, it retrieves the packet's 3-tuple, and matches a hash of it with the entries in the table. If there is a match, the packet is forwarded to the corresponding switch port, otherwise it is forwarded to the host stack (then it might be dropped). The module also includes additional filtering to make sure that stacks cannot spoof packets, applying the registered 3-tuple to the source IP, protocol type and source port.

To be able to analyze the performance of the module separately from that of any user-level stack that may make use of it, we run a test that forwards packets through the module and between two 10 Gbit/s ports; this functionality represents the same costs in mSwitch and the module that would be incurred if an actual stack were connected to a port. Figure 2.12(b) shows throughput results. We see a high rate of 5.8 Gbps (12.1 Mpps or 81% of line rate) for 60-byte packets, and full line rate for larger packet sizes, values that are only slightly lower than those from the learning bridge module.

To take this one step further, we built a simple user-level TCP stack library along with an HTTP server built on top of it. Using these, and relying on mSwitch's 3-tuple module for back-end support, we were able to achieve HTTP rates of up to 8 Gbps for 16 KB and higher fetch sizes, and approximately 100 K requests per second for short requests.

2.2.6.3 Open vSwitch

In our final use case, we attempt to see whether it is simple to port existing software packages to mSwitch, and whether doing so provides a performance boost. To this end we target Open vSwitch, modifying code of the version in the 3.9.0 Linux kernel in order to adapt it to mSwitch; we use the term mSwitch-OVS to refer to this mSwitch module.

As a short background, Figure 2.13 illustrates Open vSwitch's datapath. The datapath typically runs in the kernel, and includes one or more NICs or virtual interfaces as ports; it is, at this high level, similar to mSwitch's architecture (recall Figure 2.2). The datapath takes care of receiving packets on NICs ("Device-specific receive" and "Common Receive"), processing them to identify their destination ports ("matching") and scheduling transmission at the corresponding destination interfaces ("Common send" and "Device-

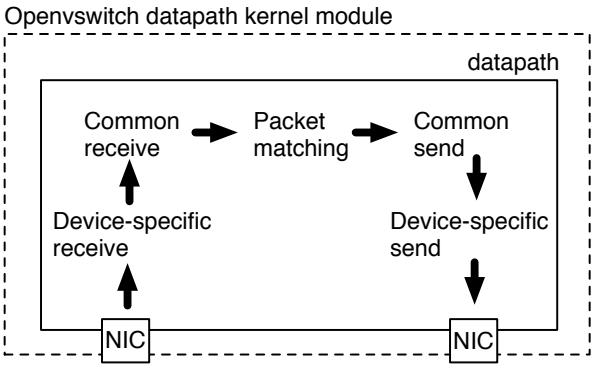


Figure 2.13: Overview of Open vSwitch's datapath.

Specific send").

We accelerate Open vSwitch's datapath by exploiting mSwitch's fast switching fabric while preserving Open vSwitch's packet processing logic and its control interface. In essence, we modify the "Device-specific send/receive" stages of Figure 2.13. The following describes the process in greater detail:

- **Datapath Initialization**: When a datapath starts up, Open vSwitch creates an mSwitch switch instance and registers a packet processing module with it. This module implements a small glue (approximately 80 LoC) between mSwitch's switching fabric and Open vSwitch's common send/receive functions.

- **Attaching an interface**: When a datapath wants to attach a NIC or virtual interface, it connects that interface to mSwitch's switch instance. These operations are implemented using APIs exposed by mSwitch.

- **Packet Processing**: Upon arrival, a packet is first handled by mSwitch's switching fabric (see Figure 2.2). The fabric then passes this packet to the registered packet processing function, which sends it to the "Common receive" stage. From there, the packet goes to "Device-specific send", where, instead of scheduling transmission, mSwitch-OVS returns the switch port index of the interface. Finally, the packet is actually transferred by mSwitch's switching logic.

In all we modified 476 lines of code, mostly added lines. Changes to the original files are only 59 LoC, which means that porting future versions of Open vSwitch should be feasible with relative ease. Of the added lines, about half are to add netmap support to Open vSwitch internal devices. The other half have to do with control functions that glue Open vSwitch commands (e.g., to attach an interface, as well as those from the user-space daemon) to mSwitch ones, including code to have the Open vSwitch package comply with mSwitch's module API.

Figure 2.12(c) shows the packet forwarding rate of the original Open vSwitch (OVS) and that of mSwitch-OVS. We observe important gains ranging from 1.9 times for 508-byte packets all the way up to 2.6 times for minimum-sized ones. As a final point of comparison, recent work [130] provides performance figures for an Intel DPDK-accelerated Open vSwitch implementation on a 2.3 GHz CPU; despite the fact that we have not

optimized the Open vSwitch processing logic in our module at all, we still achieve a (slight) improvement of 14% for minimum-sized packets over the DPDK-based implementation (at 2.3 GHz). Note also that we rely on a single CPU core instead of the two used in the cited work.

2.2.6.4 Throughput versus Filtering Function

As a final data point, we report how the throughput of mSwitch is affected by the complexity of the filtering function. For this test we use minimum-sized packets, and we vary the CPU frequency to verify whether we are CPU bound or limited by other factors. Our baseline is the dummy module used for the experiments in Section 2.2.5.

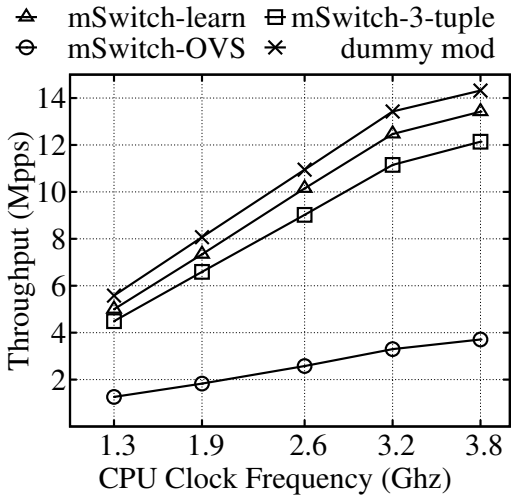


Figure 2.14: Throughput comparison between different mSwitch modules: the dummy module from Section 2.2.5, a L2 learning bridge one, a 3-tuple based one use to support user-level network stacks, and an mSwitch-accelerated Open vSwitch module. Figures are for minimum-sized packets.

We plot the results in Figure 2.14. The graph shows that the hash-based modules (learning bridge or 3-port filter) are relatively inexpensive and do not significantly impact the throughput of the system. Also, at high clock rates the slope of the curves decreases, suggesting a performance bottleneck other than CPU speed.

Conversely, Open vSwitch processing is much more CPU intensive, which is reflected by a much lower forwarding rate, and also an almost linear curve even at the highest clock frequencies.

2.2.7 Discussion and Conclusions

We have presented mSwitch, a high performance, modular software switch. Through an extensive performance evaluation we have shown that it can switch packets at rates of hundreds of gigabits per second on inexpensive, commodity hardware.

From a design perspective, we have attempted to illustrate that logically separating mSwitch into a fast, optimized switching fabric and a specialized, modular packet processing achieves the best from both worlds: the specialization required to reach high performance with the flexibility and ease of development typically found in general packet processing systems. The three use cases we presented are witness to this, with relatively few lines of code (or changes to lines of code) needed in order to either construct high performance

network systems or to boost the performance of existing ones.

Another design decision has to do with how to split functionality between an mSwitch module and the user-level application connecting to the switch. In the latter case the application would receive and inject packets through virtual port(s). Experience with writing the modules presented in this paper suggests that the best approach is to place computationally expensive processing in the application, leaving the module to take care of more basic functionality (e.g., running some filtering and demuxing packets to the right switch ports). Following this model ensures that the switch does not unnecessarily drop "cheap" packets while handling expensive ones. This is akin to OpenFlow's model: fast, simple rule matching in hardware with more advanced processing carried out by software.

As future work, we are aiming to push mSwitch's rates even further. For instance, modern NICs come with hardware features that could further boost performance (e.g., TCP segmentation and checksum offload). We certainly do not want to make our system's performance dependent on such features, but where available, it would be a waste not to support them.

Beyond features in NICs, the results we presented push our single CPU package close to its limits. While higher-frequency CPUs and CPUs with more cores are always on the way, this becomes a case of diminishing returns: for instance, going from a 6-core CPU to a 12-core one can more than quadruple the cost. Consequently, multiple CPU package (NUMA) systems provide better performance for the money. We are thus considering extending mSwitch to be NUMA-aware, such that, as much as possible, memory accesses are always local to the CPU doing them (e.g., we could run one mSwitch instance per CPU package) and so performance scales linearly with additional CPUs.

Finally, mSwitch relies on packet batching, among other things, to achieve high rates. Batching increases burstiness, and so we are planning on investigating how this affects TCP throughput.

2.3 Platform Data Plane (ClickOS)

Over the past years, the presence of hardware-based network appliances (also known as middleboxes) has exploded, to the point where they are now an intrinsic and fundamental part of today's operational networks. Their usefulness is clear enough: they perform a diverse set of functions ranging from security (firewalls, IDSes, traffic scrubbers), traffic shaping (rate limiters, load balancers), dealing with address space exhaustion (NATs) or improving the performance of network applications (traffic accelerators, caches, proxies), to name a few. Today, middleboxes are almost ubiquitous: a third of access networks show symptoms of stateful middlebox processing [76] and in enterprise networks there are as many middleboxes deployed as routers and switches [151].

Despite their usefulness, recent reports and operator feedback reveal that such proprietary middleboxes come with a number of significant drawbacks [61]: middleboxes are expensive to buy and manage [151]. Introducing new features means having to deploy new hardware at the next purchase cycle, a process which on average takes four years. Hardware middleboxes cannot easily be scaled up and down with shifting demand, and so

must provisioned to cope with peak demand, which is wasteful. Finally, a considerable level of investment is needed to develop new hardware-based devices, which leaves potential small players out of the market, and leads to an innovation barrier.

To address these issues, Network Function Virtualization (NFV) has been recently proposed so as to shift middlebox processing from hardware appliances to software running on inexpensive, commodity hardware (e.g., x86 servers with 10Gb NICs). NFV has already gained a considerable momentum: seven of the world's leading telecoms network operators, along with 52 other operators, IT and equipment vendors and technology providers, have initiated a new standards group for the virtualization of network functions [60].

NFV platforms must support multi-tenancy, since they are intended to run concurrently software belonging to the operator and (potentially untrusted) third parties: co-located middleboxes should be isolated not only from a security but also a performance point of view [68]. Further, as middleboxes implement a large range of functionality, platforms should accommodate a wide range of OSes, APIs and software packages.

Is it possible to build a software-based virtualized middlebox platform that fits these requirements? Hypervisor-based technologies such as Xen or KVM are well established candidates and offer security and performance isolation out-of-the-box. However, they only support small numbers of tenants and their networking performance is not satisfactory[4]. At a high-level, the reason for the poor performance is simple: neither the hypervisors (Xen or KVM), nor the guest OSes (e.g. Linux) have been optimized for middlebox processing.

In this paper we present the design, implementation and evaluation of ClickOS, a Xen-based software platform optimized for middlebox processing. To achieve high performance, ClickOS implements an extensive overhaul of Xen's I/O subsystem including changes to the back-end switch, virtual net devices and back and front-end drivers. These changes enable ClickOS to significantly speed up networking in legacy virtual machines: Linux throughput increases from 6.46 Gb/s to 9.68 Gb/s for 1500B packets and from 0.42 Gb/s to 5.73 Gb/s for minimum-sized packets.

A key observation is that Linux (and other commodity OSes) are overkill for most middlebox processing used today, as they include heaps of irrelevant code. A much better choice is to use Click [90] as a language to program middleboxes: Click allows users to build complex middlebox processing configurations by using simple, well known processing elements. Click is great for middlebox processing, but it currently needs Linux to function and so it inherits the overheads of commodity OSes.

To support fast, easily programmable middleboxes, ClickOS implements a minimalistic guest virtual machine that is optimized from the ground up to run Click processing at rates of millions of packets per second. ClickOS images are small (5MB), so it is possible to run a large number of them (up to 400 in our tests). ClickOS virtual machines can boot and instantiate middlebox processing in under 30 milliseconds, and can saturate a 10Gb/s link for almost all packets sizes while concurrently running as many as 100 ClickOS virtual

[4] In our tests, a Xen guest domain running Linux can only reach rates of 6.5 Gb/s on a 10Gb card for 1500-byte packets out of the box; KVM reaches 7.5 Gb/s.

machines on a single CPU core.

2.3.1 Problem Statement

Our goal is to build a multi-tenant, high performance software middlebox platform on commodity hardware. Such a plaform must satisfy a number of performance and security requirements:

Isolation: To support multiple tenants on common infrastructure, the platform should provide isolation in terms of memory, CPU and device access, but also in terms of performance so that middleboxes cannot negatively affect each other's operation.

High Throughput and Low Delay: Middleboxes are typically deployed in operator environments so that it is common for them to have to handle large traffic rates (e.g., multiple 10Gb/s ports); the platform should be able to handle such rates, while adding only negligible delay to end-to-end RTTs.

Scalability: The platform should be able to quickly scale out processing with demand to make better use of additional resources on a server (e.g., more CPU cores or NICs) or additional servers. Likewise, it should be able to quickly scale processing back down when demand diminishes. This implies that virtual machines should boot as quickly as possible.

Consolidation: Having large numbers of middleboxes running concurrently on a single server reduces both capital (less servers to purchase) and operational costs (e.g., energy costs). Further, consolidation helps to simplify middlebox management and its related costs. This implies VMs should have small memory footprints.

How should middleboxes be programmed? The default today is to code them as applications or kernel changes on top of commodity OSes. This allows much **flexibility** in choosing the development tools and languages, at the cost of having to run one commodity OS to support a middlebox.

In addition, a large fraction of functionality is common across different middleboxes, making it important to support **code re-use** to reduce prototyping effort [148].

2.3.2 Related Work

There is plenty of related work we could leverage to build NFV platforms. Given that the goal is to virtualise, the choice is either containers (`chroot`, FreeBSD Jails, Solaris Zones, OpenVZ [170, 171, 121]) which are limited to a single operating system, or hypervisors (VMWare Server, Hyper-V, KVM, Xen [162, 111, 88, 34]).

Hypervisors provide the flexibilty needed for multi-tenant middleboxes (i.e. different guest operating systems are able to run on the same platform), but this is at the cost of high-performance, especially in networking. For high-performance networking with hypervisors, the typical approach today is to utilize device pass-through, whereby, virtual machines are given direct access to a device (NIC). The downside is obvious, first, this complicates live migration, second, scalability is greatly reduced since the device is effectively monopolized by a given virtual machine. Though the latter issue is partly addressed by modern NICs supporting technologies such as hardware multi-queuing, VMDq and SR-IOV [80], we are still limited by the

number of queues offered by the device. This in affect only temporarily alleviates the problem. In this work we will show that it possible to maintain performance scalability without resorting to the rigidity of device pass-through.

Minimalistic OSes and VMs: Minimalistic OSes or micro kernels are attractive because unlike traditional OSes they aim provide just the required functionality for the job. While many minimalist OSes have been proposed and implemented (see for instance [112, 168, 20, 166, 169]), they typically lack driver support for a wide range of devices, especially with respect to NICs and most dont run in virtualized environments. With respect to ClickOS, Mirage [103] is also a Xen VM built on top of MiniOS, but the focus is to create Ocaml, type-safe virtualized applications and as such its performance is not particularly optimized. Erlang on Xen, LuaJIT, HalVM also leverage MiniOS to provide Erlang, Lua, and Haskell programming environments; none of them target middlebox processing nor are optimize network I/O.

Network I/O Optimization: The past few years have seen a numerous efforts towards optimizing the performance of network I/O on commodity hardware. Routebricks [55], looked into creating fast software routers by scaling out to a number of servers. PacketShader [71] took advantage of low cost GPUs to speed up certain types of network processing. More recently, PFQ, PF_RING, Intel DPDK and netmap [117, 51, 79, 139] focused on accelerating networking by directly mapping NIC buffers into user-space memory; in this work we leverage the last of these to provide a more direct pipe between NIC and VMs.

Regarding virtualization, work in the literature has looked at improving the performance of Xen networking [136, 145], and we make use of some of the techniques suggested, such as grant re-use. The works in [176, 116] look into modifying scheduling in the hypervisor scheduling in order to improve I/O performance, however, the results reported are considerably lower than ClickOS. Finally, Hyper-Switch [85] proposes placing the software switch used to mux/demux packets between NICs and VMs inside the hypervisor. Unfortunately, the switch's data plane relies on open vSwitch code [120], resulting in sub-optimal performance. More recently, two separate efforts have looked into optimizing network I/O for KVM [42] [140]; however neither of these has focused on virtualizing middlebox processing and the rates reported are lower than those in this paper.

Software Middleboxes: Comb [148] introduces an architecture for middlebox deployments targeted at consolidation. However, it does not support multi-tenancy or isolation, and the performance figures reported (about 4.5Gb/s for two CPU cores assuming maximum-sized packets) are lower than the line-rate results we present in section 4.2. The work in [151] uses Vyatta software (see below) to run software middleboxes on Amazon EC2 instances. Finally, while number of commercial offerings exist(Cisco [45], Vyatta [163]), there are no publically available detailed evalautions.

2.3.3 ClickOS Design

To achieve isolation and multi-tenancy, we must rely on hypervisor virtualisation, which adds an extra software layer between the hardware and the middlebox software and could potentially hurt throughput

Middlebox	Key Click Elements
Load balancer	`RatedSplitter, HashSwitch`
Firewall	`IPFilter`
NAT	`[IP|UDP|TCP]Rewriter`
DPI	`Classifier, IPClassifier`
Traffic shaper	`BandwidthShaper, DelayShaper`
Tunnel	`IPEncap, IPsecESPEncap`
Multicast	`IPMulticastEtherEncap, IGMP`
BRAS	`PPPControlProtocol, GREEncap`
Monitoring	`IPRateMonitor, TCPCollector`
DDoS prevention	`IPFilter`
IDS	`Classifier, IPClassifier`
IPS	`IPClassifier, IPFilter`
Congestion control	`RED, SetECN`
IPv6/IPv4 proxy	`ProtocolTranslator46`

Table 2.1: Key Click elements available for developing a wide range of middleboxes.

or increase delay. To minimize these effects, para-virtualization is preferable to full virtualization: para-virtualization makes minor changes to the guest OSes, greatly reducing the overheads inherent in full virtualization such as VM exits [26] or the need for instruction emulation [34].

Consequently, we base ClickOS on Xen [34] since its support for para-virtualized VMs provides the possibility to build a low delay, high throughput platform, but potential is not fulfilled out of the box (see section 2.3.5). KVM also supports driver para-virtualization through `virtio` [143], but it yields lower performance (section 2.3.5).

Programming abstractions. Finding the best programming abstraction for middleboxes is an interesting research topic, but we do not set out to tackle it in this paper. Instead, we want to pragmatically choose the best tool out of the ones we have available today.

Software middleboxes are written either as user-space applications on top of commodity operating systems (e.g., Snort or Bro) or as kernel changes (e.g., iptables, tc). Either way, C is the de-facto programming language as it allows programmers to get high performance. C offers great flexibility but has high development and debugging costs, especially in the kernel. In addition, there is not much software one can reuse when programming a new type of middlebox.

To achieve both code re-use and flexibility, we leverage the Click modular router software. Previous work [148] showed that a significant amount of functionality is common across a wide range of middleboxes; Click already makes this explicit, abstracting this functionality into a set of re-usable elements. Second, Click comes with over 300+ stock elements which make it possible to construct middleboxes with minimal effort. Finally, it is extensible, so we are not limited to the functionality provided by the stock elements. As table 2.1 shows, a large number of middleboxes can be implemented using Click.

Click is no panacea: it does not cover all types of middlebox processing, for instance middleboxes that need a fully-fledged TCP stack. In such cases it may be better to use a standard Linux VM. ClickOS helps in this case by optimizing the network I/O (see section 2.3.6).

Running Click Efficiently: By default, Click runs on top of Linux either as a userland process (with poor performance) or as a kernel module. To get proper isolation, we would have to run each Click middlebox

inside a Linux virtual machine. This, however, violates our consolidation and scalability requirements: even stripped down Linux VMs are memory hungry (128MB or more) and take 5s to boot.

Instead, we take a step back and ask *what support does Click need from the operating system to be able to support a wide range of middlebox processing?* The answer is, surprisingly, not much:

- Driver support to be able to handle different types of network interfaces.
- Basic memory management to allocate different data structures, packets, etc.
- A simple scheduler that can switch between running Click element code and servicing interrupts (mostly from the NICs). Even a cooperative scheduler is enough - there is no need for pre-emptive scheduling, or multi-threading.

The first requirement seems problematic, given the large number of interface vendors and variety of models. However, Xen elegantly solves this issue by paravirtualizing drivers: this allows guest domains to access the NIC in a hardware agnostic manner by leveraging device drivers running in the driver-domain, a full-blown Linux machine with driver support for most hardware. The guest VM only has to implement a simple, device-agnostic driver.

Almost all operating systems meet the other two requirements, so there is no need to build one from scratch: we just need an OS that is minimalistic, and is able to boot quickly. Xen comes with MiniOS, a tiny operating system that fits the bill and allows us to build efficient, virtualized middleboxes without all of the unnecessary functionality included in a conventional operating system. MiniOS is the basis for our ClickOS VMs.

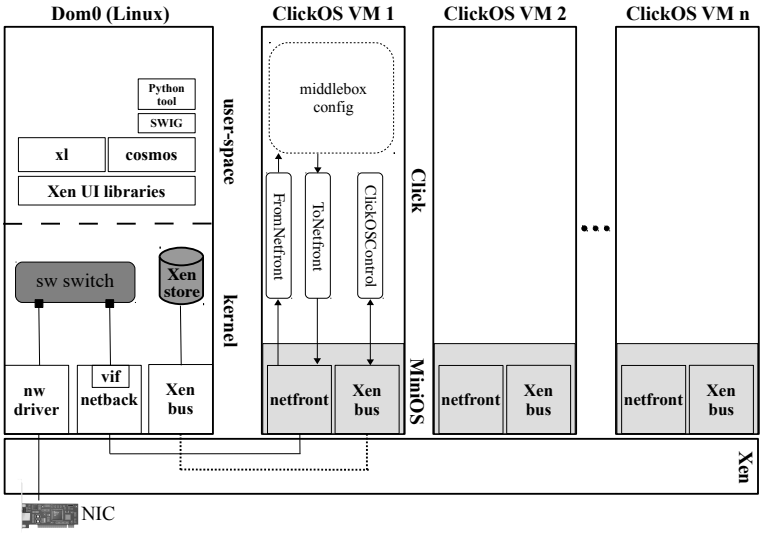


Figure 2.15: ClickOS architecture.

In short, our ClickOS virtualized middlebox platform consists of (1) a number of optimizations to Xen's network I/O sub-system (discussed in detail in section 2.3.6); (2) the actual middlebox virtual machines; and (3) tools to build and manage the ClickOS VMs, including inserting, deleting, and inspecting middlebox state (Fig. 2.15).

2.3.4 ClickOS Virtual Machines

Before describing what a ClickOS virtual machine is, it is useful to give a brief background about Xen. Xen is split into a privileged virtual machine or *domain* called dom0 (typically running Linux), and a set of guest

or user domains comprising the users' virtual machines (also known as domUs). In addition, Xen includes the notion of a *driver domain* VM which hosts the device drivers, though in most cases dom0 acts as the driver domain. Further, Xen has a split-driver model, where the back half of a driver runs in a driver domain, the front-end in the guest VM, and communications between the two happen using shared memory [5] and a common, ring-based API. Xen networking follows this model, with dom0 containing a `netback` driver and the guest VM implementing a `netfront` one. Finally, *event channels* are essentially Xen inter-VM interrupts, and are used to notify VMs about the availability of packets.

MiniOS implements all of the basic functionality needed to run as a Xen VM. MiniOS has a single address space so no kernel/user space separation, and a co-operative scheduler, reducing context switch costs. MiniOS does not have SMP support, but this does not pose a problem for our platform: our model is to have large numbers of tiny VMs rather than a few large VMs using several CPU cores.

Each ClickOS VM consists of the Click modular router software running on top of MiniOS, but building such a VM image is not trivial. MiniOS is intended to be built with standard GCC and as such we can link any standard C library to it, provided we handle for missing functionality appropriately. However, Click is written in `c++`, and so it requires special precautions. The most important of these is that standard `g++` depends on (among others) `ctypes.h` (via `glibc`) which contains OS (Linux) specific dependencies that end up breaking the standard MiniOS `iostream` libraries. To resolve this we developed a new build tool which creates a Linux-independent `c++` cross-compiler using `newlibc` [152].

In addition, our build tool re-designs the standard MiniOS toolchain so that it is possible to quickly and easily build arbitrary, MiniOS-based VMs by simply linking an application's entry point so that it starts on VM boot; this is useful for supporting middleboxes that cannot be easily supported by Click. Regarding libraries, we have been conservative in the number of them we link, and have been driven by need rather than experimentation. In addition to the standard libraries provided with the out-of-the-box MiniOS build (`lwip, zlib, libpci`) we add support for `libpcre, libpcap and libssl`, libraries that certain Click elements depend on. The result is a ClickOS image with 216/282 Click elements, with many of the remaining ones requiring a filesystem to run, something we are planning to add.

Once built, booting a ClickOS image consists of a couple of steps. First, we create the virtual machine itself, which involves reading its configuration, the image file, and writing a set of entries to the Xen store [6] (for instance, the ID of the VM, addresses for packet buffers, event, etc). Second, we attach the VM to the back-end switch, essentially connecting it to physical NICs.

Next, MiniOS boots, after which a special control thread is created. At this point, the control thread creates a special *install* entry in the Xen store to allow users to install Click configurations in the ClickOS VM. Since Click is designed to run on conventional OSes such as Linux or FreeBSD which, among other things,

[5] Memory is shared between Xen VMs through *memory grants*.

[6] The Xen store is a `proc`-like database which resides in dom0 and is used to share control information between it and guest domains.

provide a console through which configurations can be controlled and given that MiniOS does not provide these facilities, we leverage the Xen store to emulate such functionality.

Once the install entry is created, the control thread sets up a watch on it that monitors changes to it. When written to, the thread launches a second MiniOS thread which runs a Click instance, allowing several Click configurations to run within a single ClickOS VM. Removing the config is done by writing an empty string to the install Xen store entry.

We also need to implement Click *element handlers*, which are used to set and retrieve state in elements (e.g, the `AverageCounter` element has a read counter to get the current packet count and a write one to reset the count); to do so, we once again leverage the Xen store. For each VM, we create additional entries for each of the elements in a configuration and their handlers. We further develop a new Click element called `ClickOSControl` which gets transparently inserted into all configurations. This element takes care of interacting, on one end, with the read and write operations happening on the Xen store, and communicating those to the corresponding element handlers within Click.

In order to control these mechanisms which are not standard to all Xen VMs, ClickOS comes with its own dom0 CLI called Cosmos (as opposed to the standard, Xen-provided `xl` tool). Cosmos is built directly on top of the Xen UI libraries (figure 2.15) and therefore does not incur any extraneous costs when processing requests. To simplify development and user interaction, Cosmos implements a SWIG [155] wrapper enabling users to automatically generate Cosmos bindings for any of the SWIG supported languages For convenience, we have also implemented a Python-based ClickOS CLI.

Finally, it is worth mentioning that while MiniOS represents a difficult development environment, programming for ClickOS is relatively painless: development, building and testing can take place in user-space Click, and the resulting code/elements simply added to the ClickOS build process when ready.

2.3.5 Xen Networking Analysis

In this section we investigate where the Xen networking bottlenecks are. Figure 2.15 illustrates the Xen network I/O sub-system: the network driver, software switch, virtual interface and netback driver in dom0 and the netfront driver (either the Linux or MiniOS one) in the guest domains, any of which could be bottlenecks.

In order to get some baseline numbers, we begin by performing a simple throughput test. For this test we used a server with an Intel Xeon E3-1220 3.1GHz 4-core CPU, 16GB memory and an Intel x520-t2 dual Ethernet port 10Gb/s card (about $1,500 including the NIC). The server had Xen 4.2, Open vSwitch as its back-end switch and a single ClickOS virtual machine. The VM was assigned a single core, with the remainder given to dom0.

The first result (labeled “NF-MiniOS” in figure 2.16) shows the performance of the MiniOS netfront driver when sending (Tx, in which case we measure rates at the netback driver in dom0) and receiving (Rx) packets. Out of the box, the MiniOS netfront driver yields poor rates, especially for Rx, where it can barely handle 8 Kp/s. This is unsurprising since it was not optimized for such speeds.

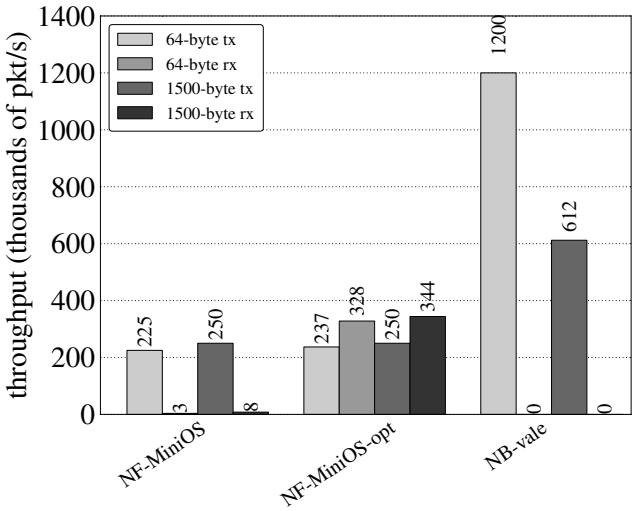


Figure 2.16: Xen performance bottlenecks using a different back-end switch and netfront (NF) and netback (NB) drivers ("opt" stands for optimized).

To improve this receive rate, we modified the netfront driver to re-use memory grants. Memory grants are Xen's mechanism to share memory between two virtual machines, in this case the packet buffers between dom0 and the ClickOS VM. By default, the driver requests a grant for *each* packet, requiring an expensive *hypercall* to the hypervisor (essentially the equivalent of a system call for an OS); we changed the driver so that it gets grants for packet buffers at initialization time, and to re-use these buffers for all packets handled. The driver now also uses polling, further boosting performance.

The results are labeled "NF-MiniOS-opt" in the figure. We see important improvements in Rx rates, from 8 Kp/s to 344 Kp/s for maximum-sized packets. Still, this is far from the 10Gb/s line rate figure of 822 Kp/s, and quite far from the 14.8 Mp/s figure for minimum-sized packets, meaning that other significant bottlenecks remain.

Next, we took a look at the software switch. By default, Xen uses Open vSwitch, which previous work reports as maxing out at 300 Kp/s [141]. As a result, we decided to replace it with the VALE switch [142]. Because VALE ports communicate using the netmap API, we modified the netback driver to implement that API, and removed the *vif* interface in the process. These changes ("NB-vale") gave a noticeable boost of up to 1.2 Mp/s for 64B packets, confirming that the switch was at least partly to blame [7].

Despite the improvement, the figures were still far from line rate speeds. Sub-optimal performance in the presence of a fast software switch, no *vif* and an optimized netfront driver seem to point to issues in the netback driver, or possibly in the communication between netback and netfront drivers. To dig in deeper, we carried out a per-function analysis of the netback driver to determine where the major costs were coming from.

The results in table 2.2 show the main costs in the code path when transmitting a batch of 32 packets. We obtain timings via the `getnstimeofday()` function, and record them using the `trace_printk` function from the lightweight FTrace tracing utility.

[7] We did not implement Rx on this modified netback driver as the objective was to see if the only remaining major bottleneck was the software switch.

description	function	ns
get vif	poll_net_schedule_list	119
handle frags if any	netbk_count_requests	53
alloc skb	alloc_skb reserve_skb	384
alloc page for packet data	xen_netbk_alloc_page	293
build grant op struct	fills *gnttab_copy*	96
extends the skb with the expected size	__skb_put	96
build grant op struct (for frags)	xen_netbk_get_requests	61
add the skb to the Tx queue	__skb_queue_tail	53
checks for packets received packet grant copy dequeue packet from Tx queue	check_rx_xenvif HYPERCALL __skb_dequeue	206 24708 94
copy pkt data to skb	memcpy	90
put a response in the ring	fills *xen_netif_tx_response* notify_via_remote_irq	52
copy frag data	xen_netbk_fill_frags	179
calc checksum	checksum_setup	78
forward pkt to bridge	xenvif_receive_skb	3446

Table 2.2: Per-function netback driver costs when sending a batch of 32 packets. Small or negligible costs are not listed for readability. Timings are in nanoseconds.

The main cost, as expected, comes from the hypercall, essentially a system call between the VM and the hypervisor. Clearly this is required, though its cost can be significantly amortized by techniques such as batching. The next important overhead comes from transmitting packets from the netback driver through the *vif* and onto the switch. The *vif*, basically a tap device, is not fundamental to having a VM communicate with the netback driver and switch, but as shown adds non-negligible costs arising from extra queuing and packet copies. Other further penalties come from using the Xen ring API, which for instance requires responses to all packets transmitted in either direction. Finally, a number of overheads are due to `sk_buff` management, not essential to having a VM transmit packets to the network back-end – especially a non-Linux VM such as ClickOS.

In the next section we discuss how we revamped the Xen I/O network pipe in order to remove or alleviate most of these costs, arriving at the high packet rates reported in section 2.3.7.

2.3.6 Network I/O Re-Design

The Xen network I/O pipe has a number of components and mechanisms that add overhead but that are not fundamental to the task of getting packets in and out of VMs. In order to optimize this, it would be ideal if we could have a more direct path between the back-end NIC and switch and the actual VMs. Conceptually, we would like to directly map ring packet buffers from the device driver or back-end switch all the way into the VMs' memory space, much like certain fast packet I/O frameworks do between kernel and user-space in non-virtualized environments [139, 117, 51].

To achieve this, and to boost overall performance, we take three main steps. First, we replace the standard

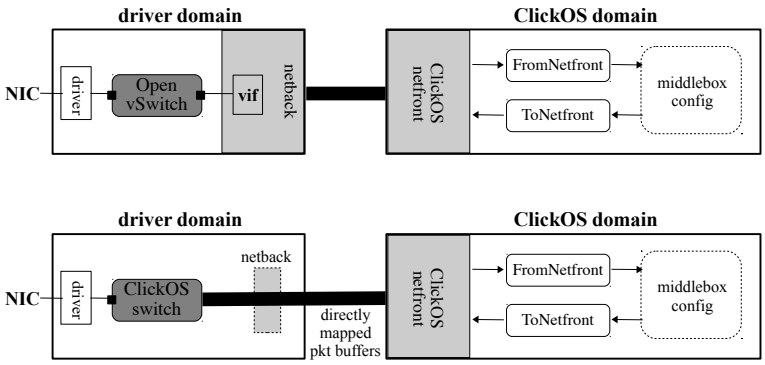


Figure 2.17: Standard Xen network I/O pipe (top) and our optimized, ClickOS one with packet buffers directly mapped into the VM's memory space.

but sub-optimal Open vSwitch back-end switch with a high-speed, ClickOS switch; this switch exposes per-port ring packet buffers which are the ones we map into VM memory space. Second, we observe that in our model the ClickOS switch and netfront driver transfer packets to each other directly so that the netback driver becomes a redundant component of the data plane. As a result, we remove it from the pipe, but keep it as a control plane driver for things like communicating ring buffer addresses (grants) to the netfront driver. Finally, we revamp the netfront driver to map the ring buffers into its memory space.

These changes are illustrated in figure 2.17, which contrasts the standard Xen network pipe (top diagram) with ours (bottom). We dedicate the rest of this section to providing a more detailed explanation of our optimized switch, netback and netfront drivers (both MiniOS' and the Linux one) and finally a few modifications to Click.

2.3.6.0.1 ClickOS Switch Given the throughput limitations of Xen's standard Open vSwitch back-end switch, we decided to replace it with the VALE high-speed switch [102], and to extend its functionality in a number of ways. First, VALE only supports virtual ports, so we add the ability to connect NICs directly to the switch. Second, we increase the maximum number of ports on the switch from 64 to 256 in order to accommodate a large number of VMs.

In addition, we add support for each individual VM to configure the number of slots that the packet buffer ring has, up to a maximum of 2048 slots. As we will see in the evaluation section, larger ring sizes can improve performance at the cost of larger memory requirements.

Finally, we modify the switch so that its switching logic is modular, and replace the standard learning bridge behavior with static MAC address-to-port mappings to boost performance (since in our environment we are in charge of assigning MAC addresses to the VMs this change does not in any way limit our platform's functionality). All of these changes have been now upstreamed into VALE's main code base.

2.3.6.0.2 Netback Driver We redesign the netback driver to turn it (mostly) into a control-plane only driver. Our modified driver is in charge of allocating memory for the receive and transmit packet rings and their buffers and to set-up memory grants for these so that the VM's netfront driver can map them into its memory space. We use the Xen store to communicate the rings' memory grants to the VMs, and use the rings

themselves to tell the VM about the ring buffers' grants; the latter is because these are numerous and would overload the Xen store with entries.

On the data plane side, the driver is only in charge of (1) setting up the `kthreads` that will handle packet transfers between switch and netfront driver; and (2) proxy event channel notifications between the netfront driver and switch to signal the availability of packets.

We also make a few other optimizations to the netback driver. Since the driver is no longer involved with actual packet transfer, we no longer use *vifs* nor OS-specific data structures such as `sk_buffs` for packet processing. As suggested in [173], we adopt a 1:1 model for mapping kernel threads to CPU cores: this avoids unfairness issues. The standard netback uses a single event channel (a Xen interrupt) for notifying the availability of packets for both transmit and receive. Instead, we implement separate Tx and Rx event channels that can be serviced by different cores.

2.3.6.0.3 Netfront Driver We modify MiniOS' netfront driver to be able to map the ring packet buffers exposed by the ClickOS switch into its memory space. Further, since the switch uses the netmap API [139], we implement a netmap module for MiniOS. This module uses the standard netmap data structures and provides the same abstractions as user-space netmap: `open`, `mmap`, `close` and finally `poll` to transmit/receive packets.

Beyond these mechanisms, our netfront driver includes a few other changes

- **Asynchronous Transmit:** In order to speed up transmit throughput, we modify the transmit function to run asynchronously.
- **Grant Re-Use:** Unlike the standard MiniOS netfront driver, we set-up grants once, and re-use them for the lifetime of the VM. This is a well-known technique for improving the performance of Xen's network drivers [145].
- **Linux support:** While our modifications result in important performance increases, the departure from the standard Xen network I/O model means that we break support for other, non-MiniOS guests. To remedy this, we implemented a new Linux netfront driver suited to our optimized network pipe. Using this new netfront results in 10 Gb/s rates for most packet sizes (see section 2.3.7) and allows us to run, at speed, any remaining middleboxes that cannot be easily implemented in Click or on top of MiniOS.

2.3.6.0.4 Click Modifications Finally, we have made a few small changes to Click (version 2.0.1), including adding new elements to send and receive packets via the netfront driver, and optimizations to the InfiniteSource element to allow it to reach high packet rates.

2.3.7 Base Evaluation

Having presented ClickOS' architecture, its components and their optimization, we now provide a thorough, base evaluation of the system. After this, in section 4.2, we will describe the implementation of several middleboxes as well as performance results for them.

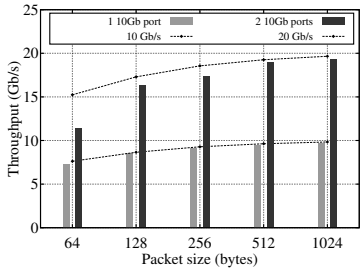


Figure 2.18: ClickOS switch performance using one and two 10 Gb/s NIC ports.

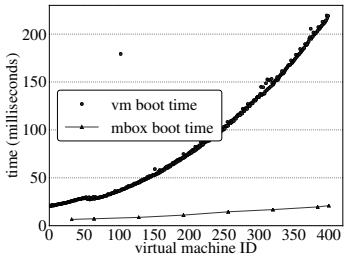


Figure 2.19: Time to create and boot 400 ClickOS virtual machines in sequence and to boot a Click configuration within each of them.

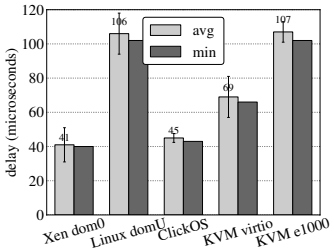


Figure 2.20: Idle VM ping delays for ClickOS, a Linux Xen VM, dom0, and KVM using the e1000 or virtio drivers.

2.3.7.0.5 Experimental Set-up The ClickOS tests in this section were conducted using either (1) a *low-end*, single-CPU Intel Xeon E3-1220 server with 4 cores at 3.1 GHz and 16 GB of DDR3-ECC RAM (most tests); or (2) a *mid-range*, single-CPU Intel Xeon E5-1650 server with 6 cores at 3.2 GHz and 16 GB of DDR3-ECC RAM (switch and scalability tests). In all cases we used Linux 3.6.10 for dom0 and domU, Xen 4.2.0, Click 2.0.1 and netmap's `pkt-gen` application for packet generation and rate measurements.

All packet generation and rate measurements on an external box are done using one or more of the low-end servers, and all NICs are connected through direct cables. For reference, note that 10Gb/s equates to about 14.8 Mp/s for minimum-sized packets and 822 Kp/s for maximum-sized ones.

2.3.7.0.6 ClickOS Switch The goal is to ensure that the switching capacity of our switch is high so that it does not become a bottleneck as more ClickOS VMs, cores and NICs are added to the system.

For this test we rely on a Linux (i.e., non-Xen) system. We use a user-space process running `pkt-gen` to generate packets towards the switch, and from there onto a single 10 Gb/s Ethernet port; a separate, low-end server then uses `pkt-gen` once again to receive the packets and to measure rates. We then add another `pkt-gen` user-process and 10Gb/s Ethernet port to test scalability. Each `pkt-gen`/port pair uses a single CPU core (so two in total for the 20Gb/s test).

For the single port pair case, the switch saturated the 10Gb/s pipe for all packet sizes (figure 2.18). For the two port pairs case, the switch fills up the entire cumulative 20Gb/s pipe for all packet sizes except minimum-sized ones, for which it achieves 70% of line rate. Finally, we also conducted receive experiments (where packets are sent from an external box towards the system hosting the switch) which resulted in roughly similar rates.

2.3.7.0.7 Memory Footprint As stated previously, the basic memory footprint of a ClickOS image is 5MB. In addition to this, a certain amount of memory is needed to allocate the netmap ring packet buffers. How much memory depends on the size of the rings (i.e., how many slots or packets the ring can hold at a time), which can be configured on a per-ClickOS VM basis.

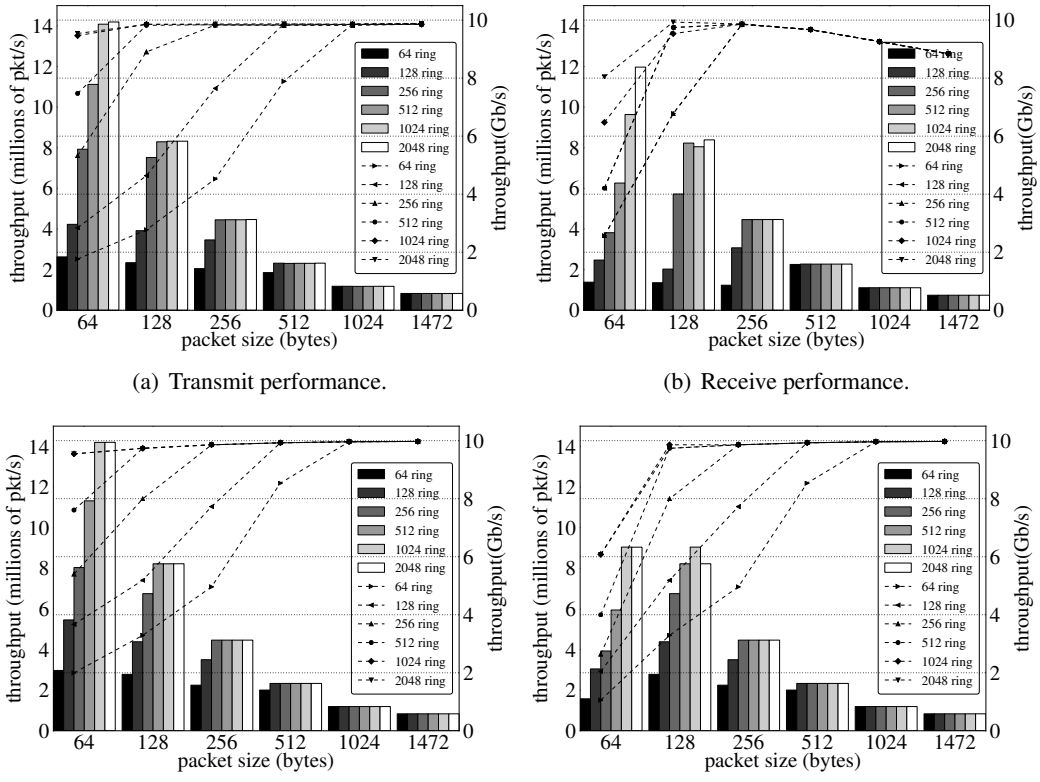
To get an idea of how much memory might be required, table 2.3 shows amounts for different ring sizes,

ring size	amount of memory (KB)	# of grants
64	264	65
128	516	129
256	1032	258
512	2064	516
1024	4128	1032
2048	8260	2065

Table 2.3: Memory requirements for different netmap ring sizes.

ranging from kilobytes for small rings all the way up to 8MB for a 2048-slot ring. As we will see later on in this section, this is a trade-off between the higher throughput that can be achieved with larger rings and the larger number of VMs that can be concurrently run when using small ring sizes. Ultimately, it might be unlikely that a single ClickOS VM will need to handle very large packet rates, so in practice a small ring size might suffice. It is also worth pointing out that larger rings require more memory grants; while there is a maximum number of grants per VM that a Xen system can have, this limit is configurable at boot time.

What about the state that certain middleboxes might contain? To get a feel for this, we inserted 1,000 forwarding rules into an IP router, 1,000 rules into a firewall and 400 into an IDS (see section 4.2 for a description of these middleboxes); the memory consumption from this was 20KB, 87KB and 30KB, respectively, rather small amounts. All in all, even if we use large ring sizes, a ClickOS VM requires approximately 15MB of memory.



(a) Transmit performance.

(b) Receive performance.

(c) Transmit performance (ClickOS).

(d) Receive performance (ClickOS).

Figure 2.21: Performance of a single VM `pkt-gen` running on top of MiniOS/ ClickOS on a single CPU core. The line graphs correspond to the right-hand y-axis.

2.3.7.0.8 Boot Times In this set of tests we use the Cosmos tool to create ClickOS VMs and measure how long it takes for them to boot. A detailed breakdown of the ClickOS boot process may be found in [107], for brevity, here we provide a summary. During boot up most of the time is spent issuing and carrying out the hypercall to create the VM (5.2 milliseconds), building the image (7.1 msecs) and creating the console (4.4 msecs), for a total of about 20.8 msecs. Adding roughly 1.4 msecs to attach the VM to the back-end switch and about 6.6 msecs to install a Click configuration brings the total to about 28.8 msecs from when the command to create the ClickOS VM is issued until the middlebox is up and running.

Next we looked into into how booting large numbers of ClickOS VMs on the same system would affect their boot times. For this test we boot an increasing number of VMs in sequence, up to a maximum of 400, and measure how long it takes for each of them to boot and, additionally, to have a Click configuration installed (figure 2.19). The boot times increase roughly linearly with increasing number of VMs, up to a maximum of 219 msecs for the 400th VM, as do the start-up times for the middleboxes (up to 20.9 msecs). This increase is due to contention on the Xen store and could be improved upon.

2.3.7.0.9 Delay Most middleboxes are meant to work transparently with respect to end users, and as such, should introduce little delay when processing packets. Virtualization technologies are infamous for introducing extra layers and with them additional delay, so we wanted to see how ClickOS' streamlined network I/O pipe would fare.

To set-up the experiment, we create a ClickOS VM running an ICMP responder configuration based on the `ICMPPingResponder` element. We use an external server to ping the ClickOS VM and measure RTT. Further, we run up to 11 other ClickOS VMs that are either idle, performing a CPU-intensive task (essentially an infinite loop) or a memory-intensive-one (repeatedly allocating and deallocating several MBs of memory). The results show low delays of roughly 45 μsecs for the test with idle VMs, a number that stays fairly constant as more VMs are added. For the memory intensive task test the delay is only slightly worse, starting again at 45 μsecs and ramping up to 64 μsecs when running 12 VMs. Finally, the CPU intensive task test results in the largest delays (RTTs of up to 300 μsecs), though these are still small compared to Internet end-to-end delays.

Next, we compared ClickOS' idle delay to that of other systems such as KVM and other Xen domains (figure 2.20). Unsurprisingly, dom0 has a small delay of 41 μsecs since it does not incur the overhead of going through the netback and netfront drivers. This overhead does exist when measuring delay for the standard, unoptimized netback/netfront drivers of a Xen Linux VM (106 μsecs). KVM, in comparison, clocks in at 69 μsecs when using its para-virtualized `virtio` drivers and 107 μsecs for its virtualized e1000 driver.

2.3.7.0.10 Throughput In the next batch of tests we perform a number of baseline measurements to get a basic understanding of what packet rates ClickOS can handle. All of these tests are done on the low-end servers, with one CPU core dedicated to the VM and the remaining three to dom0.

Before testing a ClickOS VM we would like to benchmark the underlying network I/O pipe, from the NIC

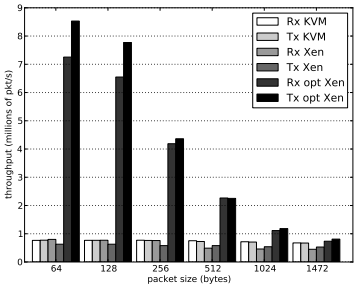


Figure 2.22: Linux domU performance with an optimized (opt), netmap-based netfront driver. For comparison, the graph also plots the performance of out-of-the-box Xen and KVM Linux virtual machines.

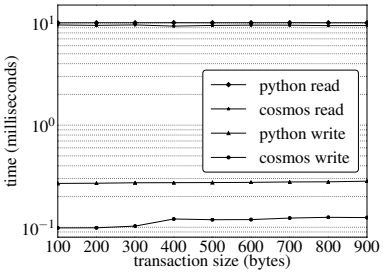


Figure 2.23: ClickOS middlebox state insertion (write) and retrieval (read) for different transaction sizes.

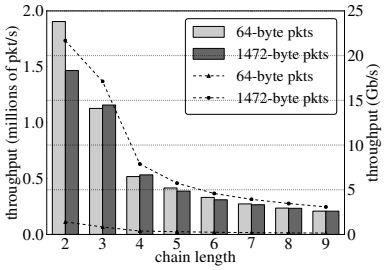


Figure 2.24: Performance when chaining ClickOS VMs back-to-back. The first VM generates packets, the ones in the middle forward them and the last one measures rates. Ring size is set to 64 slots.

through the back-end switch, netback driver and the netfront one. To do so, we employ our build tool to create a special VM consisting of only MiniOS and `pkt-gen` on top of it. After MiniOS boots, `pkt-gen` begins to immediately generate packets (for Tx tests) or measure rates (Rx). We conduct the experiment for different ring sizes (set using a `sysctl` command to the netmap kernel module) and for different packet sizes (for Tx tests this is set via Cosmos before the VM is created).

Figure 2.21 reports the results of the measurements. On transmit, the first thing to notice is that our optimized I/O pipe achieves close to line rate for minimum-sized packets (14.2 Mp/s using 2048-slot rings out of a max of 14.8 Mp/s) and line rate for all other sizes. Further, ring size matters, but mostly for minimum-sized packets. The receive performance is also high but somewhat lower due to extra queuing overheads at the netfront driver.

With these rates in mind, we proceed to deriving baseline numbers for ClickOS itself. In this case, we use a simple Click configuration based on the `AverageCounter` element to measure receive rates and another one based on our modified `InfiniteSource` to generate packets. Figure 2.21(c) shows ClickOS' transmit performance, which is comparable to that produced by the pkt-gen VM, meaning that at least for simple configurations ClickOS adds little overhead. The same is true for receive, except for minimum-sized packets, where the rate drops from about 12.0 Mp/s to 9.0 Mp/s.

For the last set of throughput tests we took a look at the performance of our optimized Linux domU netfront driver, comparing it to that of a standard netfront/Linux domU and KVM. For the latter, we used Linux version 3.6.10, the emulated e1000 driver, Vhost enabled, the standard Linux bridge, and `pkt-gen` once again to generate and measure rates. As seen in figure 2.22 the Tx and Rx rates for KVM and the standard Linux domU are fairly similar, reaching only a fraction of line rate for small packet sizes and up to 7.88 Gb/s (KVM) and 6.46 Gb/s (Xen) for maximum-sized ones. The optimized netfront/Linux domU, on the other hand, hits 8.53 Mp/s for Tx and 7.26 Mp/s for Rx for 64-byte frames and practically line rate for 256-byte

packets and larger.

2.3.7.0.11 State Insertion In order for our middlebox platform to be viable, it has to allow the middleboxes running on it to be quickly configured. For instance, this could involve inserting rules into a firewall or IDS, or adding extra external IP addresses to a carrier-grade NAT. In essence, we would like to test the performance of ClickOS element handlers and their use of the Xen bus and store to communicate state changes. In this test we use Cosmos to perform a large number of reads and writes to a dummy ClickOS element with handlers and measure how long these take for different transaction sizes (i.e., the number of bytes in question for each read and write operation).

Figure 2.23 reports read times of roughly 9.4 milliseconds and writes of about 0.1 milliseconds, numbers that fluctuate little across different transaction sizes. Note that read takes longer since it basically involves doing a write, waiting for the result, and then reading it. However, the more critical operation for middleboxes should be write, since it allows state insertion and deletion. For completeness, we also include measurements when using the XEN python API; in this case, the read and write operations jump to 10.1 and 0.3 milliseconds, respectively.

2.3.7.0.12 Chaining Is it quite common for middleboxes to be chained one after the other in operator networks (e.g., a firewall followed by an IDS). Given that ClickOS has the potential to host large numbers of middleboxes on the same server, we wanted to measure the system's performance when chaining different numbers of middleboxes back-to-back. In greater detail, we instantiate one ClickOS VM to generate packets as fast as possible, another ClickOS VM to measure them, and an increasing number of intermediate ClickOS VMs to simply forward them. As with other tests, we use a single CPU core to handle the VMs and assign the rest to dom0.

As expected, longer chains result in lower rates, from 21.7 Gb/s for a chain of length 2 (just a generator VM and the VM measuring the rate) all the way down to 3.1 Gb/s for a chain with 9 VMs (figure 2.24). Most of the decrease is due to the single CPU running the VMs being overloaded, but also because of the extra copy operations in the back-end switch and the load on dom0. The former could be alleviated with additional CPU cores, while the latter by having multiple switch instances (which our switch supports) or driver domains (which Xen does).

2.3.7.0.13 Scaling Out In the final part of our platform's base evaluation we use our mid-range server to test how well ClickOS scales out with additional VMs, CPU cores and 10 Gb/s NICs. For the first of these, we instantiate an increasing number of ClickOS VMs, up to 100 of them. All of them run on a single CPU core and generate packets as fast as possible towards an outside box which measures the cumulative throughput. In addition, we measure the *individual* contribution of each VM towards the cumulative rate in order to ensure that the platform is fairly scheduling the VMs, that is, that all of them contribute a roughly similar amount to the rate and that none of them are starved of cycles.

Figure 2.25 plots the results. Regardless of the number of VMs, we get a cumulative throughput equivalent

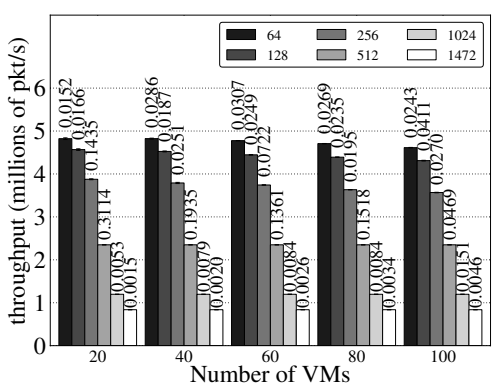


Figure 2.25: Cumulative throughput when running a large number of ClickOS packet generator VMs on a single CPU core and a 10 Gb/s port. Fairness among the VMs is shown by the standard deviation numbers on top of the bars.

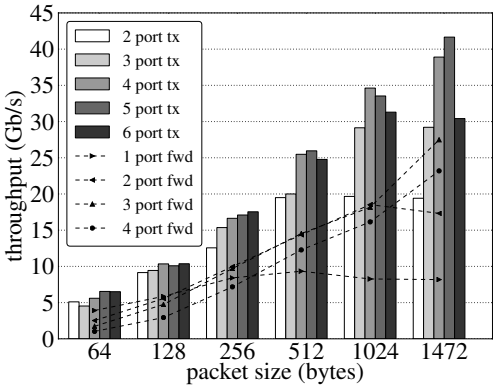


Figure 2.26: Cumulative throughput when using multiple 10 Gb/s ports and one ClickOS VM per port to (1) send out traffic (tx) or (2) forward traffic (fwd).

to line rate for 512-byte packets and larger and a rate of 4.85 Mp/s for minimum-sized ones. The values on top of the bars represent the standard deviation for all the individual rates contributed by each VM; the fact that these values are rather low confirms fairness among the VMs; we further set the ring size to 64.

Next, we test ClickOS' scalability with respect to additional CPU cores and 10 Gb/s ports. We use one packet generator ClickOS VM per port, up to a maximum of six ports. In addition, we assign two cores to dom0 and the remaining four to the ClickOS VMs in a round-robin fashion. Each pair of ports is connected via direct cables to one of our low-end servers and we calculate the cumulative rate measured at them; we further set the ring size to 1024.

For maximum-sized packets we see a steady, line-rate increase as we add ports, VMs and CPU cores, up to 4 ports. After this point, VMs start sharing cores (our system has six of them, with four of them assigned to the VMs) and the performance no longer scales linearly.

For the final experiment we change the configuration that the ClickOS VMs are running from a packet generator to one that simply bounces packets back onto the same interface that they came on (line graphs in figure 2.26). Using this configuration, ClickOS achieves rates of up to 27.5 Gb/s.

Scaling these experiments further would require a CPU with more cores than in our system, or adding NUMA support to ClickOS so that performance scales linearly with additional CPU packages; we leave the latter as future work.

3 The CHANGE Architecture

In this section we present the results of work done with respect to the CHANGE architecture, that is, scenarios involving networks with multiple CHANGE platforms. The main "brains" behind the CHANGE architecture is Symnet, a tool developed in the project that can statically check the correctness and security issues of networks which can contain *stateful* middleboxes (section 3.1). As more and more network processing is shifted towards a software world, it becomes paramount to be able to check that such software will not wreak havoc in operational networks. Symnet is able to check configurations with hundreds of network nodes in seconds, and can use Click-type configurations as input, useful for checking CHANGE platform and architecture set-ups.

In addition, we introduce a novel tool called Tracebox (section 3.2) which can detect a large range of middleboxes in network paths. Such a tool is useful to detect potentially troublesome paths when inter-connecting CHANGE platforms. Finally, section 3.3 provides a detailed description of the CHANGE inter-platform signaling framework.

3.1 Inter-Platform Verification (Symnet)

3.1.1 Introduction

Middleboxes have become nearly ubiquitous in the Internet because they make it easy to augment the network with security features and performance optimizations. Network function virtualization, a recent trend towards virtualizing middlebox functionality, will further accelerate middlebox deployments as it promises cheaper, scalable and easier to upgrade middleboxes.

The downside of this trend is increased complexity: middleboxes make today's networks difficult to operate and troubleshoot and hurt the evolution of the Internet by transforming even the design of simple protocol extensions into something resembling black art [134].

Static checking is a promising approach which helps understanding whether a network is configured properly. Unfortunately, existing tools such as HSA [87] are insufficient as they only focus on routers or assume all middleboxes are stateless. Checking packet forwarding alone only tells a part of the story because middleboxes can severely restrict reachability. The one common trait of most middleboxes is maintenance of per-flow state, and taking packet actions based on that state. Such middleboxes include network address translators, stateful firewalls, application-level proxies, WAN optimizers, traffic normalizers, and so on. Finally, existing tools only answer questions limited to packets; with stateful processing everywhere, we must also be able to answer questions about ***packet flows***.

In this paper we propose a new static analysis technique that can model stateful middleboxes and packet flows in a scalable way. Our solution stems from two key observations:

(i) TCP endpoints and middleboxes can be viewed as parts of a distributed program, and packets can be modeled as variables that are updated by this program. This suggests that symbolic execution, an

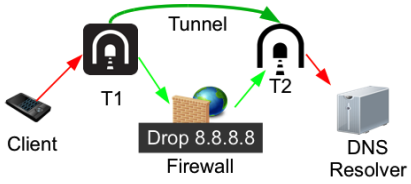


Figure 3.1: An example network containing a firewall and a tunnel

established technique in static analysis of code can be used to check networks.

(ii) Middleboxes keep both global and per-flow state. For instance, a NAT box will keep a list of free ports (global state) and a mapping for each flow to its assigned port (per-flow state). Modeling global state requires model-checking and does not scale. Flow-state, however, can be easily checked if we assume that flow-state creation is independent for different flows.

Starting from these two observations, we have built a tool called SymNet that statically checks network configurations by using symbolic execution and inserting flow state into packets. The tool allows answering different types of network questions, including:

- **Network configuration checking.** Is the network behaving as it should? Is the operator's policy obeyed? Does TCP get through?
- **Dynamic middlebox instantiation.** What happens if a middlebox is removed or added to the current network? How will the traffic change ?
- **Guiding protocol design.** Will a TCP extension work correctly in the presence of a stateful firewall randomizing initial sequence numbers?

SymNet is scalable as its complexity depends linearly on the size of the network. Our prototype implementation takes as input the routers and middleboxes in the network together with the physical topology. Each box is described by a Click [90] configuration. The prototype can check networks of hundreds of boxes in seconds.

We have used SymNet to check different network configurations, network configuration changes and interactions between endpoint protocol semantics and middleboxes. We discuss our findings in Section 4.4.5.

3.1.2 Problem Space

Consider the example in Figure 3.1, where an operator deploys a firewall to prevent its customers from using public DNS resolvers. Clients, however, tunnel their traffic by routing it to T_1, which encrypts it and forwards it to T_2 that decrypts it and sends it onwards to its final destination. A number of interesting questions arise:

(i) Is the firewall doing the job correctly?

(ii) Given the firewall configuration, which addresses are reachable via the tunnel, and via which protocols?

(iii) Is the payload modified en-route to the Internet?

(iv) What if the firewall is stateful? Do the answers to the above questions remain the same?

To address these issues, we require techniques that can model both *stateful middleboxes* as well as *TCP endpoints* and allow us to reason about the properties of different network configurations. Specifically, these techniques need to:

- **Determine the value of header fields** of a packet at different network ports. For instance, a packet accepted by the firewall should not be changed before it reaches the client.
- **Model flow state:** this allows capturing middlebox behaviour that is dependent on flow state. Such behaviour includes network address translators, stateful firewalls, tunnels, proxies and so forth.

Static analysis can be used to answer such questions. A prerequisite of static analysis is an accurate view of the network routing tables and middlebox functionalities in order to model them appropriately. With this information, it is possible to test IP reachability by tracking the possible values for IP source and destination addresses in packets, in different parts of the network [174]. However, [174] only models routers and stateless firewalls. Header Space Analysis (HSA) [87] is an extension of [174] which models arbitrary stateless middleboxes as functions which transform header spaces. The latter are multi-dimensional spaces where each dimension corresponds to a header bit.

HSA is not sufficient to answer our previously-stated questions for the following reasons: first it does not capture middlebox state, and thus cannot perform an accurate analysis on most networks as these contain NATs, stateful firewalls, DPI boxes, etc.

Second, while HSA is designed to determine what packets can reach a given destination, it is unable to efficiently examine how packets are changed by the network. Consider our example: answering the first question boils down to establishing whether the firewall reads the original destination address of the packet, as sent by the client. Given a fixed destination address d of a packet sent by the client, HSA is able to check if the firewall does indeed read d. However, it is unable to perform the same verification for an arbitrary (and a-priori unknown) destination address. Using HSA, this can only be achieved by doing reachability tests for each possible destination address d, which is exponential in the number of bits used to represent d.

Question 3 is the same as asking whether a packet can be modified by the tunnel. Assume any packet can arrive at the tunnel: this will be modelled by a headerspace containing only unknown bit values. As the packet enters the tunnel, HSA will capture the change to the outer header by setting the IP source and destination address bits to those of the tunnel endpoints; as the packet comes out, these are again replaced with unknown bit values. However, HSA is unable to infer that the latter unknown values coincide with the ones which entered the tunnel in the first place. Having a "don't know"[1] packet go in, and a similar packet go out says nothing about the actual change which occurs in the tunnel; The output is similar to that of a middlebox that

[1] packet header with unknown address field values

randomly changes bits in the header: whenever the input is a "don't know" packet, the output is also a "don't know" packet.

AntEater [105] takes a different approach to static network modeling. It expresses desirable network properties using boolean expressions and then employs a SAT-solver to see if these properties hold. If they don't, AntEater will provide example packets that violate that property. AntEater does not model state either; additionally, its reliance on SAT-solvers makes it inapplicable to large networks.

A blend of static checking and packet injection is used to automatically create packets that exercise known rules, such as ATPG [177]. Another hybrid approach uses HSA to dynamically check the correctness of updates in SDN networks [86]. Both these tools inherit the drawbacks of HSA: they cannot model stateful middleboxes.

3.1.3 Symbolic Network Analysis

SymNet relies on symbolic execution—a technique prevalent in compilers—to check network properties such as TCP connectivity between specified endpoints, the existence of loops, reachability of a certain node, etc.

The key idea underlying SymNet is to treat sets of possible packets as *symbolic packets* or *flows* and network devices as transformation functions or *rules*. As packets travel through the network, they are transformed much in the same way symbolic values are modified during the symbolic execution of a program. Flows consist of variables which are possibly bound to expressions. The latter have a two-fold usage. On one hand they model the header fields of a packet. On the other, they are used to keep track of the device state. Thus, stateful devices such as NATs, tunnels, (stateful) firewalls, proxies, etc. can be accurately modeled as transformations which operate both on header as well as on state variables. The set of reachable packets from a given source to a destination, as well as the detection of loops are achieved by computing the least fix-point of an operator which aggregates all transformation functions modeling the network. This is achieved in linear time, with respect to the network size.

Modeling packets. Following HSA (Headerspace Analysis) [87], we model the set of all possible headers having n fields as an n-dimensional space. A flow is a subset of such a space. HSA assigns to each header bit one of the values *0*, *1*, *x*. The latter models an unknown bit value. Unlike HSA, we introduce symbolic expressions as possible values assigned to header fields. The most basic (and useful) such expression is the variable. Thus, we replace HSA assignments such as $bit_i = x$ by $bit_i = v$, where v is a variable[2]. Even without any additional information regarding the value of v, the latter assignment is more meaningful, as the unknown value of bit i can be properly referred (as v), and also used in other expressions. For instance, Question 1 from Section 3.1.2 can now be answered by assigning a variable v_d to the destination address set by the client, and checking if the incoming flow at the firewall also contains v_d as destination address.

Let $\mathbb{V}ars$ and *Expr* designate finite sets whose elements are called (header) *variables* and *expressions*, respec-

[2] Also, variable-value pairs need not refer to individual bits of the header. For instance, an IP address $a.b.c.d$ can be modeled in the standard way as a sequence of 32 bits, but also as a string "`a.b.c.d`". The representation choice is left to the modeler.

tively. In this paper we only consider expressions generated by the following BNF grammar:

$$expr ::= c \mid v \mid \neg expr$$

where $c \in Expr$ and $v \in \mathbb{V}ars$.

Let CC $\subseteq \mathcal{P}$ ($\mathbb{V}ars \times Expr$) . such that $(port, v) \in C$ for some $v \in Expr$. C models a *compact space* of packets. The set $U \subseteq \mathcal{P}(C)$ models an arbitrary space of packets, that is, a reunion of compact spaces. An element $f \in U$, $f \neq \emptyset$, models a *flow* on a given port from the network and is of the form $f = \{cf_1, \ldots, cf_n\}$ where each $cf_i \in C$ is a compact space. f models the subspace $cf_1 \cup cf_2 \ldots \cup cf_n$ of U. Whenever a flow is a compact space (i.e. it is of the form $\{cf\}$), we simply write cf instead of $\{cf\}$, in order to avoid using double brackets. We write $f|_{v=c}$ to refer to the flow obtained from f by enforcing that v has value c and $f|_v$ to refer to the flow obtained from f where v has no associated value.

Example 1 (Flow)**.** *The flow* $f_1 = \{(port, p_1),$
$(IP_{src}, 1.1.1.1), (TCP_{src}, v_1), (IP_{dst}, 2.2.2.2), (TCP_{dst}, 80)\}$
models the set of all packets injected at port p_1 *which are sent to an app running at device* 2.2.2.2 *on port* 80*.*

Modeling state. Unlike HSA, which models stateless networks only, SymNet can model stateful devices by treating per-flow state as additional dimensions in the packet header. State is not explicitly bound to devices, rather "pushed" in the flow itself. The simplest example is a stateful firewall that only allows outgoing connections: as the symbolic packet goes out to the Internet, the firewall pushes a new variable called *firewall-ok* into the packet. At the remote end, this variable is copied in the response packet. On the reverse path, the firewall only allows symbolic packets that contain the *firewall-ok* variable.

Such state modelling scales well, as it avoids the complexity explosion of model-checking techniques. However, our model makes a few strong assumptions. First, we assume each flow's state is independent of the other flows, so flow ordering that not matter—in the firewall example this is obviously true, as long as the firewall has enough memory to "remember" the flow. Second, we completely bypass global variables held by the middleboxes, and assume these do not (normally) affect flow state. Packet-counters and other statistics normally do not affect middlebox functionality, so this assumption should hold true.

Modeling the network. The network is abstracted as a collection of *rules* which can be (i) matched by certain flows and (ii) whenever this is the case can modify the flow accordingly. Formally, a rule is a pair $r = (m, a)$ where $m : U \to \{0, 1\}$ decides whether the rule is applicable and $a : U \to U$ provides the transformation logic of the rule. We occasionally write $r(f)$ instead of $a(f)$ to refer to the application of rule r on flow f.

Example 2 (Rule, matching, application)**.** *The rule* r_1 *takes any packet arriving at port* p_1 *and forwards it on port* p_2*.* r_1 *is applicable on* f_1 *and once applied, it produces the flow* $f_2 = f_1|_{port=p_2}$*. Formally,* $r_1 = (m_1, a_1)$*, where* $m_1(f) = 1$ *if* $(port, p_1) \in f$ *and* $m_1(f) = 0$ *otherwise, and* $a_1(f) = f|_{port=p_2}$*.*

The rule r_2 models the behaviour of a NAT device: the source IP and TCP addresses are overridden by those of the NAT. Also, the flow will store the device state, i.e. the original IP and TCP addresses, using the (state) variables IP_{nat} and TCP_{nat}. $r_2 = (m_2, a_2)$ where $m_2(f) = 1$ is a constant function and:

$$a_2(f) = f|_{IP_{nat}=f(IP_{src}),TCP_{nat}=f(TCP_{src}),IP_{src}=v_3,TCP_{src}=v_4}$$

In order to build up more complex rules from simpler ones, we introduce the *rule composition operator* $\circ$, defined as follows: $(m, a) \circ (m', a') = (m_c, a_c)$ where $m_c(f) = m(f) \wedge m'(a(f))$ and $a_c = a'(a(f))$. m_c verifies if a flow is matched by the first rule, and whenever this holds, if the subsequent transformation also matches the second rule. a_c simply encodes standard function composition. Thus, rule $r_{nat} = r_1 \circ r_2$ combines the topology-related port-forwarding with the actual NAT behavior. Rule composition is an essential means for building rules in a modular way, and which is also suitable for merging configuration files.

We define a *network function* $NF : U \rightarrow \mathcal{P}(U)$ with respect to a set R of rules, as: $NF(f) = \bigcup_{r \in match_R(f)} r(f)$, where $match_R(f)$ are the rules from R which match f.

Reachability Given a flow f where $port = p_s$ and a destination port p_d, reachability is the problem of establishing the set of packets which can reach p_d if f is sent from p_s. In what follows NF is a network function.

Solving reachability amounts to exploring all network transformations of f, that is, applying all rules from NF which match f and recursively applying the same procedure on all flows resulted from the (previous) rule application(s). Formally, this amounts to the application of the operator $OP_A : \mathcal{P}(U) \rightarrow \mathcal{P}(U)$, defined with respect to NF and a set of flows A, as follows:

$$OP_A(X) = \begin{cases} A & \text{if } X = \emptyset \\ A \cup \bigcup_{f \in X} NF(f) & \text{otherwise} \end{cases}$$

Consider the following (infinite) sequence:

$$X_0 = \emptyset,\ X_1 = OP_{\{f\}}(X_0),\ X_2 = OP_{\{f\}}(X_1)\ \ldots$$

Note that X_1 is the singleton set containing the initial flow f. X_2 is the set of all flows which result from the application of all matching rules on f. In other words, X_2 contains all flows which are reachable from f in a single step. Similarly, each X_i is the set of flows which are reachable from f in $i-1$ steps. The *least fix-point of* $OP_{\{f\}}$, that is, the smallest set X_i, such that $X_{i+1} = OP_{\{f\}}(X_i) = X_i$, contains all flows which are reachable at all ports from the network.

Algorithm 1 describes our reachability procedure. At lines 2-7, the set X contains all previously computed flows, or is $\emptyset$ if the current iteration is the first one. Y contains precisely those flows which have been computed in the previous step. The loop will build up new flows from the former ones (lines 4-6), until the

Algorithm 1: Reach(NF,f,p_d)

```
1  X = ∅, Y = {f}, R = ∅;
2  while X ≠ X ∪ Y do
3  |   X = X ∪ Y, Y' = ∅;
4  |   for f ∈ Y do Y' = Y' ∪ NF(f) ;
5  |   Y = Y';
6  end
7  for f ∈ X do
8  |   if (port, p_d) ∈ f then R = R ∪ {f} ;
9  end
10 return R
```

set of all flows computed at the current step ($X \cup Y$) coincides with the set computed in the previous step (X). Finally, in lines 7-9, the flows reaching p_d will be extracted from the fix-point.

Proposition 3.1. *For any network function NF and set A, OP_A^{NF} has a least fix-point.*

Proof. According to Tarski's Theorem [157], it is sufficient to show that OP_A is monotone, that is $X \subseteq Y$ implies $OP_A(X) \subseteq OP_A(Y)$. Assume $X \subseteq Y$. Then $\cup_{f \in X} NF(f) \subseteq \cup_{f \in Y} NF(f)$ and thus we also have that $OP_A(X) \subseteq OP_A(Y)$. □

Proposition 3.2. *The algorithm Reach is correct.*

Proof. *Reach* computes the least fix-point of $OP_{\{f\}}$, and thus the set of all flows reachable in the network. The proof is done via structural induction and is straightforward. Termination is guaranteed by the existence of a fix-point for *OP*, via Proposition 3.1. □

In the absence of loops, the fix-point computation is $O(P * |NF|)$, where $|NF|$ is the number of rules from the network, and P is the number of network ports.

Loop detection *Topological loops* are identified by flow histories of the type $hph'p$, where p is a port and h, h' are (sub)histories. They are not necessarily infinite. For instance, in a loop where network nodes always decrease the TTL field, packets will eventually be dropped. Such a loop is *finite*. In what follows, we will focus on detecting *infinite loops* only. The principle is similar to that applied for reachability. First, we fix the set A to contain the most general flows which originate from each port: $A = \cup_{p \in Ports}\{(port, p)\}$. Second, we introduce an additional rule r_{loop} handling a variable *History* which stores the sequence of ports explored up to the current moment, in each flow. All rules are subsequently composed with r_{loop}. Thus, the application of each rule also updates the *History* variable. Third, when computing the least fix-point of OP_A we shall compare flows $f|_{History}$ and $f'|_{History}$ instead of simply f and f', to ensure the monotonicity of OP_A. Finally, a loop is identified by any two flows f and f' in the least fix-point of OP_A, such that the history of f is hp, that of f' is $hph'p$ and $f'|_{History}$ is more general than $f|_{History}$. In other words, there is a topological loop at port p, and the set of packets reaching p the second time is guaranteed to match the same network rules as the first time.

Modelling a NAT. The previously introduced rule r_{nat}, applied NAT transformations on all incoming packets from port p_1, and forwarded them on port p_2. In what follows we continue Example 2 and illustrate how reachability can be used to establish whether communication at the TCP layer is possible between two devices (*client*, *server*) separated by a NAT. The network configuration is shown in Figure 3.2(a). The client is modelled by the rule which injects the flow $f_0 = f_1|_{port=p_0}$ (on port p_0). f_0 models the set of packets generated by *client* and which are destined for *server*. We model topology links as rules matching and transforming the appropriate ports, and leaving the rest of the flow variables unchanged. Finally, the behaviour of the NAT for packets arriving at port p_2 is modelled by a rule which looks up the state variables IP_{nat} and TCP_{nat} and restores their content to the header variables IP_{dst} and TCP_{dst}. The sequence of flows and how they are transformed is depicted in Figure 3.2(a). Note that all outgoing flows from port p_2 will store the NAT's state, which is further used once a response from the server arrives at the same port.

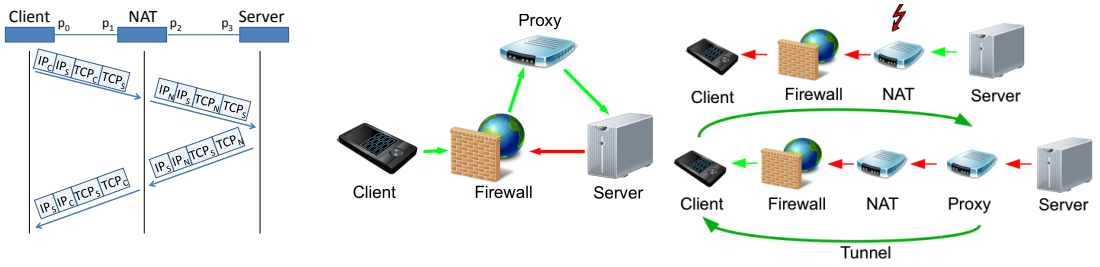
Using reachability to check (stateful) network properties. One interesting property is whether or not a header field of a packet sent from a source port p_s can be read at a destination port p_d. Question 3 from Section 3.1.2 boils down to this property, which can be checked by creating a flow f where the header field variable at hand is bound to an unused variable (i.e. the field can have any possible value), running reachability with f from p_s to p_d, and checking if the header field variable binding remains unchanged. Question 2 from Section 3.1.2 can also be answered similarly. In this case, we simply look at the expression bound to the destination address of the flow which is reachable at p_d.

TCP connectivity between p_s and p_d can be checked by (i) building up a flow f where TCP and IP source and destination variables are bound to unused variables and the variable modelling the SYN flag is set to true, (ii) building a "response rule" at p_d that swaps the IP addresses and TCP ports in the packet, and also sets the ACK flag. (iii) performing reachability from p_s with flow f to p_s, and examining the reachable flow at p_s to test if the IP addresses and ports are mirrored (i.e. the destination address of the outgoing flow is the source address of the incoming flow). If this is the case, TCP connectivity is possible. This is how we answer Questions 1 to 4 at the transport level.

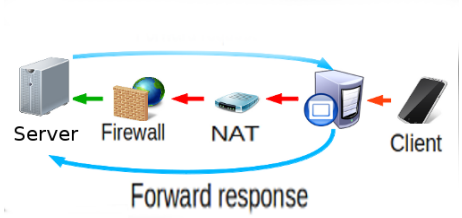
3.1.4 Implementation

Our implementation of SymNet has two components. The first is developed in Haskell, and allows building up abstract models of a network configuration and/or topology. The second component, developed in Scala, uses Antlr to parse network configurations specified using the Click[90] language and generates abstract Haskell models. Symbolic execution is performed in Haskell on such models following the method described in the previous section. [3]

[3]The combination of two languages has the advantage that model generation and symbolic execution have no interdependencies. Thus, our approach can be naturally extended from Click to any network specification language.



(a) Modelling a NAT (b) Enterprise network (c) Inbound connectivity in cellular networks



(d) Cellular network (B)

Figure 3.2: Simple network configurations to be checked with SymNet

3.1.5 Evaluation

Our evaluation of SymNet focuses on several questions. First, we would like to check whether SymNet works correctly and gives appropriate answers in simple topologies that we can reason about. Second, we are interested in understanding how to model per-flow state for specific middleboxes. Finally, we want to see if SymNet can help guide protocol design decisions.

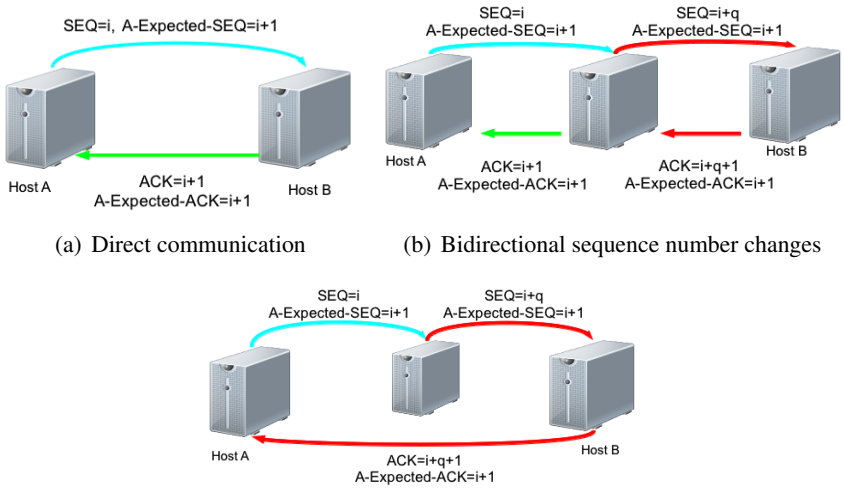
In our experiments, we use header variables for the IP and TCP source and destination address fields, the flags field (i.e. SYN/FIN/ACK), as well as the sequence number, segment length and acknowledgment number fields.

Enterprise network. Consider a typical small enterprise network running a stateful firewall and a client (C) of that network that uses a proxy (Figure 3.2(b)). All the client's requests pass through a stateful firewall. The proxy forwards the traffic to the server, as instructed by the client. To model connection state changes of the firewall, we insert a firewall-specific variable that records the flow's 5-tuple as the SYN flow goes from the client to the server.

We consider two behaviors at the proxy. If the proxy overwrites just the destination address of the flow, the server will reply directly to the client and the returning flow will not travel back through the proxy. As the flow arrives at the firewall, the firewall state (saved in the flow) does not match the flow's header. The output of SymNet in this case is the empty flow: there is no TCP connectivity between C and S, despite the fact that a flow from C arrives at S.

If the proxy also changes the source address of the flow, the returning flow will pass through and allow it to perform the reverse mapping. In this case, SymNet shows there is connectivity.

Tunnel. A client machine wants to do a DNS lookup using the Google Public DNS resolver (8.8.8.8). Its operator disallows external resolvers by deploying a firewall that explicitly forbids packets to 8.8.8.8, as



(a) Direct communication (b) Bidirectional sequence number changes

(c) Unidirectional sequence number changes

Figure 3.3: Modeling sequence number consistency in TCP

shown in Figure 3.1.

We would like to know how the tunnel should be configured to allow the client to send packets to 8.8.8.8. To answer this question, we bind the IP addresses of the tunnel endpoints to unitialized variables—i.e. they can take any value. Next, we run a reachability test from the client to the input port of the box T_2. At T_2, the constraints on the IP destination address give us the answer: 'Not "8.8.8.8"'.

Hence, as long as T_2's address is different from 8.8.8.8, the flow will reach T_2. The listing below shows the output of SymNet (the client uses the IP "141.85.37.151" and DST address "8.8.8.8"). Port numbers are omitted to improve readability. Note the variables named "tunnel-...": these record the original header values, allowing SymNet to perform the operations needed at tunnel exit.

```
[("SRC-ADDR" := CVar "Var-1") ^ "DST-ADDR" := And (CVar "Var-2") (Not (CVal "8.8.8.8")) ^
"tunnel-SRC-ADDR" := CVal "141.85.37.151" ^ "tunnel-DST-ADDR" := CVal "8.8.8.8")]
```

By running reachability one step further, after the traffic exits the tunnel, we see that the IP addresses in the packets are exactly the same as set by the source: hence, the source can reach 8.8.8.8:

```
[("SRC-ADDR" := CVal "141.85.37.151") ^ "DST-ADDR" := CVal "8.8.8.8" )]
```

Cellular connectivity. A mobile application wishes to receive incoming TCP/IP connections (e.g. push notifications). However, the network operator runs a NAT and a stateful firewall. We use SymNet to check for connectivity (Figure 3.2(c), top) between the server and the mobile application. This fails because the NAT does not find a proper mapping of it's state variables in the flow.

The standard solution to this issue is to use a proxy server outside the NAT, to which the client establishes a long running connection. The proxy then forwards requests to the mobile, as shown in Figure 3.2(d), bottom. We check this configuration in two steps. First, we model opening a connection from the mobile client to the proxy. As the firewall allows this connection, the TCP flow has all the state variables pushed by the stateful devices: firewall, NAT and proxy server. In step two, the outside connectivity request is tunneled to

the mobile using the previous TCP flow as outer header. SymNet shows that connectivity is now possible: the flow has bindings describing an existing connection that can be matched by the firewall. The bindings of the source IP address and TCP source port point to the proxy. The true identity of the remote end is hidden by the proxy—rendering the firewall useless because it cannot filter blacklisted IP addresses.

Modeling middleboxes that change TCP sequence numbers. In this example we model common firewalls that randomize the initial sequence number of TCP connections, and modify all sequence and acknowledgment numbers afterwards. ISN randomization is done to protect vulnerable endpoint stacks that choose predictable sequence numbers against blind in-window injection attacks.

Does TCP still function correctly through such middleboxes? We first model the case when there is no middlebox (Figure 3.3(a)). Besides the regular TCP addresses, we also model sequence number and acks. After sending a segment with a sequence number, the host expects an ACK for that sequence number plus 1. When running reachability, we model this state by pushing a control variable called 'Expected-ACK' into the symbolic packet generated by A.

As this packet reaches B, the latter issues a new symbolic packet with the SYN/ACK flags set, containing the TCP and IP addresses from the original packet but with their values switched. The packet issued by B also includes a new value for SEQ representing B's initial sequence number and two new bindings: the ACK field and its own variable describing the Expected-ACK from its peer. We ommit the sequence number in the ACK packet from the figure to ease readability.

When A receives the SYN/ACK packet, it checks to see if the Expected-Ack matches the ACK received. If it does, A will generate the third ACK. Finally, B will match its Expected-ACK and the ACK variable. A match is found and the flow will hold state corresponding to the newly established connection. One can observe that the flow may hold state variables with the same meaning but corresponding to different entities, Expected-ACK in this case. A simple solution is to incorporate an identifier of the device within the variable name, for example: 'A-Expected-ACK'.

In Figure 3.3(b) we add a box that performs ISN randomization. This box will add some value to the SEQ field of packets in one direction and subtract it from the ACK in the reverse direction. After running the same tests as before, we can observe that TCP/IP connectivity is still possible: at each side the values of the symbolic variables Expected-ACK and ACK match.

Finally, suppose the return traffic $B - A$ does not pass through the middlebox. This time, the test fails: B sets the ACK field to 1 + SEQ, but SEQ has already been altered by the middlebox. A can not match Expected-ACK and ACK for validating the response.

Scalability. To validate our complexity analysis, in Figure 3.4 we plot the time needed to check increasingly large networks. The results confirm that SymNet checking time scales linearly with the size of the network.

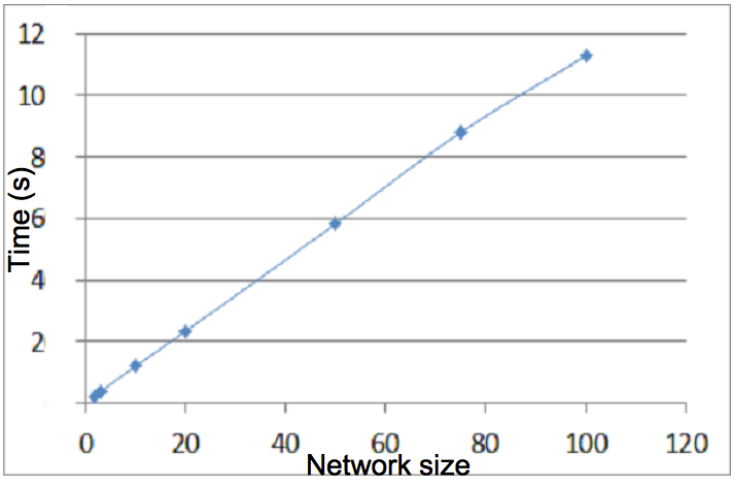


Figure 3.4: SymNet checking scales linearly

.

3.1.6 Conclusions

Middleboxes make networks difficult to debug and understand, and they are here to stay. To escalate the problems, recent industry trends advocate for software-only middleboxes that can be quickly instantiated and taken down. Existing tools to analyze networks do not model stateful middleboxes and do not capture even basic network properties.

In this paper we have shown that modeling stateful networks can be done in a scalable way. We model packet headers as variables and use symbolic execution to capture basic network properties; middlebox flow state can also be easily modeled using such header variables. We have proven our solution is correct and its complexity is linear.

We have implemented these algorithms in SymNet, a tool for checking stateful networks. SymNet takes middlebox descriptions written in Click together with the network topology and allows us to model a variety of use-cases. SymNet scales well, being able to check a network containing one hundred middleboxes in 11s.

3.2 Inter-Platform Connectivity (Tracebox)

3.2.1 Introduction

The TCP/IP architecture was designed to follow the end-to-end principle. A network is assumed to contain hosts implementing the transport and application protocols, routers implementing the network layer and processing packets, swit-ches operating in the datalink layer, etc. This textbook description does not apply anymore to a wide range of networks. Enterprise networks, WiFi hotspots, and cellular networks often include various types of middleboxes in addition to traditional routers and switches [149]. A *middlebox*, defined as “any intermediary box performing functions apart from normal, standard functions of an IP router on the data path between a source host and destination host” [43], manipulates traffic for purposes other than simple packet forwarding. Middleboxes are often deployed for performance or security reasons. Typical middleboxes include Network Address Translators, firewalls, Deep Packet Inspection boxes, transparent proxies, Intrusion Prevention/Detection Systems, etc.

Recent papers have shed the light on the deployment of those middleboxes. For instance, Sherry et al. [149] obtained configurations from 57 enterprise networks and revealed that they can contain as many middleboxes as routers. Wang et al. [165] surveyed 107 cellular networks and found that 82 of them used NATs. Although these middleboxes are supposed to be transparent to the end-user, experience shows that they have a negative impact on the evolvability of the TCP/IP protocol suite [75]. For example, after more that ten years of existence, SCTP [153] is still not widely deployed, partially because many firewalls and NAT may consider SCTP as an unknown protocol and block the corresponding packets. Middleboxes have also heavily influenced the design of Multipath TCP [75, 62].

Despite of their growing importance in handling operational traffic, middleboxes are notoriously difficult and complex to manage [149]. One of the causes of this complexity is the lack of debugging tools that enable operators to understand where and how middleboxes interfere with packets. Many operators rely on `ping`, `traceroute`, and various types of `show` commands to monitor their networks.

In this paper, we propose, validate, and evaluate `tracebox`. `tracebox` is a `traceroute` [160] successor that enables network operators to detect which middleboxes modify packets on almost any path. `tracebox` allows one to easily generate complex probes to send to any destination. By using the quoted packet inside of ICMP replies, it allows to identify various types of packet modifications and can be used to pinpoint where a given modification takes place.

The remainder of this paper is organized as follows: Sec. 3.2.2 describes how `tracebox` works and how it is able to identify middleboxes along a path. Sec. 3.2.3 analyses three use cases from a deployment of `tracebox` on PlanetLab. Sec. 3.2.4 shows how `tracebox` can be used to debug networking issues. Sec. 3.2.5 compares `tracebox` regarding state of the art. Finally, Sec. 3.2.6 concludes and discusses further work.

3.2.2 Tracebox

To detect middleboxes, `tracebox` uses the same incremental approach as `traceroute`, i.e., sending probes with increasing TTL values and waiting for ICMP `time-exceeded` replies. While `traceroute` uses this information to detect intermediate routers, `tracebox` uses it to infer the modification applied on a probe by an intermediate middlebox.

`tracebox` brings two important features.

Middleboxes Detection `tracebox` allows one to easily and precisely control all probes sent (IP header, TCP or UDP header, TCP options, payload, etc.). Further, `tracebox` keeps track of each transmitted packet. This permits to compare the quoted packet sent back in an ICMP `time-exceeded` by an intermediate router with the original one. By correlating the different modifications, `tracebox` is able to infer the presence of middleboxes.

Middleboxes Location Using an iterative technique (in the fashion of `traceroute`) to discover middleboxes also allows `tracebox` to approximately locate, on the path, where modifications occurred and so

the approximate middleboxes position.

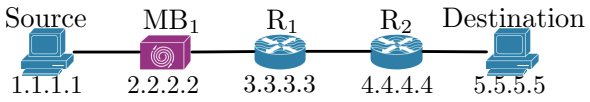
When an IPv4 router receives an IPv4 packet whose TTL is going to expire, it returns an ICMPv4 `time-exceeded` message that contains the offending packet. According to RFC792, the returned ICMP packet should quote the IP header of the original packet and the first 64 bits of the payload of this packet [131]. When the packet contains a TCP segment, these first 64 bits correspond to the source and destination ports and the sequence number. RFC1812 [32] recommended to quote the entire IP packet in the returned ICMP, but this recommendation has only been recently implemented on several major vendors' routers. Discussions with network operators showed that recent routers from Cisco (running IOX), Alcatel Lucent, HP, Linux, and Palo-Alto firewalls return the full IP packet. In the remainder of this paper, we use the term *Full ICMP* to indicate an ICMP message quoting the entire IP packet. We use the term RFC1812-compliant router to indicate a router that returns a *Full ICMP*.

By analyzing the returned quoted packet, `tracebox` is able to detect various modifications performed by middleboxes and routers. This includes changes in the Differentiated Service field and/or the Explicit Congestion Notification bits in the IP header, changes in the IP identification field, packet fragmentation, and changes in the TCP sequence numbers. Further, when `tracebox` receives a *Full ICMP*, it is able to detect more changes such as the TCP acknowledgement number, TCP window, removal/addition of TCP options, payload modifications, etc.

`tracebox` also allows for more complex probing techniques requiring to establish a connection and so multiple probes to be sent, e.g., to detect segment coalescing/splitting, Applica-tion-level Gateways, etc. In this case `tracebox` works in two phases: the *detection* and the *probing* phases. During the detection phase, `tracebox` sends probes by iteratively increasing the TTL until it reaches the destination. This phase allows `tracebox` to identify RFC1812-compliant routers. During the probing phase, `tracebox` sends additional probes with TTL values corresponding to the previously discovered RFC1812-compliant routers. This strategy allows `tracebox` to reduce its overhead by limiting the number of probes sent.

```
# tracebox -p 'IP / TCP / mss(9000)' -n 5.5.5.5
tracebox to 5.5.5.5 (5.5.5.5): 30 hops max
1: 3.3.3.3 TCP::SequenceNumber
2: 4.4.4.4 IP::TTL IP::CheckSum TCP::CheckSum TCP::SequenceNumber
           TCPOptionMaxSegSize::MaxSegSize
3: 5.5.5.5
}
```

Fig. 3.5(a) shows a simple network, where MB_1 is a middlebox that changes the TCP sequence number and the MSS size in the TCP MSS option but that does not decrement the TTL. R_1 is an old router while R_2 is a RFC1812-compliant router. The server always answer with a TCP reset. The output of running `tracebox` between "Source" and "Destination" is given by Fig. 3.5(b). The output shows that `tracebox` is able to



(a) topology

(b) output

Figure 3.5: `tracebox` example

effectively detect the middlebox interference but it may occur at a downstream hop. Indeed, as R_1 does not reply with a *Full ICMP*, `tracebox` can only detect the TCP sequence change when analyzing the reply of R_1. Nevertheless, when receiving the *Full ICMP* message from R_2, that contains the complete IP and TCP header, `tracebox` is able to detect that a TCP option has been changed upstream of R_2. At the second hop, `tracebox` shows additional modifications on top of the expected ones. The TTL and IP checksum are modified by each router and the TCP checksum modification results from the modification of the header.

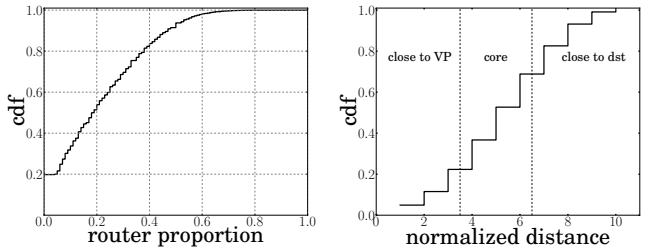
The detection of middleboxes depends on the reception of ICMP messages. If the router downstream of a middlebox does not reply with an ICMP message, `tracebox` will only be able to detect the change at a downstream hop similarly as the above example. Another limitation is that if the server does not reply with an ICMP (as in Fig. 3.5), then the detection of middleboxes in front of it is impossible.

`tracebox` is implemented in C++ in about 2,000 lines of code and embeds LUA [77] bindings to allow a flexible description of the probes as well to ease the development of more complex middlebox detection scripts. `tracebox` aims at providing the user with a simple and flexible way of defining probes without requiring a lot of lines of code. `tracebox` indeed allows to use a single line to define a probe (see as example the argument `-p` of `tracebox` in Fig. 3.5(b)) similarly to Scapy [146]. `tracebox` provides a complete API to easily define IPv4/IPv6 as well as TCP, UDP, ICMP headers and options on top of a raw payload. Several LUA scripts are already available and allows one to detect various types of middleboxes from Application-level Gateways to HTTP proxies. It is open-source and publicly available [52].

To verify the ability of `tracebox` to detect various types of middlebox interference, we developed several Click elements [90] modeling middleboxes. We wrote Click elements that modify various fields of the IP or TCP header, elements that add/remove/modify TCP options and elements that coalesce or split TCP segments. These elements have been included in a python library [73] that allows to easily describe a set of middleboxes and that generates the corresponding Click configuration. This library is used as unit tests to validate each new version of `tracebox`.

3.2.3 Validation & Use cases

In this section, we validate and demonstrate the usefulness of `tracebox` based on three use cases. We first explain how we deploy `tracebox` on the PlanetLab testbed (Sec. 3.2.3.1), next we asses the coverage of `tracebox` (Sec. 3.2.3.2) and finally discuss our use cases (Sec. 3.2.3.3, and 3.2.3.4).



(a) Proportion of RFC1812-compliant routers on a path

(b) Normalized distance from VP to RFC1812-compliant router

Figure 3.6: RFC1812-compliant routers

3.2.3.1 PlanetLab Deployment

We deployed `tracebox` on PlanetLab, using 72 machines as vantage points (VPs). Each VP had a target list of 5,000 items build with the top 5,000 Alexa web sites. Each VP used a shuffled version of the target list. DNS resolution was not done before running `tracebox`. This means that, if each VP uses the same list of destination names, each VP potentially contacted a different IP address for a given web site due to the presence of load balancing or Content Distribution Networks. Our dataset was collected during one week starting on April 17th, 2013.

In this short paper, we focus on analyzing some interferences between middleboxes and TCP. In theory, PlanetLab is not the best place to study middleboxes because PlanetLab nodes are mainly installed in research labs with unrestricted Internet access. Surprisingly, we noticed that seven VPs, from the 72 considered for the use cases, automatically removed or changed TCP options at the very first hop. They replaced the Multipath TCP [62], MD5 [72], and Window Scale [82] options with NOP and changed the value of the MSS option. We also found that two VPs always change the TCP Sequence number.

3.2.3.2 RFC1812-compliant routers

`tracebox` keeps track of each original packet sent and makes a comparison with the quoted IP packet when the ICMP `time-exceeded` message is received. Further, `tracebox` can potentially detect more middleboxes when routers return a *Full ICMP* packet. `tracebox`'s utility clearly increases with the number of RFC1812-compliant routers. Fig. 3.6 provides an insight of the proportion of RFC1812-compliant routers and their locations.

In particular, Fig. 3.6(a) gives the proportion of RFC1812-compliant routers (the horizontal axis) as a CDF. A value of 0, on the horizontal axis, corresponds to paths that contained no RFC1812-compliant router. On the other hand, a value of 1 corresponds to paths made only of RFC1812-compliant routers. Looking at Fig. 3.6(a), we observe that, in 80% of the cases, a path contains at least one router that replies with a *Full ICMP*. In other words, `tracebox` has the potential to reveal more middleboxes in 80% of the cases.

Fig. 3.6(b) estimates the position of the RFC1812-compliant routers in the probed paths. It provides the distance between the VP and the RFC1812-compliant routers on a given path. Note that, on Fig. 3.6(b), the X-Axis (i.e., the distance from the VPs) has been normalized between 1 and 10. Distances between 1 and

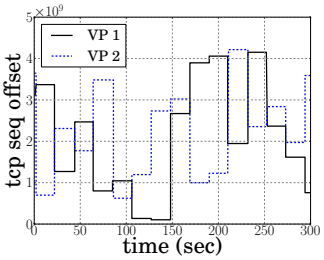


Figure 3.7: Time evolution of the TCP sequence number offset introduced by middleboxes

3 refer to routers close to the VP, 4 and 6 refer to the Internet core while, finally, distances between 7 and 10 refer to routers closer to the `tracebox` targets. The widespread deployment of RFC1812-compliant routers in the Internet core is of the highest importance since `tracebox` will be able to use these routers as "mirrors" to observe the middlebox interferences occurring in the access network [165].

Fig. 3.6(b) shows that for 22% of the paths, the RFC1812-compliant routers are close to the VP. This is approximatively the same proportion for routers close to `tracebox` targets. However, the majority of routers sending back a *Full ICMP* are located in the network core.

3.2.3.3 TCP Sequence Number Interference

The TCP sequence number is not supposed to be modified by intermediate routers. Still, previous measurements [75] showed that some middleboxes change sequence and acknowledgement numbers in the processed TCP segments. As the sequence number is within the first 64 bits of the TCP header, `tracebox` can detect its interference even though there are none RFC1812-compliant routers.

We analyze the output of `tracebox` from the 72 VPs. Our measurements reveal that two VPs always modify the TCP sequence numbers. The position of the responsible middlebox is close to the VP, respectively the first and third hop. We suspect that the middlebox randomizes the TCP sequence number to fix a bug in old TCP/IP stacks where the *Initial Sequence Number* (ISN) was predictable [110].

When used on a path that includes such a middlebox, `tracebox` can provide additional information about the sequence number randomization. Depending on the type of middlebox and the state it maintains, the randomization function can differ. To analyze it, we performed two experiments. First, we generated SYN probes with the same destination (IP address and port), the same sequence number, and different source ports. `tracebox` revealed that the middlebox modified all TCP sequence numbers as expected. A closer look at the modified sequence numbers revealed that the difference between the ISN of the probe and the randomized sequence number can be as small as 14510 and as large as 4294858380 (which corresponds to a negative difference of 108916 when the 32 bits sequence number space wrap). Our measurements show that these differences appear to be uniformly distributed for the different source ports.

For our second experiment, we used `tracebox` to verify how the randomization evolves over time. For this, we sent SYN probes using the same 5-tuple and different ISN during five minutes and evaluated the evolution of the TCP sequence number modifications. Fig. 3.7 shows the offset between the sent ISN and

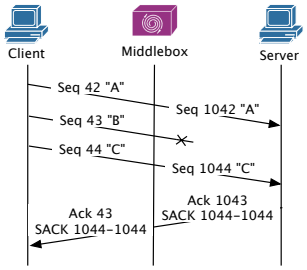


Figure 3.8: Example of invalid SACK blocks generated due to a middlebox.

the randomized one for two different 5-tuples. `tracebox` reveals that the two middleboxes seem to change their randomization approximatively every 20 seconds. This suggests stateful middleboxes.

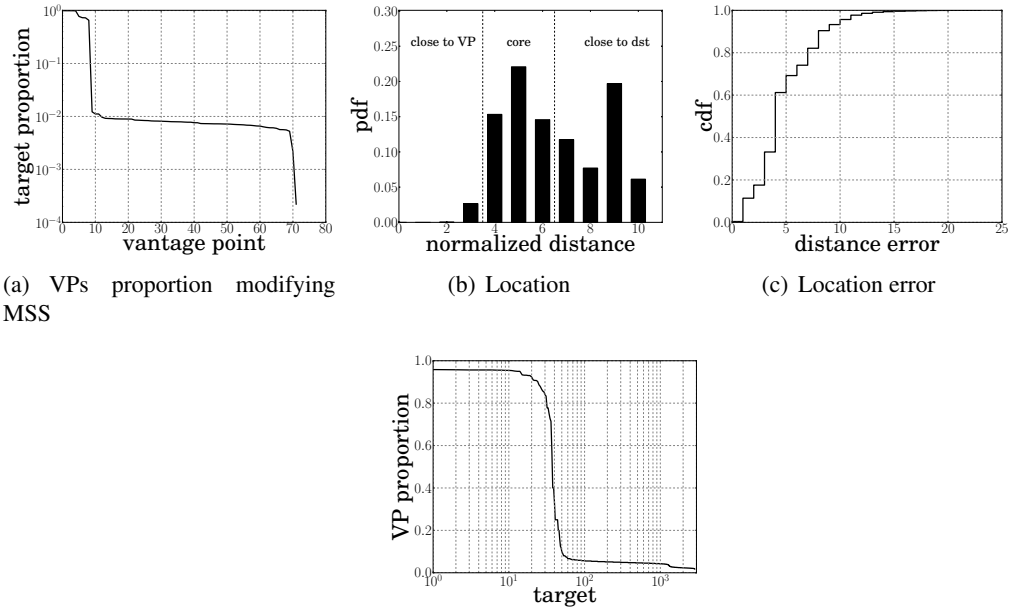
As explained by Honda et al. [75], changing the TCP sequence numbers has an impact on the TCP protocol evolvability. Unfortunately, it has also an impact on the utilization of widely deployed TCP extensions. Consider the TCP *Selective Acknowledgement* (SACK) option [108]. This TCP option improves the ability of TCP to recover from losses. One would expect that a middlebox changing the TCP sequence number would also update the sequence numbers reported inside TCP options. This is unfortunately not true, almost 18 years after the publication of the RFC [108]. We used `tracebox` to open a TCP connection with the SACK extension and immediately send SACK blocks. `tracebox` reveals that the middlebox changes the sequence number but does not modify the sequence number contained in the SACK block.

Fig. 3.8 shows the behavior of such a middlebox on the TCP sequence number and SACK blocks. In this scenario, the middlebox increases the TCP sequence number by 1,000 bytes causing the client to receive a SACK block that corresponds to a sequence number that it has not yet transmitted. This SACK block is invalid, but the acknowledgement is valid and correct. For the receiver, it may not be easy to always detect that the SACK information is invalid. The receiver may detect that the SACK blocks are out of the window, but the initial change may be small enough to carry SACK blocks that are inside the window.

If we know that a SACK block is invalid, algorithms that use SACK should understand that the SACK option does not give more information than a simple acknowledgment. In this view, such algorithms should have at least the same performance as they would have if SACK was not used at all. Unfortunately, this is not the case as the Linux TCP stack does not consider duplicate acknowledgment when SACK is enabled. When the offset is small the SACK blocks are potentially in-window. In this case the Linux TCP stack reacts correctly. However, when the SACK blocks are out-of-window then the TCP stack has to wait for a complete RTO instead of doing fast-retransmit. We performed a small measurement in a controlled environment and observed up to a 50% drop in performance with a large offset [124].

3.2.3.4 TCP MSS Option Interference

Our third use case for `tracebox` concerns middleboxes that modify the TCP MSS option. This TCP option is used in the SYN and SYN+ACK segments to specify the largest TCP segment that a host sending the option can process. In an Internet that respects the end-to-end principle, this option should never be modified. In the



(a) VPs proportion modifying MSS

(b) Location

(c) Location error

(d) Targets proportion observing an MSS modification

Figure 3.9: MSS option modification

current Internet, this is unfortunately not the case. The main motivation for changing the TCP MSS option on middleboxes is probably to fix some issues caused by other middleboxes with Path MTU Discovery [113]. On top of changing the MSS option, we also discovered middleboxes, in a couple of ISPs, that add the option if it is missing.

Path MTU Discovery is a technique that allows a host to dynamically discover the largest segment it can send without causing IP fragmentation on each TCP connection. For that, each host sends large segments inside packets with the `Don't Fragment` bit set. If a router needs to fragment the packet, it returns an ICMP `destination-unreachable` (with code "Packet fragmentation is required but the 'don't fragment' flag is on") back to the source and the source updates its segment size. Unfortunately, some routers do not return such ICMP messages [109] and some middleboxes (e.g., NAT boxes and firewalls) do not correctly forward the received ICMP message to the correct source. MSS clamping mitigates this problem by configuring middleboxes to decrease the size reported in the MSS option to a smaller MSS that should not cause fragmentation.

We use our dataset to identify middleboxes that modify the MSS option in SYN segments. Fig. 3.9(a) provides, for each VP (the horizontal axis), the proportion of paths (the vertical axis, in log-scale) where the MSS option has been changed. We see that a few VPs encountered at least one MSS modification on nearly all paths while, for the vast majority of VPs, the modification is observed in only a couple of paths. We decided to remove those VPs from our data set for further analyses, meaning that only 65 VPs were finally considered for the use case.

Similarly to Fig. 3.9(a), Fig. 3.9(d) provides, for each target, the proportion of paths affected by an MSS modification. We see about ten targets that have a middlebox, probably their firewall or load balancer, always

changing the MSS option. In the same fashion as the VPs that changed the MSS option, they also removed the Multipath TCP, MD5 and Window Scale options.

Fig. 3.9(b) indicates where, in the network, the MSS option is modified. In the fashion of Fig. 3.6(b), the distance from VP has been normalized between 1 and 10, leading to the rise of three network regions (i.e., close to VP, core, and close to targets). As shown by Fig. 3.9(b), `tracebox` can detect the MSS modification very close to the source (2.7% of the cases) while this detection mostly occurs in the network core (52% of the cases).

Remind that this distance does not indicate precisely where is actually located the middlebox responsible for the MSS modification. Rather, it gives the position of the router that has returned a *Full ICMP* and, in this ICMP packet, the quoted TCP segment revealed a modification of the MSS field. Actually, the middlebox should be somewhere between this position and the previous router (on that path) that has also returned a *Full ICMP* (or the VP if it was the very first *Full ICMP* on that path).

Fig. 3.9(c) refines our location of MSS modification by taking this aspect (i.e., the middlebox is somewhere on the path between the modification detection and the previous RFC1812-compliant router) into account. It gives thus an estimation of middlebox location error. This error is simply obtained by subtracting the distance at which `tracebox` reveals the modification and the distance at which the previous RFC1812-compliant router was detected by `tracebox` on that path. Obviously, lower the error, more accurate the location given in Fig. 3.9(b). On Fig. 3.9(c), we see that in 61% of the cases, the location estimation error is below (or equal to) four hops. All errors above 13 hops, that represents the length of around 60% of the paths, are uncommon (less than 1% each).

3.2.4 Discussion

In Sec. 3.2.3, we showed that `tracebox` can provide a useful insight on known middleboxes interference. We believe that `tracebox` will also be very useful for network operators who have to debug strange networking problems involving middleboxes. While analyzing the data collected during our measurement campaign (see Sec. 3.2.3.1), we identified several strange middlebox behaviors we briefly explain in this section. We also discuss how `tracebox` can be used to reveal the presence of proxies and network address translators (NATs).

3.2.4.1 Unexpected Interference

We performed some tests with `tracebox` to verify whether the recently proposed Multipath TCP [62] option could be safely used over the Internet. This is similar to the unknown option test performed by Honda et al. [75]. However, on the contrary to Honda et al., `tracebox` allows one to probe a large number of destinations. To our surprise, when running the tests, `tracebox` identified about ten Multipath TCP servers based on the TCP option they returned. One of those server, `www.baidu.com`, belongs to the top 5 Alexa. All these servers where located inside China. A closer look at these options revealed that these servers (or their load balancers) simply echo a received unknown TCP option in the SYN+ACK. This is clearly an

incorrect TCP implementation.

3.2.4.2 Proxy Detection

`tracebox` can also be used to detect TCP proxies. To be able to detect a TCP proxy, `tracebox` must be able to send TCP segments that are intercepted by the proxy and other packets that are forwarded beyond it. HTTP proxies are frequently used in cellular and enterprise networks [165]. Some of them are configured to transparently proxy all TCP connections on port 80. To test the ability of detecting proxies with `tracebox`, we used a script that sends a SYN probe to port 80 and, then, to port 21. If there is an HTTP proxy on the path, it should intercept the SYN probe on port 80 while ignoring the SYN on port 21. We next analyze the ICMP messages returned.

From our simple PlanetLab deployment, we identified two oddities. First, we found an HTTP proxy or more probably an IDS within a National Research Network (SUNET) as it only operated for a few destinations and that the path for port 80 was shorter than for port 21. Second, and more disturbing, `tracebox` identified that several destinations where behind a proxy whose configuration, inferred from the returned ICMP messages, resulted in a forwarding loop for probes that are not HTTP. We observed that the SYN probe on port 21, after reaching the supposed proxy, bounced from one router to the other in a loop as `tracebox` received ICMP replies from one router then another alternatively.

3.2.4.3 NAT Detection

NATs are probably the most widely deployed middleboxes. Detecting them by using `tracebox` would likely be useful for network operators. However, in addition to changing addresses and port numbers of the packets that they forward, NATs often also change back the returned ICMP message and the quoted packet. This implies that, when inspecting the received ICMP message, `tracebox` would not be able to detect the modification.

This does not prevent `tracebox` from detecting many NATs. Indeed, most NATs implement *Application-level Gateways* (ALGs) [123] for protocols such as FTP. Such an ALG modifies the payload of forwarded packets that contain the `PORT` command on the `ftp-control` connection. `tracebox` can detect these ALGs by noting that they do not translate the quoted packet in the returned ICMP messages. This detection is written as a simple script (shown in Fig 3.10) that interacts with `tracebox`. It builds and sends a SYN for the FTP port number and, then, waits for the SYN+ACK. The script makes sure that the SYN+ACK is not handled by the TCP stack of the host by configuring the local firewall (using the filter functionality, shown at line 7, of `tracebox` that configures `iptables` on Linux and `ipfw` on Mac OS X). It then sends a valid segment with the `PORT` command and the encoded IP address and port number as payload. `tracebox` then compares the transmitted packet with the quoted packet returned inside an ICMP message by an RFC1812-compliant router and stores the modification applied to the packet. If a change occurs and a call-back function has been passed as argument, `tracebox` triggers the callback function. In Fig 3.10, the callback `cb` checks whether there has been a payload modification. If it is the case a message showing the

```
[numbers=left,fontsize=\scriptsize]
-- NAT FTP detection
-- To run with: tracebox -s <script> <ftp_server>
-- Build the initial SYN (dest is passed to tracebox)
syn = IP / tcp{dst=21}
-- Avoid the host's stack to reply with a reset
fp = filter(syn)
synack = tracebox(syn)
if not synack then
print("Server did not reply...")
fp:close()
return
end
-- Check if SYN+ACK flags are present
if synack:tcp():getflags() ~= 18 then
print("Server does not seems to be a FTP server")
fp:close()
return
end
-- Build the PORT probe
ip_port = syn:source():gsub("%.", ",")
data = IP / tcp{src=syn:tcp():getsource(), dst=21,
seq=syn:tcp():getseq()+1,
ack=synack:tcp():getseq()+1, flags=16} /
raw('PORT '.. ip_port .. ',189,68\r\n')
-- Send probe and allow cb to be called for each reply
function cb(ttl, rip, pkt, reply, mods)
if mods and mods:__tostring():find("Raw") then
print("There is a NAT before " .. rip)
return 1
end
end
tracebox(data, {callback = "cb"})
fp:close()
```

Figure 3.10: Sample script to detect a NAT FTP.

approximate position of the ALG on the path is printed (see line 29).

3.2.5 Related Work

Since the end of the nineties, the Internet topology discovery has been extensively studied [57, 69]. In particular, `traceroute` [160] has been used for revealing IP interfaces along the path between a source and a destination. Since then, `traceroute` has been extended in order to mitigate its intrinsic limitations. From simple extensions (i.e., the types of probes sent [158, 101]) to much more developed modifications. For instance, `traceroute` has been improved to face load balancing [30] or the reverse path [84]. Its probing speed and efficiency has also been investigated [58, 35, 37].

To the best of our knowledge, none of the available `traceroute` extensions allows one to reveal middlebox interference along real Internet paths as `tracebox` does.

Medina et al. [109] report one of the first detailed analysis of the interactions between transport protocols and middleboxes. They rely on active probing with `tbit` and contact various web servers to detect whether Explicit Congestion Notification (ECN) [137], IP options, and TCP options can be safely used. The `TCPExposure` software developed by Honda et al. [75] is closest to `tracebox`. It also uses specially crafted packets to test for middlebox interference.

Wang et al. [165] analyzed the impact of middleboxes in hundreds of cellular networks. This study revealed various types of packet modifications. These three tools provide great results, but they are limited to specific paths as both ends of the path must be under control. This is a limitation since some middleboxes are configured to only process the packets sent to specific destination or ports. On the contrary, `tracebox` does not require any cooperation with the service. It allows one to detect middleboxes on any path, i.e., between a source and any destination. Our measurements reveal middleboxes that are close to the clients but also close to the server.

Sherry et al. [149] have relied on network configuration files to show the widespread deployment of middleboxes. Still, their study does not reveal the impact of these middleboxes on actual packets.

3.2.6 Conclusion

Middleboxes are becoming more and more popular in various types of networks (enterprise, cellular network, etc.). Those middleboxes are supposed to be transparent to users. It has been shown that they frequently modify packets traversing them, sometimes making protocols useless. Further, due to the lack of efficient and easy-to-use debugging tools, middleboxes are difficult to manage.

This is exactly what we tackled in this paper by proposing, discussing, and evaluating `tracebox`. `tracebox` is a new extension to `traceroute` that allows one to reveal the presence of middleboxes along a path. It detects various types of packet modifications and can be used to locate where those modifications occur. We deployed it on the PlanetLab testbed and demonstrated its capabilities by discussing several use cases. `tracebox` is open-source and publicly available [52].

`tracebox` opens new directions to allow researchers to better understand the deployment of middleboxes in the global Internet. In the coming months, we plan to perform large-scale measurement campaigns to analyze in more details middlebox interferences in IPv4 and IPv6 networks. `tracebox` could also be extended to fingerprint specific middleboxes.

3.3 Inter-Platform Signaling

The inter-platform signaling prototype released by WP4 has been integrated and tested into the CHANGE testbed. The latter activities, executed in the scope of WP5, further improved the inter-platform signaling prototype in the following ways:

- improvements to the Signaling Manager / Platform Controller integration within the CHANGE Platform;

- improvements to the software configuration and building framework;
- Improvements to the components stability;
- Improvements to the Service Manager User Interface (UI).

Most of the software refinements and bug-fixes that have been applied do not introduce either major updates to the signaling framework design or additional functionalities. Therefore, their detailed description has not been included into this document that only presents the major noticeable differences with respect to [9] and [10].

The following sections present the final building, installation and configuration instructions that application developers have to follow in order to deploy the framework components into their CHANGE Domains/testbeds. These sections refer to CHANGE architectural and functional concepts without explaining them in depth. Details are provided by specific documents delivered by WP2 [4], WP3 [5], and WP4 ([7], [8], [9] and [10]).

3.3.1 Components configuration

The inter-platform signaling framework components (i.e. Service Manager, Signaling Manager and Service Broker) are bundled together in the same software package. The package relies on the autotools framework for its configuration and building functionalities [9]. It ships the `configure` scripts that allows to automatically retrieve and opportunely configure the tools required to build the software in the target system.

The `configure` script defaults to build and install all the signaling framework components (i.e. Signaling Manager, Service Manager and Service Broker) into the target machine and allows their opt-out, via configuration options. The following options allow the removal of one or more software components from the default set:

- `--disable-service-uni-support`: Disables the Service Manager component;
- `--disable-internal-nni-support`: Disables the Signaling Manager component;
- `--disable-inter-as-nni-support`: Disables the Service Broker component.

Therefore, application developers - and signaling framework adopters in general - are able to select only the components required, depending on the particular deployment of the CHANGE domains/testbeds they want to obtain (Service Manager co-located with one of the Signaling Managers, Service Manager and Service Broker colocated, each component deployed in a different Network Element (NE) etc.).

3.3.2 Signaling Manager

The rule of thumb that must be fulfilled for a correct CHANGE testbed/domain deployment is that each Signaling Manager must be co-located with the corresponding Platform Controller. As described in [9] and [10], the Signaling Manager `signald` couples with the Platform Controller - using the Platform API [6] to

issue allocation requests, obtain replies etc. - while the `transportd`, is in charge of transparently exchange messages with the peering platforms. While the latter only requires a working Control INTerface (CINTF) where to exchange messages with its signaling adjacencies, the former couples to the Platform Controller directly, using the Platform API as described in [6]. The Platform API can only be invoked locally on the platform and therefore the Signaling Manager has to be installed on the same machine where the APIs have to be invoked (i.e. the Platform Controller host).

Particular attention must be paid during Control Plane setup. The CINTFs, representing the interfaces where the signaling messages are exchanged, have to be reachable between Flow Processing Route (FPR) peers. The Signaling Managers will be forming signaling adjacencies over these CINTFs to finally exchange signaling messages.

3.3.2.1 Extending the Signaling Manager

The Signaling Manager code base has been consolidated with the aims for both flexibility and easiness for extensions, to allow application developers rapidly develop their extensions. The choice for Python as the language for the Signaling Application (as the top element in the layers splitting exposed in [9]) aims toward the mentioned goals. Therefore, the Signaling Manager code base has been re-organized enforcing the following principles:

- Functionalities are grouped per-file, such as:
  - `service.py`: It contains the whole protocol FSM. Application developers can change the pre and post-transitions behaviors easily, either by simply adding their code into the hooked functions (e.g. for Routing/Service helping, Notifications management) or wrapping the transition functions;
  - `resourced.py`: It contains the code gluing the Signaling Manager with the Platform APIs. System integrators can easily integrate the Signaling Manager with non-FlowStream compliant platforms.
- Functionalities can be overridden: All the functionalities of the Signaling Manager are contained into derived classes and their behaviors can be changed either in the base classes or in their derivates.
- All the functionalities are scaffolded and follow well defined object oriented principles (Private class data, publisher/subscribers, Singletons, Façades, Containers etc.)
- External executables have been wrapped: the Signaling Manager has been extended to run external executables with wrapping scripts, in order to enhance the overall extendability even more. The following wrappers are supported:
  - The Platform API wrapper: This external executable wraps the Platform API script (i.e. the `flowstream_exec.py`, [6]) and decouples the Signaling Manager from the Platform.

- The Notification wrapper: This external executable allows to hook external code to the Notify message reception event. The notification wrapper gets called by the Signaling Manager once a Notify message reaches the `signald`. The wrapper obtains the Service Identifier (SID), Platform Identifier (PID) as well as the FPR and can elaborate and finally present a reply to the Notify message received.
- The Routing and Service helper: This external executable allows for routing completions or service completions. The wrapper gets called when the Platform Flow Processing Route Object (FPRO), contained into the received FPR, is incomplete.

3.3.2.2 Building and installation

The Signaling Manager building and installation steps can be summarized as follows:

(i) Type `./configure --disable-service-uni-support --disable-inter-as-nni-support` into the root directory of the signaling framework sources, to setup only the Signaling Manager component with the necessary NNI support (i.e. Internal-NNI)

(ii) Type `make` to compile the package

(iii) Finally, type `make install` to install the component and its related data files

Please note that the package is assumed to be installed into system-wide directories in its final deployment into the testbed. Therefore, the `--prefix` option described in [9] has not to be used (assuming the defaults values).

3.3.2.3 Configuration and application integration

The Signaling Manager only requires configuration values that are used during its initialization phases. These values can be provided to the Signaling Manager by editing its configuration file (i.e. `/etc/sigmgr.conf`). This file has an INI-like format [13]: The configuration-entries are described as key/value pairs. Multiple entries are grouped into sections in order to keep the file human-readable.
In order to setup the Signaling Manager into each platform, the following entries have to be properly configured into the configuration file:

```
...
[change-nslp]
change-nslp-api-port = 50000
platform-id          = "<platform-name>"
...
[signald]
service-api-addr     = "<CINTF-IP>"
service-api-port     = 50001
```

```
sigmgr-api-addr      = "<CINTF-IP>"
sigmgr-api-port      = 50002
...
plugin-platform      = "<path-to-exec>"
#plugin-notify       = "<path-to-exec>"
...
```

Where:

- `platform-id`: is the unique name associated to the platform (i.e. PID)
- `service-api-addr` and `service-api-port`: the pair represents the Service-UNI end-point of the platform. The address must be one of the CINTFs available on the platform
- `sigmgr-api-addr` and `sigmgr-api-port`: this pair represents the Internal-NNI end-point of the platform. The address must be one of the CINTFs available on the platform
- `plugin-platform`: specifies the path of the Platform wrapper
- `plugin-notify`: specifies the path of the Notification wrapper

3.3.2.4 Execution

Once the Signaling Manager is properly configured, it can be executed with the following command:

- `sigmgr start`

The Signaling Manager will detach itself from the console, running as a daemon in the system. To stop its execution, the following command must be issued:

- `sigmgr stop`

During its execution, the Signaling Manager `signald` and `transportd` components will append their logs to the `/var/signald.log` and `/var/transportd.log` files respectively [9].
Please note that the Signaling Manager accepts other options than the ones presented in this document. Refer to its help for their complete list.

3.3.3 Service Manager

3.3.3.1 Building and installation

The same instructions previously described for the Signaling Manager building and installation can be followed for the Service Manager, with only slight changes:

(i) Type `./configure --disable-inter-as-nni-support` into the root directory of the signaling framework sources, to setup only the Service Manager component with the necessary NNI support (i.e. Service-NNI and Internal-NNI)

(ii) Type `make` to compile the package

(iii) Finally, type `make install` to install the component and its related data file

Please note that the package is assumed to be installed into system-wide directories in its final deployment into the testbed and therefore the `--prefix` option described in [9] has not to be used (assuming the defaults values).

3.3.3.2 Configuration

The Service Manager must be configured by editing the `/etc/svcmgrd.conf` file, in order to provide the configuration values for its `signald` layer [9]. The file format is the same as for the Signaling Manager configuration file

3.3.3.3 Execution

Once the Service Manager is properly configured, it can be executed with the following command:

- `svcmgrd`

The commands can be issued either to the Service Manager directly, using the interactive Command Line Interface (CLI), or passed as an external file. In the former case, the Service Manager will start the REP Loop (REPL) of its CLI based UI and it will be waiting for new commands. In the latter case, the Service Manager will be reading and executing the commands from the external file, in the order they appear.
In order to execute commands from an external file, the following invocation syntax must be used:

- `svcmgrd -k <command-file>`

Please note that the Service Manager accepts other options than the ones presented. Refer to its help for their complete list.

3.3.3.4 The updated Service Manager CLI

The Service Manager UI has been updated since [10]. Refinements have been applied in order provide to the signaling framework users greater decoupling between declarations and commands towards providing an user interface that is both flexible and easy to use.
This section presents the up-to-date Service Manager CLI commands, summarized in the following table. The table only describes the major commands available and their meanings. Refer to the Service Manager CLI help for further details.

Command	Meaning
action-attract-dns	Adds an Attract-DNS action into the FPR
action-delete	Deletes a previously defined action
action-filter	Adds a Filter action into the FPR
action-forward	Adds a Forward action into the FPR
action-redirect	Adds a Redirect action into FPR
action-reroute	Adds a Reroute action into FPR
action-tun-downstream	Defines a downstream Tunnel End-Point (TEP)
action-tun-upstream	Defines an upstream TEP
configuration-erase	Deletes ALL configuration parameters from the Service Manager data-model (e.g. PMs, interfaces, messages)
fpr-add-actions	Adds action(s) to an existing FPR
fpr-build	Builds a FPR placeholder
fpr-create	Creates a FPR, filling it up using a previously created placeholder
fpr-delete	Deletes a FPR
fpr-show	Shows FPR information
interface-add	Adds an interface or a port to an existing PM (the interface-type parameter will be used as discriminant)
interface-config-string	Configures an interface, by using an user-provided string
interface-del	Deletes an interface (external interface or port)
interface-show	Shows an interface configuration
notify-build	Builds a Notify message placeholder
notify-create	Creates a Notify message, filling it up from a previously created placeholder
notify-delete	Deletes a Notify message
notify-send	Sends Notify message
platform-add	Adds a PM to the Service Manager data-model
platform-config-connection	Configures a platform with inter-PMs connections
platform-config-interface	Configure a platform with (external) interfaces
platform-config-pm	Configure a platform with specific PM(s)
platform-del	Deletes a platform from the Service Manager data-model
platform-show	Shows the current platforms configuration
pm-add	Adds a PM to the Service Manager data-model
pm-config-file	Configures a PM, using the contents of a file
pm-config-string	Configures a PM, using the contents of string
pm-connect	Specifies connections between PMs
pm-del	Deletes a PM from the data-model
pm-show	Show the current PMs configuration
sdreq-build	Builds Service Deletion Request message (placeholder)
sdreq-create	Creates a Service Deletion Request message
sdreq-delete	Deletes of a Service Deletion Request message
sdreq-send	Sends a previously create Service Deletion Request message (i.e. using the sdreq-create command)
service-processing	Processes an input file by using the CHANGE processing
ssreq-build	Builds Service Setup Request message (placeholder)
ssreq-create	Creates a Service Setup Request message
ssreq-delete	Deletes a Service Setup Request message
ssreq-send	Sends a previously created Service Setup Request message (i.e. using the ssreq-create command)

3.3.4 A service provisioning example

The following example presents a simple service, deployed among three different platforms (i.e. `platform0`, `platform1` and `platform2`). The example employs the updated UI described in the previous section as well as the tunnel functionalities introduced in [10].

```
#
# Commands for the Service Manager
#

# Define the interfaces that will be used later on. Interfaces are
# either internal or external. External interfaces are bound
# to real NICs, internal ones represent PM ports instead

# External interfaces (real intfs)
interface-add external intfext1 eth0 FromNetFront ingress 192.168.1.1
interface-add external intfext2 eth0 ToNetFront egress 192.168.1.2
interface-add external intfext3 eth0 FromNetFront ingress 192.168.1.3
interface-add external intfext4 eth1 ToNetFront egress 192.168.1.4
interface-add external intfext5 eth0 FromNetFront ingress 192.168.1.5
interface-add external intfext6 eth1 ToNetFront egress 192.168.1.6

# Internal interfaces (ports)
interface-add internal interface1 eth3 virtual ingress 10
interface-add internal interface2 eth4 virtual egress 20
interface-add internal interface3 eth5 virtual ingress 30
interface-add internal interface4 eth6 virtual egress 40
interface-add internal interface5 eth3 virtual ingress 50
interface-add internal interface6 eth4 virtual egress 60
interface-add internal interface7 eth6 virtual ingress 70
interface-add internal interface8 eth5 virtual egress 80
interface-add internal interface9 eth6 virtual ingress 90
interface-add internal interface10 eth5 virtual egress 100

# Define the processing modules (PMs) by defining their images, constraints and
# ports
pm-add firewall1 image1 Firewall True constraints interface1 interface2
pm-add mirror2   image2 Mirror True constraints interface3 interface4
pm-add queue3    image1 SimpleQueue True constraints interface5 interface6
pm-add firewall2 image1 Firewall True constraints interface7 interface8
pm-add firewall3 image1 Firewall True constraints interface9 interface10
```

```
# Configure, with additional data, the external interfaces declared
# previously
interface-config-string intfext1 Eth0
interface-config-string intfext2 MACADDR
interface-config-string intfext3 MACADDR, BURST 10
interface-config-string intfext4 Eth4
interface-config-string intfext5 MACADDR
interface-config-string intfext6 Eth6

# Configure the previously declared PMs
pm-config-string firewall1 http port 8080
pm-config-string mirror2   udp  port 10000
pm-config-string queue3    8012
pm-config-string firewall2 udp  port 4600
pm-config-string firewall3 http port 8080

# Connects the PMs together (and to external interfaces as well)
pm-connect external conn1 intfext1 interface1
pm-connect internal conn2 interface1 interface2 firewall1
pm-connect internal conn3 interface2 interface3
pm-connect internal conn4 interface3 interface4 mirror2
pm-connect internal conn5 interface4 interface5
pm-connect internal conn6 interface5 interface6 queue3
pm-connect external conn7 interface6 intfext2

pm-connect external conn8 intfext3 interface7
pm-connect internal conn9 interface7 interface8 firewall2
pm-connect external conn10 interface8 intfext4

pm-connect external conn11 intfext5 interface9
pm-connect internal conn12 interface9 interface10 firewall3
pm-connect external conn13 interface10 intfext6

#
```

```
# Configure the Platforms involved in the service
#

# Add a platform
platform-add platform0 10.0.2.163 50002

# Configure its (external) interfaces
platform-config-interface platform0 intfext1 intfext2

# Configure its PMs
platform-config-pm platform0 firewall1 mirror2 queue3

# And the inter-PMs connections
platform-config-connection platform0 conn1 conn2 conn3 conn4 conn5 conn6 conn7

# Repeat for platform1
platform-add platform1 10.0.2.213 50002
platform-config-interface platform1 intfext3 intfext4
platform-config-pm platform1 firewall2
platform-config-connection platform1 conn8 conn9 conn10

# And for platform2 as well
platform-add platform2 10.0.2.238 50002
platform-config-interface platform2 intfext5 intfext6
platform-config-pm platform2 firewall3
platform-config-connection platform2 conn11 conn12 conn13

# Create the FPR placeholder
fpr-create test

# Fill the FPR with the platforms (they will automatically inherit
# the previously declared bindings and configurations)
fpr-build test platform0 platform1 platform2

# Define a tunnel, declaring its endpoints
```

```
action-tun-upstream tun1up tun1down intfext3 intfext2
action-tun-downstream tun1down tun1up intfext2 intfext3

# Define another tunnel
action-tun-upstream tun2up tun2down intfext5 intfext4
action-tun-downstream tun2down tun2up intfext4 intfext5

# Add the tunnels into the data-model
# previously defined)
fpr-add-actions test platform0 tun1down
fpr-add-actions test platform1 tun1up
fpr-add-actions test platform1 tun2down
fpr-add-actions test platform2 tun2up

# The following commands will create a Service-Setup-Request, binding
# all the previously defined (and partially binded) information
# altogether in a single FPR
ssreq-create 1
ssreq-build 1 1 99 fid1 10.0.2.163 test

# That will be sent out with the following command
ssreq-send 1 NOME 10.0.2.163 50001 60

# The following commands create a Service-Deletion-Request
sdreq-create del1
sdreq-build del1 1 99 fid1 10.0.2.163

# And finally the message previously built is sent out
sdreq-send del1 NOME2 10.0.2.163 50001 500
```

4 The CHANGE Applications

4.1 Overview

This section introduces the various applications that were developed for the CHANGE platform. Since the main platform implementation has centered around ClickOS, Click has become the de-facto development "language" for these applications. We begin in section 4.2 with a description an implementation of a number of middleboxes run within ClickOS virtual machines: a software Broadband Remote Access Server (BRAS), a load balancer, a carrier-grade NAT and a flow monitor, among others. Then, in section 4.3 we introduce a number of applications developed directly on Click (i.e., not ClickOS) in order to show that the platform is not dependent on a single technology; these include a DPI, an IDS, an a firewall/policy enforcer. It is worth noting that, because these were developed for Click, it would not require much work to port them to ClickOS. Finally, section 4.4 describes NetPaaS, a system that allows better content delivery by enabling CDN providers and ISPs to collaborate. This work is an instance of a more general trend towards deployment of so-called micro data-centers, whereby ISPs install hardware in their operational networks at different Points of Presence (POPs), and let third parties run software on it (e.g., content caches). While this provides new avenues for revenue, letting third-party software loose in an operational network could lead to security breaches or undesired behavior. The Symnet tool described previously in this document, along with the ClickOS platform, provide some of the building blocks towards making this new paradigm a reality.

4.2 ClickOS Applications Implementation

In the previous section we presented an evaluation of ClickOS' basic performance. We now turn our attention to finding out how it fares when running actual middleboxes. Clearly, middleboxes cover a wide range of processing, so it would be impossible to be exhaustive. However, by presenting results from several different middleboxes we aim to give a good idea of how ClickOS would perform under different types of workloads. For these set of tests we use two of our low-end servers connected via two direct cables, one per pair of Ethernet ports. One of the servers generates packets towards the other server, which runs them through a ClickOS middlebox and forwards them back towards the first server where their rate is measured. The ClickOS VM is assigned a single CPU core, with the remaining three are given to dom0. We test each of the following middleboxes in turn:

Wire A simple "middlebox" which sends packets from its input to its output interface. This is configuration serves to give a performance baseline.

EtherMirror Like wire, but also swap the Ethernet source and destination fields.

IP router A standards-compliant IPv4 router configured with a single rule.

Firewall Based on the `IPFilter` element and configured with ten rules, that do not match incoming packets.

Carrier Grade NAT An almost standards-compliant carrier-grade NAT. To stress the NAT, each packet has

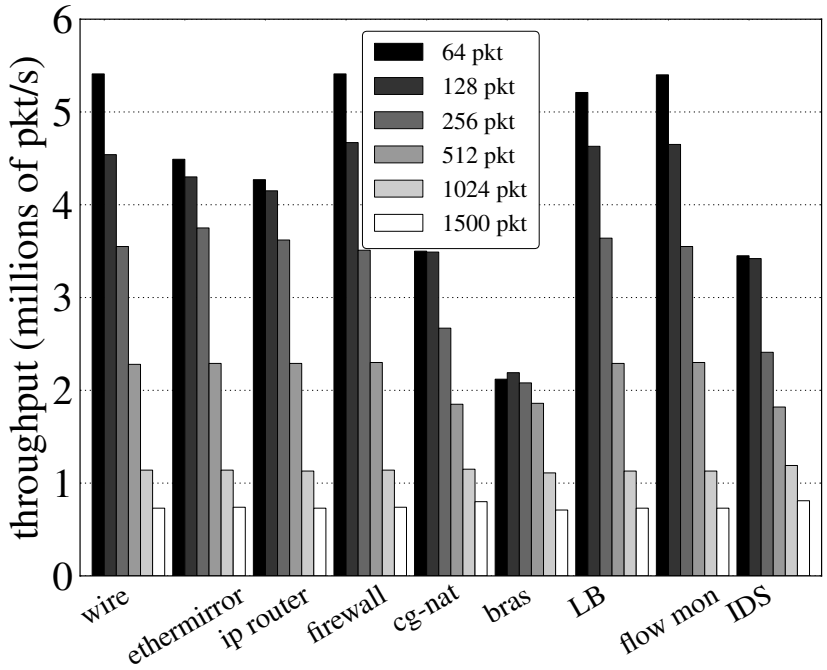


Figure 4.1: Performance for different ClickOS middleboxes and packet sizes using a single CPU core.

a different set of source and destination port numbers. Using a single flow/set of ports results in a higher rate of 5.1 Mp/s for minimum-sized packets.

Software BRAS An implementation of a Broadband Remote Access Server (BRAS), including PPPoE session handling. The data plane checks session numbers and PPPoE/PPP message types, strips tunnel headers, and performs IP lookup and MAC header re-writing.

IDS A simple Intrusion Detection System based on regular expression matching. The reported results are for a single rule that matches the incoming packets.

Load Balancer This re-writes packet source MAC addresses in a round-robin fashion based on the IP src/dst, port src/dst and type 5-tuple in order to split packets to different physical ports.

Flow Monitor To retain per flow (5-tuple) statistics.

Figure 4.1 shows throughput results for the various middleboxes. Overall, ClickOS performs rather well, achieving almost line rate for all configurations for 512-byte and larger packets (the BRAS and CG-NAT middleboxes have rates slightly below the 2.3 Mp/s line rate figure). For smaller packet sizes the percentage of line rate drops, but ClickOS is still able to process packets in the millions per second.

To get and idea of how this relates to a real-world traffic matrix, compare this to an average packet size of 744 bytes reported by a recent study done on a tier-1 OC192 (about 10Gb/s) backbone link [144]: if we take our target to be packets of around this size, all middleboxes shown can sustain line rate.

Naturally, some of these middleboxes fall short of being fully functional, and different configurations (e.g., a large number of firewall rules) would cause their performance to drop from what we present here. Still, we believe these figures to be high enough to provide a sound basis upon which to build production middleboxes.

The carrier-grade NAT, for instance, is proof of this: it is fully functional, and in stress tests it is still able to handle packets in the millions per second.

4.3 Click Applications Implementation

4.3.1 Distributed Deep Packet Inspection

A deep packet inspection system (DPI) represents a common deployed network functionality. A flow based processing system, as the CHANGE platform, takes adavantage in doing this task and remove the need for specialized DPI devices.

DPIs use more or less complex traffic patterns to identify common network protocol behaviour. Different alternative approaches can be used in protocol decoding. While the pattern-based approach (that is, looking for a particular string or regular expression all along the flow streams) is the most common used approach (and it is used in our prototype implementation); it is worth to name another approach that is gaining popularity nowadays. It consists in a ”lighter” approach, in the sense that instead of looking for complex regular expression patterns in the whole flow data data stream, it tries to compare (that it using only exact string match) very small string pattern (usually few bytes) in specific location of the data stream (for example in the first 2 bytes); from this lighter approach it takes its name: Light Packet (or Protocol) Inspection (**LPI**). *Libprotoident* [1] is a distinguishing representative of this second approach.

4.3.1.1 Implemented Click Elements

In this and the following sections we'll describe with greater details the implemented Click elements used to build the choosen use-case scenarios.

4.3.1.1.1 FlowCache Hearth of the DPI's implementation the *FlowCache* element.

As the next L7 element, they are based upon an other GPL'ed Open Source project [39] from the University of Cambridge developed under the unbrella of the *NetOS*[2] project.

In the FlowCache element resides the most important data structure: a lookable hash table implementation called *FlowTable*. It stores the states (as multiple are allowed to exist concurrently depending by the Click configuration) of the flows seen during the ClickOS run time. To identify and correlate the flows a packet, traversing the system, belongs to a classical 4-tuple hash calculation is made (IP addresses and TCP/UDP ports). The calculated hash is used in the FlowTable to store the status of the already seen flows.

A typical FlowCache configuration file is like in the next listing:

```
fc::FlowCache(
    TCP_TIMEOUT 600,
    TCP_DONE_TIMEOUT 15,
    RESOURCE_TCP_MAX_FLOWS 65535,
    RESOURCE_TCP_MAX_BUCKETS 65535,
```

[1] http://research.wand.net.nz/software/libprotoident.php
[2] https://www.cl.cam.ac.uk/research/srg/netos/

```
RESOURCE_TCP_MAX_BUCKET_LENGTH 65535,
RESOURCE_TCP_INITIAL_BUCKETS 1024,
UDP_TIMEOUT 6000,
RESOURCE_UDP_MAX_FLOWS 65535,
RESOURCE_UDP_MAX_BUCKETS 65535,
RESOURCE_UDP_MAX_BUCKET_LENGTH 65535,
RESOURCE_UDP_INITIAL_BUCKETS 1024,
ANNO <ANNO_OFFSET>)
```

A couple of timeout parameters are present in the configuation, for both the TCP and UDO. It's worth to spend some words about the *memory* "resource" parameters. They allows to start the ClickOS with a pool of flow table objects, statically allocated at the startup. When a *new* flow is identified (i.e. a packet with a hash not present in the flow table) the memory pool is asked for the next available memory data chunk and in the case there are some available, one is returned and used immediately otherwise the packet is discarded. For example, when a flow exceedes the configured timeouts it is removed from the flow table and returned back to the memory pool for the next requests.

The last *ANNO* ClickOS parameters represents a ANNO_OFFEST, it specifies where, in the packet, the annotation will be find.

One of our contribution was to extend the existing memory pool allocation code to work with the ClickOS infrastructure as it was designed to run inside the Linux userlevel.

4.3.1.1.2 Layer-7 Protocol and Pattern Info Elements

The Layer-7 Protocol element (hereafter *L7*) is the building block for the IDS and DPI use-case scenario implementation.

As the name suggests, its main purpose it to perform an application protocol identification (the *Layer 7* in the OSI reference model). To achieve this purpose a set of protocol patterns (typically regular expressions) has to be defined in the ClickOS configuration file. The original patterns set come from the well known repository l7-filter [3], modified in the manner to reduce the false positive and negative identification rates.

Our main contributions in the L7 elements consist, on one hand, on a general modularization of the code to allow a more easely code re-utilization. It needed to be modified to permit it running inside the ClickOS platform and allowing it to parse the protocol patterns directly from the ClickOS configuration file instead of the l7-filter pattern files on the filesystem. It was found a more convenient way do work as ClickOS has a very limited support for filesystem access and every peace of information coming from the driver domain has to deal with the Xen store virtualized filesystem via the, already cited, ClickOSControl element.

A new Click element was developed (*L7ProtocolInfo*) and can be used as a pattern database from the other elements in the Click's pipeline.

[3] http://l7-filter.sourceforge.net/

Here it is how a typical configuration should look like:

```
pi::L7PatternInfo(
sip ^(invite|register|cancel) sip[\x09-\x0d -~]*sip/[0-2]\.[0-9]/,
imap ^(\* ok|a[0-9]+ noop)/,
aim ^(\*[\x01\x02].*\x03\x0b|\*\x01.?.?.?.?\x01)|flapon|toc_signon.*0x/,
jabber <stream:stream[\x09-\x0d ][ -~]*[\x09-\x0d ]xmlns=['"]jabber/,
ssh ^ssh-[12]\.[0-9]/,
telnet ^\xff[\xfb-\xfe].\xff[\xfb-\xfe].\xff[\xfb-\xfe]/
)
```

In the above listing, there are defining of some patterns for common application protocols: *sip*, *imap*, *aim*, *jabber*, *ssh*, *telnet*.

An other important configuration parameter of the *L7* element is the maximum number of packets to take into consideration for running the protocol pattern matching on; alternatively the amount of bytes of the flows can be specified; otherwise they can be specified both.

As an optimization, we modified the L7-Protocol element to take into consideration the minimal length of the regular expression to be matched. In the case we haven't seen this minimal number of bytes (or packets) yet, the regular expression is not launched and will wait until a sufficient number of packets of the flow is found.

4.3.1.2 Deep packet Inspection pipeline

In Figure 4.2 a simplified ClickOS processing pipeline, implementing a DPI processing module, is shown. Here, the processing task starts from the two *FromNetFront*s elements,where the packets are retrieved from the underlining virtual network devices attached to the ClickOS virtual machine. On the other side, the processing ends on the two *ToNetFront* element, where packets are sent back to the network devices. In the middle of this processing path more exciting things happen.

Firstly each packet are "annotated" with an integer value corresponding to the ingress interface it was seen on. For example, if the pakcet was coming from the interface **eth0** it will be annotated with a value of 0, while the packets retrieved from the second network device are instead annotated with a value equal to 1. In this way, the output element, here represented by the *PaintSwitch* element can use these informations to find the right egress interface for the packets, that is packets with an annotation equal to 0, will be, in the end, forwarded to egress interface nr. 1, otherwise the other exit path will be taken for the packet.

In the central part of the same figure, it is shown the core processing elements of this pipeline. The *SimpleRoundRobinSched* element has the function to aggregate the packets coming from its different input ports and schedule them to coming out its one output port.

Attached to the scheduler ouput port there is *FlowCache* element.

Finally a *L7* element using the state stored in the cache and the pattern specifications given though the pipeline configuration can start to match the incoming packets against the patterns defined and in case of positive

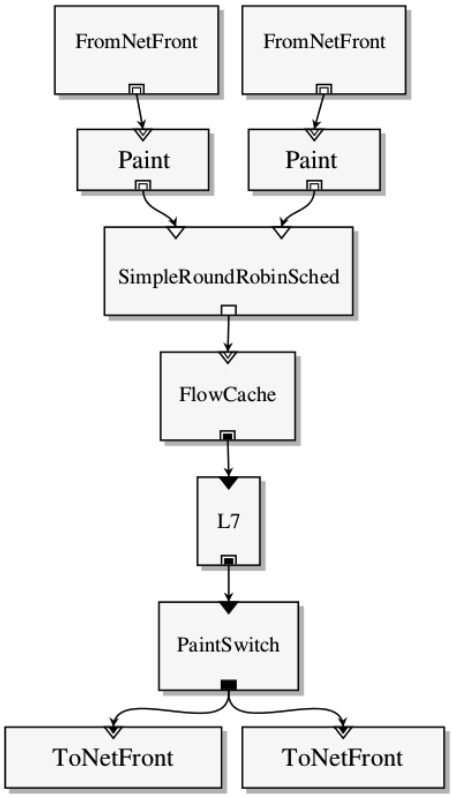


Figure 4.2: ClickOS DPI Processing Module pipeline.

match annotate the flow with the result of the match. If a matching pattern is found the same result will the applied to all the following packets belonging to the same flow.

4.3.2 Distributed IDS

An intrusion detection system (IDS) monitors network activities for malicious activities or policy violations and produces reports for the system administrator. Deploying IDS in the network involves complex processing including decoding packet data, aggregating distinct packets into streams and inspecting them according to a specific rule set.

In the context of CHANGE architecture, and in order to properly deploy DIDS systems, the CHANGE platforms should be able to communicate between them, and select the right flow to process if there is enough resource in the current platform.

Different approaches are possible in a IDS implementation: a Stateless approach, where the decision to pick-up and process a flow can be taken from a CHANGE platform with regards only of the locally available informations; or a Stateful approach, where a CHANGE platform can take advantage from the informations coming from the other deployed platforms in the network.

4.3.2.1 Implemented Click Elements

4.3.2.1.1 Signature Matcher Element Basing upon the annotated metadata of the packets seen for the DPI use-case, the Signature Matcher Click element (hereafter *SigMatcher*) can start its processing task. The

configuration of this element starts with the definition of a set of signatures agaist which the flows need to be parsed.

A signature basically consists of three kind of information:

- a couple of IP addressed identifying the end-points of the comunication;
- optionally a layer-7 protocol to handle, that need to be specifiied a the L7PAtternInfo element;
- an action that needs to be taken on the matching the signature specification.

Regarding the action to be taken, two basic alternatives are possible:

- DROP action: the matching packets have to take the output port nr 1 path, in the case further Click elements were connected to the port, discarded otherwise;
- ALLOW action: in this case the matching packets can follow their ”normal” path to the respective destination end-points.

4.3.2.2 Intrusion Detection System pipeline

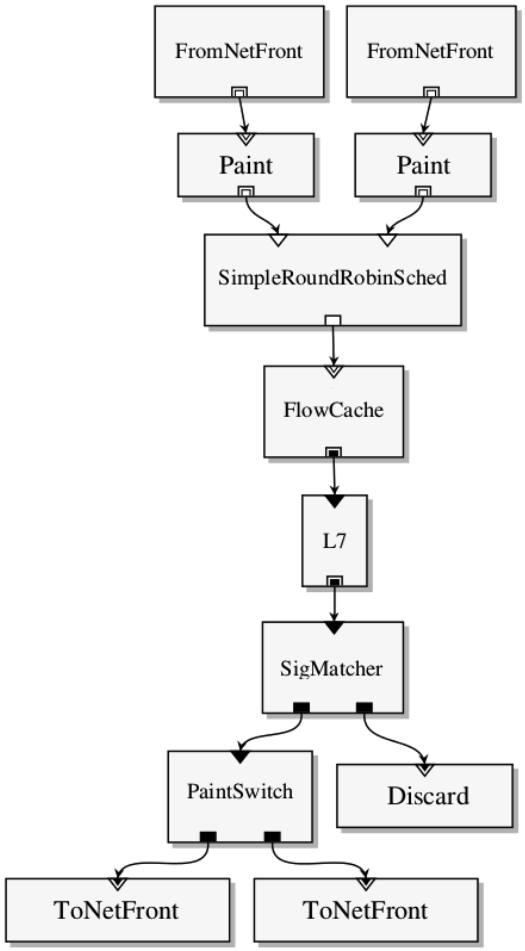


Figure 4.3: ClickOS IDS Processing Module pipeline.

In Figure 4.2 a simplified ClickOS pipeline, implementing a IDS processing module, is shown. The input and ouput parts of the pipeline are the same as in the DPI scenario. The main difference is is that a *SigMatcher* element is now attached to the *L7* ouput port.

In this way, the signatures defined in the ClickOS pipeline configuration can we ran against the incoming packets. Depending on the result of the signature application, a packet can either take the ouput port nr 0 (this means that the packets is allowed) or the output port nr 1, in the case it exists(or it is discarded otherwise). For simplicity the second port of the SigMatcher element it attached to a *Discard* element that means that the packets are effectively discarded by the platform as they are not allowed to progress further in the pipeline processing.

4.3.3 Distributed Firewall/Policy Enforcer

As oulined in D5.2 document, firewalls are common network elements used to protect users and Corporations against unwanted traffic. Running a firewall requires the ability to handle many rules and quickly change them when needed while being able to scale to traffic volume variations spanning several orders of magnitude.

While possible implementations of a firewall can use commodity OSs for low traffic level; for massive traffic volumes public clouds are increasingly acquiring popularity. Here, CHANGE platforms can play an important role as, maintaining generally a smaller dimension, CHANGE platforms can reach a more capillary deployment diffusion than the other public clouds that are generally located in few vantage locations (e.g. one per continent as in the case of the Amazon EC2).

As result of this capillary diffusion, a CHANGE platform can reside near the end-users that can benefit of a lower RTT and a more fair chargement treatment as a user has to pay only for the bandwidth really used and not for the provided one.

On the other hand, smaller dimensions help to achieve the desired scalability level as multiple resources can be allocated on a CHANGE platform or other platform can involved in the traffic filtering in presence of traffic peaks.

4.3.3.1 Implemented Click Elements

4.3.3.1.1 Firewall Element As Click is already provided of all the necessary elements to build a quite powerful firewalling processing path, no other actions are needed to be taken. Nanely the Click elements used are the *Classifier*, the *IPClassifier* and the *IPFilter* elements. In this way complex combination of addresses and protocol filtering rules can be specified.

4.3.3.2 Firewall Pipeline

In Figure 4.4 a simplified ClickOS pipeline, implementing a simple Firewall processing module, is presented. As already outlined, the in-stock elements, included in the Click distribution, can be effectively used to build a working firewall processing pipeline.

In the proposed pipeline, only a pair of *in/egress* elements are present but more can be added using the same packet metadata annotation we've already used for the other pipelines.

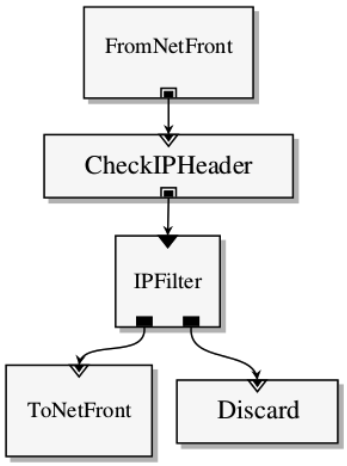


Figure 4.4: ClickOS Firewall Processing Module pipeline.

What we want to outline here is that Click already offers a powerful set of elements and more can be easily added.

4.3.4 Distributing flow streams: the FlowPinner Element

In this last section, we want to discuss the last Click element developed: the *FlowPinner*.

In order to allow the DPI (and the IDS) to successfully complete the protocol identication the complete data stream of the flow needs to be inspected, that is both the incoming/outgoing streams needs to walk though the processing modules.

Classical hashing classification scheme, for example the ones based on the 4-tuple or 5-tuple algorithm, cannot achieve this result as they suffer to be not symmetrical in regards of the end-points addresses; that is the hash, corresponding to the packets traveling from an end-point A (let's say a pair of an IP address and a TCP port) to another end-port B, is not the equal to the hash for a packet going from B to A.

As a solution, one could think of making use, in the hash calculation, of just symmetrical operators (like addictions or multiplications). Effectively this approach works but it has serious drawbacks about the quality of the hashes, in fact the hash collition probabilities become very hight.

Our contribution was to port Bob Jenkins' **lookup3** hash functions [4] in the ClickOS platform. These set of hash functions were already used in many application and proved to have a very good compromise between the hash quality and the speed of calculation.

Depending on the number of output ports the FLowPinner can, then, split the traffic on its multiple outgoing ports, reaching a certain level of traffic load balancing (that comes for free as Jenkins' hash functions have good uniformity properties). Also optimization in the code was taken in case the output ports are power of two.

With this in mind, we want to discuss a possible usage of the FlowPinner element. In the following discussion many concepts come from the FlowStream platform.

[4] the source code is freely available at the website http://www.burtleburtle.net/bob/c/lookup3.c

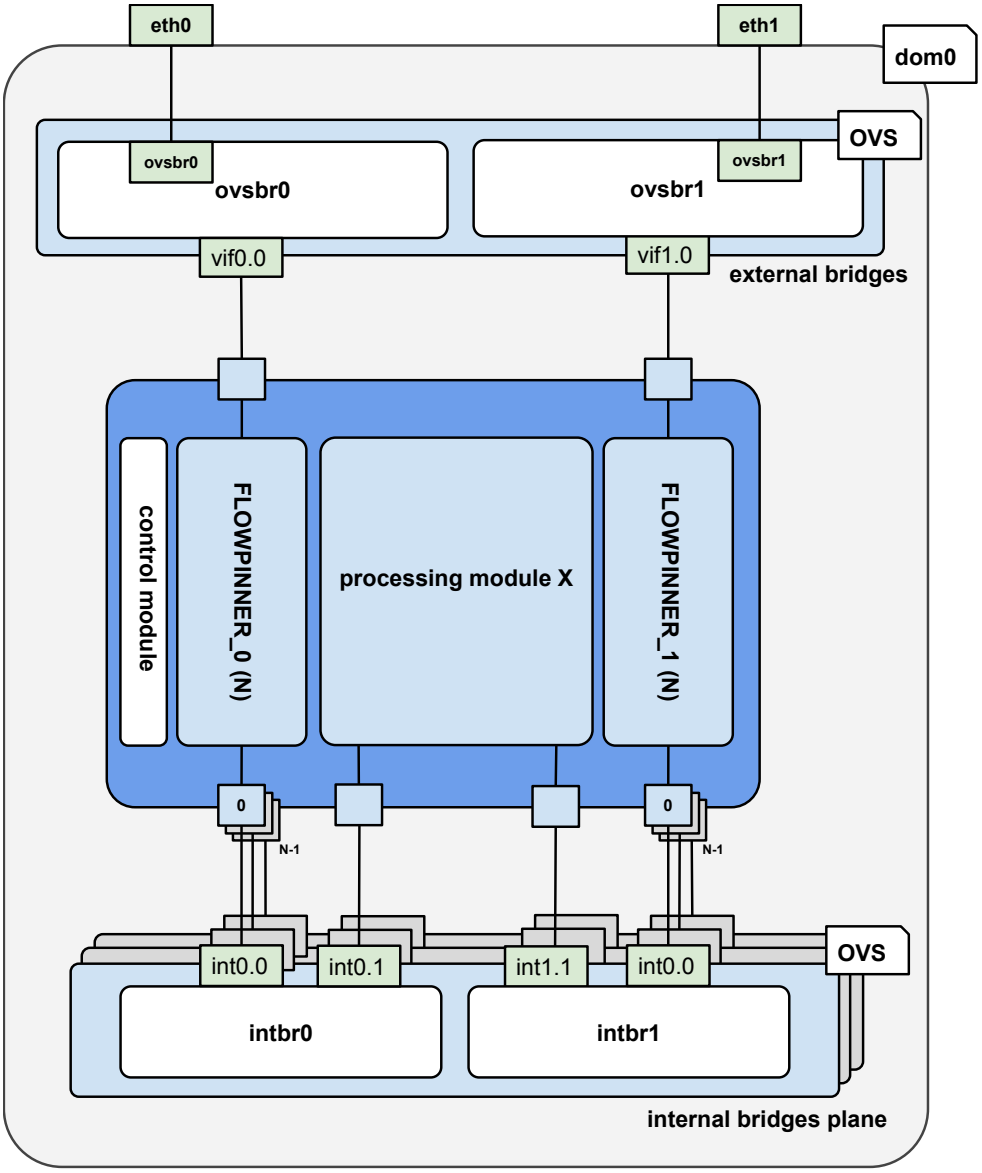


Figure 4.5: Flowpinner usage example.

In Figure 4.5, we can distinguish three main components:

- a pair of OpenVSwitch based (hereafter OVS) bridge, called external bridge, as they are connected to the external (real) network interfaces;
- a set of internal bridges (OVS based) that we call ”plane”,’cause they are not connected to any real interfaces;
- some ClickOS processing modules, a couple of FlowPinner (hereafter FP) elements with N output ports each (thatś why the name) and a generic processing module X.

From the top, the external interfaces are connected to the two OVS bridges, each one also connected to a FlowPinner_N element. As the two FPs have the some number of output ports (and they use they same hashing method), each packet of every flow stream will take the same output port. In this way, the packets forming a flow will arrive to the internal bridge plane to their respective internal OVS bridge. In same way

each internal bridge, of the same output level, is connected to one processing module (X). So in the end, we are sure that each packet of a flow will arrive to the same PM.

Also this configuration has the flexibility to work in the case on different *Module Hosts*, in exactly the same way.

4.4 CDN-ISP Collaboration (NetPaaS)

4.4.1 Introduction

Recently, Akamai formed content delivery strategic alliances with major ISPs, including AT&T [1], Orange [16], Swisscom [17], and KT [14]. The formation of CDN-ISP alliances is a paradigm shift in how content delivery networks will be deployed in the future and opens new directions for innovative solutions for CDN-ISP collaboration. It is also the natural evolution of innovative approaches for content delivery that have been deployed for more than a decade to address scalability, performance, and cost issues as well as to take advantage of business opportunities.

Today's Internet traffic is dominated by content distribution [23, 67, 95, 106] delivered by a variety of CDNs. Gerber and Doverspike [67] and Poese et al. [127] report that a few commercial CDNs account for more than half the traffic in a North American and a European tier-1 carrier, respectively. More than 10% of the total Internet inter-domain traffic originates from Google [95], and Akamai claims to deliver more than 20% of the total Internet Web traffic [119]. Netflix, which uses multiple CDNs, is responsible for around 30% of the traffic in North America during peak hours [78].

To cope with continuously increasing demand for content, a massively distributed infrastructure has been deployed by CDNs [98, 25]. Some CDNs as well as CDN-accelerated cloud and service providers rely on a number of datacenters in strategic locations on the Internet, e.g., Limelight is present in more than 70 locations, Google operates tens of data centers [156], Microsoft Azure uses 24 locations, and Amazon AWS relies on 6 large datacenters and operates caches in more than 22 locations. Others deploy highly distributed infrastructures in a large number of networks, e.g., Akamai operates more than $100,000$ servers in more than $1,800$ locations across nearly $1,000$ networks [119].

The existing content delivery platforms, however, do not always have servers in locations that can satisfy the growing demand and provide good performance. One reason is limited agility in server deployment, as it takes time to find the locations in the right places with the required capacities, make the necessary business arrangements, and install the servers [119]. Moreover, the content delivery market is very competitive, leading CDNs to investigate ways to reduce capital and operating costs [133].

To address these two challenges, a variety of designs have appeared over the last decade. These solutions expand the CDN footprint by dynamically deploying servers as needed or leveraging the resources of end-users. An overview of the spectrum of the various solutions and the level of involvement of content delivery stakeholders is shown in Figure 4.6. Commercial CDNs [21] as well as ISPs [96] operate hybrid delivery systems where end-users download content from the servers as well as other end-users to reduce the bandwidth and

energy cost respectively at the server side. Commercial CDNs also license content delivery software to ISPs that maintain servers [2]. In some cases these licensed CDNs are able to coordinate with the CDN-operated servers or with other CDNs enabling CDN federations, see e.g., the CDNI IETF group. Meta-CDNs have also been proposed to optimize for cost and performance by acting as brokers for CDN selection [100, 56]. P2P systems are also successful in utilizing the aggregate capacity of end-users that are interested in downloading the same content [47]. P4P [175] has been proposed as an ISP-P2P collaboration mechanism to better localize traffic. Content providers (CPs) are also moving to deploy application-specific CDNs with direct peering with or inside ISPs, e.g., Netflix Open Connect for video stream delivery [15] or Google Global Cache, primarily for YouTube [12, 31]. The advantage of such specialized CDNs is that they can be optimized for the application.

Another recent trend is to marry cloud resources (processing and storage) with networking resources to meet the high performance requirements of certain applications, such as high definition video streaming or online gaming on demand [147]. Moreover, many ISPs support the migration from solutions that rely on proprietary hardware to those that rely on generic appliances and take advantage of virtualization to reduce complexity and avoid vendor lock-in [19]. Large ISPs, including AT&T, Deutsche Telekom, and Telefonica, have already deployed generic appliances in relatively small datacenters, also referred to as *microdatacenters*, co-located with their major network aggregation locations. Initially, such deployments were to support their own services such as ISP-operated CDNs, IPTV, carrier-grade NAT, deep packet inspection, etc., but they now offer full virtualization services [18]. These new capabilities allow ISPs to offer network and server resources to CDNs, applications, and services, close to their end users. Recent studies [150] also show that enterprises can outsource part of their infrastructure in the cloud and take advantage of the new virtualization market.

Economics and market share are also key drivers. Large CDNs have a strong customer base of content providers and are responsible for delivering content for their customers to end-users around the world. On the other hand, ISPs have a strong end-user base in some regions and also, as mentioned above, have invested significantly in adding infrastructure at the aggregation locations (PoPs) of their networks. The combined "ownership" of content providers and end-users is a major driving force behind recent CDN-ISP alliances [1, 16, 17, 14] as both sides strive to reduce operational cost and at the same time offer better content delivery services.

Despite the clear opportunity for collaboration, the necessary mechanisms and systems to enable joint CDN deployment and operation inside the network are not yet available. Our contributions are summarized as follows:

- We revisit the design and operating space of CDN-ISP collaboration in light of recent announced alliances and we identify two major enablers for collaboration, namely informed user-server assignment and in-network server allocation.

- We design and implement a novel prototype system, called NetPaaS (Network Platform as a Service),

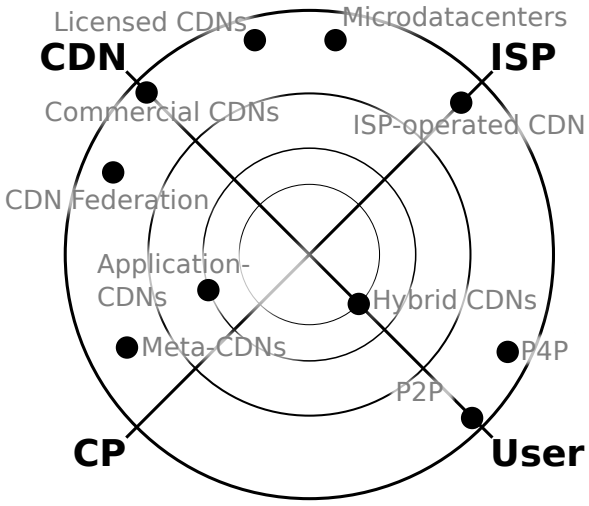


Figure 4.6: Spectrum of content delivery solutions and involvement of stakeholders.

that incorporates the two key enablers to address CDN-ISP collaboration system issues towards a joint CDN deployment and operation inside the ISP network.

- We perform the first-of-its-kind evaluation based on traces from the largest commercial CDN and a large tier-1 ISP using NetPaaS. We report on the benefits for CDNs, ISPs, and end-users. Our results show that CDN-ISP collaboration leads to a win-win situation with regards to the deployment and operation of servers within the network, and significantly improves end-user performance.

4.4.2 Enabling CDN-ISP Collaboration

CDN-ISP collaboration has to address a set of challenges regardless whether a CDN utilizes traditional or emerging solutions to deliver content. We first highlight these challenges for content delivery today and then propose two key enablers to address them and facilitate CDN-ISP collaboration.

4.4.2.1 Challenges in Content Delivery

Economics, especially cost reduction, is a main concern today in content delivery as Internet traffic grows at a annual rate of 30% [118]. Moreover, commercial-grade applications delivered by CDNs often have requirements in terms of end-to-end delay [94]. Faster and more reliable content delivery results in higher revenues for e-commerce and streaming applications [98, 119] as well as user engagement [56]. Despite the significant efforts by CDNs to improve content delivery performance, end-user mis-location, and the limited view of network bottlenecks are major obstacles to improve end-user performance.

Content Delivery Cost: CDNs strive to minimize the overall cost of delivering voluminous content traffic to end-users. To that end, their assignment strategy is mainly driven by economic aspects such as bandwidth or energy cost [100, 133]. While a CDNs will try to assign end-users in such a way that the server can deliver reasonable performance, this does not always result in end-users being assigned to the server able to deliver the best performance. Moreover, the intense competition in the content delivery market has led to diminishing returns of delivering traffic to end-users. Part of the delivery cost is also the maintenance and

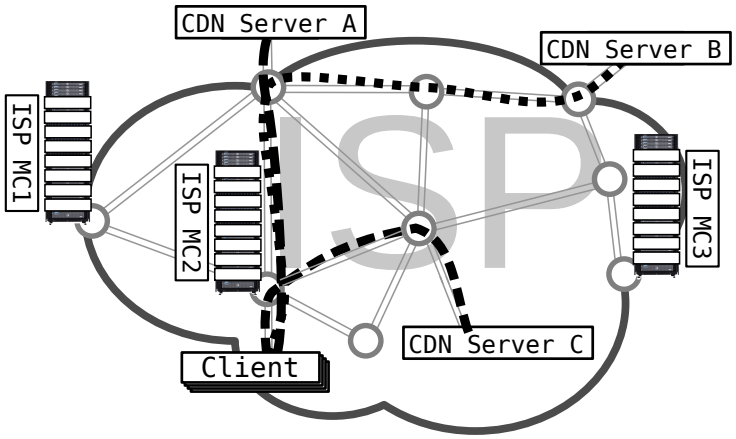


Figure 4.7: *Informed User-Server Assignment:* Assigning a user to an appropriate CDN server among those available (A, B, C), yields better end-user performance and traffic engineering. *In-network Server Allocation:* A joint in-network server allocation approach allows the CDN to expand its footprint using additional and more suitable locations (e.g., microdatacenters MC1, MC2, MC3) inside the network to cope with volatile demand. User-server assignment can also be used for redirecting users to already deployed and new servers.

constant upgrading of hardware and peering capacity in many locations [119].

End-user Mis-location: DNS requests received by the CDN name servers originate from the DNS resolver of the end-user, not from the end-user themselves. The assignment of end-users to servers is therefore based on the assumption that end-users are close to the used DNS resolvers. Recent studies have shown that in many cases this assumption does not hold [100, 133]. As a result, the end-user is mis-located and the server assignment is not optimal. As a response, DNS extensions have been proposed to include the end-user IP information [49, 122].

Network Bottlenecks: Despite their efforts to discover end-to-end characteristics between servers and end-users to predict performance [119, 94], CDNs have limited information about the actual network conditions. Tracking the ever changing network conditions, i.e., through active measurements and end-user reports, incurs an extensive overhead for the CDN without a guarantee of performance improvements for the end-user. Without sufficient information about the characteristics of the network paths between the CDN servers and the end-user, a user assignment performed by the CDN can lead to additional load on existing network bottlenecks, or even create new ones.

4.4.2.2 Enablers

Given the trends regarding increasing need of server resources and content demand by end-users, content delivery systems have to address two fundamental problems. The first is the end-user to server assignment problem, i.e., how to assign users to the appropriate servers. The key enabler for addressing this problem is *informed user-server assignment* or in short *user-server assignment*. It allows a CDN to receive recommendations from a network operator, i.e., a server ranking based on performance criteria mutually agreed upon by the ISP and CDN. The CDN can utilize these recommendations when making its final decision regarding end-user to server assignments. This enabler takes full advantage of server and path diversity, which a CDN has difficulty exploring on its own. Moreover, its design allows the coordination of CDNs, content providers and

ISPs in near real-time, as we will elaborate in section 4.4.3. Any type of CDN can benefit from this enabler including ISP-operated CDNs. The advantage of our enablers in comparison with other CDN-ISP [53, 83] and ISP-P2P [175] cooperation schemes is that no routing changes are needed.

The second is the server allocation problem, i.e., where to place the servers and content. The key enabler is *in-network server allocation*, or in short *server allocation*, where the placement of servers within a network is coordinated between CDNs, ISPs, and content providers. This enabler provides an additional degree of freedom to the CDN to scale-up or shrink the footprint on demand and thus allows it to deliver content from additional locations inside the network. Major improvements in content delivery are also possible due to the fact that the servers are placed in a way that better serve the volatile user demand. The application of this enabler is two-fold. One, it helps the CDN in selecting the locations and sizes of server clusters in an ISP when it is shipping its own hardware. The second application is suitable for more agile allocation of servers in cloud environments, such as those mentioned in [19]. Multiple instances of virtual servers running the CDN software are installed on physical servers owned by the ISP. As before, the CDN and the ISP can jointly decide on the locations and the number of servers. A big advantage of using virtual machines is that the time scale of server allocation can be reduced to hours or even minutes depending on the requirements of the application and the availability of physical resources in the network. User-server assignment can also be used for redirecting users to the new servers. We provide the high-level intuition for both enablers in Figure 4.7.

Until now, both problems have been tackled in a one-sided fashion by CDNs. We believe that to improve content delivery, accurate and up-to-date information should be used during the server selection by the CDN. This also eliminates the need for CDNs to perform cumbersome and sometimes inaccurate measurements to infer the changing conditions within the ISP. We also believe that the final decision must still be made by the CDN. In this paper, we argue that the above enablers (a) are necessary to enable new CDN architectures that take advantage of server virtualization technology, (b) allow fruitful coordination between all involved parties, including CDNs, CPs, and ISPs in light of the new CDN-ISP alliances, (c) enable the launch of new applications jointly by CDNs and ISPs, and (d) can significantly improve content delivery performance. Such performance improvements are crucial as reductions in user transaction time increase revenues by significant margins [89].

4.4.3 NetPaas Prototype

Today there is no system to support CDN-ISP collaboration and joint CDN server deployment within an ISP network. In this section we design a novel system, NetPaaS (Network Platform as a Service), which incorporates the two key enablers for CDN-ISP collaboration introduced in Section 4.4.2. First, we give a overview of NetPaaS and describe its functionalities and the protocols it utilizes to enable collaboration. Next, we give a detailed description of the NetPaaS architecture. Finally we comment on the scalability and privacy preserving properties of NetPaaS.

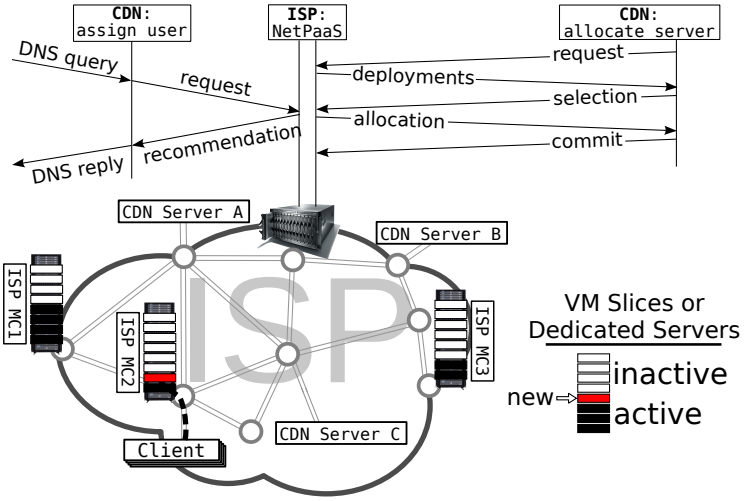


Figure 4.8: NetPaaS protocols and operation.

4.4.3.1 NetPaaS Functionalities and Protocols

NetPaaS enables CDNs and ISPs to efficiently coordinate the user to server assignment and allows the CDN to expand or shrink its footprint inside the ISPs network on demand, towards achieving performance targets [94] and traffic engineering goals [129]. Neither of them is a trivial task when dealing with large networks (thousands of routers), highly distributed microdatacenters (in tens of locations and hundreds of machines), and constant network, routing, and traffic updates.

The NetPaaS protocol allows CDNs to express required server specifications and ISPs to communicate available resources and their prices. It is designed to exchange information in very small time scales, e.g., in the order of seconds (similar to the time scale that CDNs can potentially redirect users [127]), enabling fast responses to rapid changes in traffic volumes. Any ISP operating a NetPaaS system offers the following services: ***(1)*** **User-server assignment**: allows to request recommendations for user to server mapping from the ISP. ***(2)*** **Resource discovery**: communicates information about resources, e.g., available locations or number of servers and the conditions for leasing them, e.g., price and reservation times. ***(3)*** **Server allocation**: enables a CDN to allocate server resources within the ISPs network.

The protocol utilized by NetPaaS is designed to be efficient and to minimize delay and communication overhead. The required communication for the different services are explained in more detail in the following Sections 4.4.3.1.1 and 4.4.3.1.2. For the user-server assignment service NetPaaS also supports BGP as communication protocol as this is already supported by many CDN operators, e.g., Google Global Cache [12], Netflix Open Connect [15], or the Akamai Network [119].

4.4.3.1.1 NetPaaS Protocol for User-Server Assignment

We first describe the general approach for user-server assignment today and continue with the required additional steps and protocol messages for our collaborative approach, illustrated in the top left of Figure 4.8 (“CDN: user assign”). When a CDN receives a DNS request, typically by a resolver (i.e., when the answer is not locally available in the local resolver), it utilizes internal information in order to assign a server to satisfy the request. The selection of the server depends on the location of the source of the request, as this is inferred from the resolvers that sends

it, as well as the availability of close-by servers and cost of delivery [119, 154]. When the CDN selects a set of servers to satisfy the request, it sends a DNS reply back to the resolver that sent the DNS request who then sends it to the source of the request. Notice that for scalability reasons and to deal with flash crowds, large CDNs allow all the available servers to serve the same content [159]. If the content is not locally available, the server fetches the content from other servers or the original server, stores it locally (that yields pull-based replication), and sends it to the end-user [119]. To take advantage of the ISPs NetPaaS user-server assignment service the CDN issues a recommendation request prior to answering the DNS query. The recommendation request contains the source of the DNS request and a list of eligible CDN server IPs which NetPaaS ranks based on ISP-internal information, e.g., link utilization or path delay, and possible traffic engineering goals. If the source of the DNS request is the ISP operated DNS resolver or when the EDNS0 Client Subnet Extension [49] is present, NetPaaS can precisely locate the end-user inside the ISPs network, effectively increasing the recommendations precision of the system. The ISP then returns this preference ordered list in a recommendation message to the CDN which can select the most appropriate servers based on both the ISPs and its own criteria and thus optimizing the user-server assignment while staying in complete control of the final server selection process.

4.4.3.1.2 NetPaaS Protocol for Server Allocation

We next describe the steps and required protocol messages for collaborative server allocation that are illustrated in the top right of Figure 4.8 ("CDN: allocate server"). When a CDN decides that additional servers are needed to satisfy the end-user demand or when the CDN and ISP jointly agree to deploy new servers inside the ISP, the CDN submits a request to NetPaaS. The request contains the required hardware resources, a demand forecast (e.g., per region or per subnet) together with a number of optimization criteria and possible constraints. The demand forecast allows NetPaaS to compute an optimal placement for the newly allocated server(s). Optimization criteria include minimizing network distance or deployment cost among others. Possible constraints are the number of locations, minimum resources per server, or reservation time. Based on this information NetPaaS computes a set of deployments, i.e., the server locations and the number of servers, by solving an optimization problem (namely the SiSL or the CFL problem, see Section 4.4.3.2.3). The reply contains the possible deployments and their respective prices. The CDN either selects one or more of the offered deployments by sending a selection message to NetPaaS or starts over by submitting a new request. When receiving a selection message, NetPaaS checks if it can offer the selected deployment. If all conditions are met, NetPaaS reserves the requested resources to guarantee their availability and sends an allocation message as confirmation to the CDN. If the conditions cannot be met, the selection by the CDN is denied by NetPaaS. To gain control of the allocated servers, the CDN has to send a commit message to NetPaaS which completes the communication for server allocation.

The ISP may offer physical machines or virtual machines (VMs) to CDNs. In the second case the servers are refereed to as "slices" of hardware servers. To move servers from one to another network position, NetPaaS

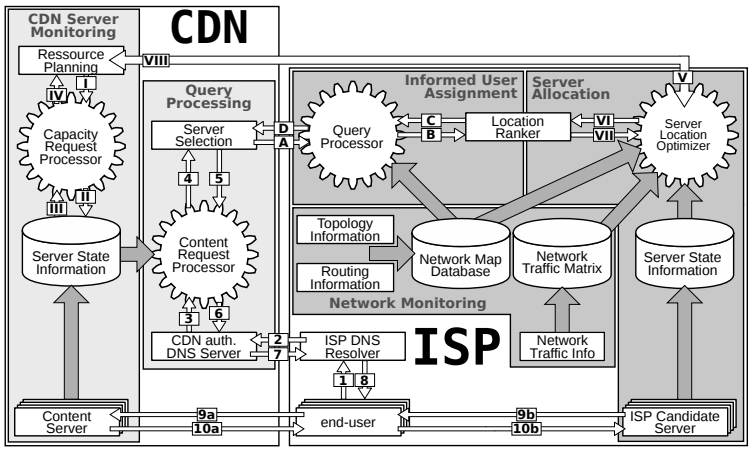


Figure 4.9: NetPaaS architecture.

supports the flexibility of VM migration or consolidation. A possible deployment scenario with VMs can be seen in Figure 4.8. To improve CDN server startup and cache warm-up times, one option for CDNs is to always keep a small number of active servers in a diverse set of locations to expand or shrink it according to the demand. They can also pre-install an image of their server in a number of locations.

4.4.3.2 Architecture

We now provide the detailed architecture of the system, aimed at providing accurate user-server assignments as well in-network server allocations for the CDN. We describe the components and processes both at the ISP as well as the CDN side. In the ISP the main tasks of our system are to: (1) maintain an up-to-date annotated map of the ISP network and its properties as well as the state of the ISP-operated servers within the network, (2) provide recommendation on where servers can be located to better satisfy the demand by the CDN and ISP traffic engineering goals, and (3) to assist the CDN in user-server assignment and server allocation by creating preference rankings based on the current network conditions. The goal of the system is to fully utilize the available server and path diversity as well as ISP-maintained resources within the network, while keeping the overhead for both the CDN and the ISP as small as possible.

NetPaaS comprises three main components: *Network Monitoring*, *Informed User Assignment*, and *Server Allocation Interface*. For an overview of the architecture, see the ISP grey area in Figure 4.9. Steps 1-10 and I-IV that illustrate the requests and responses and the CDN server selection respectively, as performed in currently deployed CDNs, for more information and details see [119].

4.4.3.2.1 Network Monitoring Component The Network Monitoring component gathers information about the topology and the state of the network to maintain an up-to-date view of the network. The Topology Information component gathers detailed information about the network topology, i.e., routers and links, annotations such as link utilization, router load as well as topological changes. An Interior Gateway Protocol (IGP) listener provides up-to-date information about routers and links. Additional information, e.g., link utilization and other metrics can be retrieved via SNMP from the routers or an SNMP aggregator. The Routing Information uses routing information to calculate the paths that traffic takes through the network. Finding the path of egress traffic can be done by using a Border Gateway Protocol (BGP) listener. Ingress points of

traffic into the ISP network can be found by utilizing Netflow data. This allows for complete forward and reverse path mapping inside the ISP. In total, this allows for a complete path map between any two points in the ISP network. The Network Map Database processes the information collected by the Topology and Routing Information components to build an annotated map of the ISP network. While it builds one map of the network, it keeps the information acquired from the other two components in separate data structures. The Topology Information is stored as a weighted directed graph, while the prefix information is stored in a Patricia trie [115]. This separation ensures that changes in prefix assignment learned via BGP do not directly affect the routing in the annotated network map. To further improve performance, the path properties for all paths are pre-calculated. This allows for constant lookup speed independent of path length and network topology. Having ISP-centric information ready for fast access in a database ensures timely responses and high query throughput.

4.4.3.2.2 Informed User-Server Assignment Component When the CDN sends a request for user-server assignment to NetPaaS, the request is handled by the Query Processor (steps A to D in Figure 4.9). The request from the CDN specifies the end-user and a list of candidate CDN servers. First, the Query Processor maps each source-destination (server to end-user) pair to a path in the network. Note that the end-user is seen through its DNS resolver, often the ISPs DNS resolver [24], unless both ISP and CDN support the EDNS0 Client Subnet Extension [49, 122]. The properties of the path are then retrieved from the Network Map Database. Next, the pairs are run individually through the Location Ranker subcomponent (see below) to get a preference value. Finally, the list is sorted by preference values, the values stripped from the list, and the list is sent back to the CDN. The ISP Location Ranker computes the preference value for individual source-destination pairs based on the path properties and an appropriate function (see steps B, C). The function depends on the goal specified by the CDN, such as a performance goal, as well as an operational one, such as a traffic engineering objective. Note that NetPaaS is not limited to a single optimization function per CDN.

4.4.3.2.3 In-network Server Allocation Component When the CDN Resource Planner sends a server allocation request to NetPaaS asking for available servers within the ISP (steps V to VIII), the request is handled by the ISP Server Location Optimizer. It uses the Network Monitoring component to get up-to-date information about the ISPs network and the current and historic network traffic matrices and the Server State Information database, which collects up-to-date state information regarding the ISP's servers (e.g., server load and connectivity).

The problem that the ISP Server Location Optimizer has to solve can be modeled as an instance of either the Simultaneous Source Location problem (SiSL) [27], or the Capacitated Facility Location problem (CFL) [91]. The locations at which facilities can be opened correspond to the locations at which servers can be placed, and there is a constraint on the amount of bandwidth available at each location or on each network link.

In SiSL, the goal is to determine where the servers should be placed so as to satisfy demand while respecting

the capacity constraints, and also possibly minimizing the distance between servers and users. Given the specification of a server, if the capacity of a location allows multiple servers to be allocated then the solution may allocate more than one server per location. The ISP has a detailed view of the network activity (e.g., traffic matrices over a period of time), the annotated network topology, and the candidate locations to install servers, along with the available resources, including the network capacity at these locations. The CDN can also express the demand that needs to be satisfied with additional servers as well as the server requirements.

In the CFL solution, to prevent the creation of hot-spots, the distance of users to servers is proportional to the utilization of the most congested link (given the background traffic) along the path from the server to the end-user. We also assume that the user-server assignment enabler is in place. In our setting users can be assigned to different servers for each request to a server. Thus, the demand is splittable. This allows for fast and accurate server allocations using standard local search heuristics for CFL [29].

The outcome of joint server allocation is the number and location of additional servers. The result is communicated to the two parties that have to agree on the calculated setting.

Joint Hardware Server Allocation: In this case the collaboration of the ISP and CDN is in large time scales (weeks) and the servers are physical machines installed and maintained by the ISP and operated by the CDN. In the setting of the ISP-operated CDN, the server allocation is an optimized way of deploying the CDN footprint inside the network. The forecast of the demand by analyzing CDN logs can also be incorporated. This joint operation also allows the launch of new and demanding applications such as video streaming and interactive online gaming.

Joint Software Server Allocation: As mentioned before, servers can be either physical machines owned by the CDN, virtual machines offered by the ISP, or both. With virtualization, the above solution can be utilized whenever software servers are allocated. This allows for flexible server allocation using a mature technology. Virtualization has been used to allocate heterogeneous resources [167, 48], computation (e.g., VMWare, Xen, and Linux VServer), storage, and network [147], in datacenters [28], as well as distributed clouds inside the network [44, 19]. Recent measurement studies have shown significant performance and cost variations across different virtualization solutions [99]. In response, a number of proposals have addressed the specific requirements of applications [33, 92, 104] and the scalability to demand [132, 172]. To capitalize on the flexibility and elasticity offered by virtualization, a number of systems have been built to automate data and server placement [22, 50, 164] and server migration [38, 97] even between geographically distributed datacenters. Other approaches have focused on the selection of locations for service mirrors and caches inside a network, to minimize the network utilization [93, 96]. In the joint server allocation setting the decision and installation time can be reduced to hours or even minutes. This is feasible as an ISP can collect near real-time data for both the network activity and availability of resources in datacenters operated within its network or in microdatacenters collocated with ISP network aggregation points [44].

4.4.3.3 Scalability

User-Server Assignment: To improve scalability and responsiveness, we do not rely on HTTP embedded JSON as proposed in by ALTO IETF group, but on light protocols that are similar to DNS. A single instance of our system is able to reply to more than $90,000$ queries/sec when serving requests with 50 candidate CDN servers. At this level, the performance of our system is comparable to popular DNS servers, e.g., BIND. The computational response time is below 1 ms for a 50 candidate server list. By placing the service inside ISP networks at well connected points, the additional overhead is small compared to the DNS resolution time [24]. This performance was achieved on a commodity dual-quad core server with 32 GB of RAM and 1Gbps Ethernet interfaces. Furthermore, running additional servers does not require any synchronization between them since each instance is acquiring the information directly from the network. Thus, multiple servers can be located in different places inside the network to improve scalability.

Server Allocation: Today, a number of off-the-shelf solutions are available to spin a virtual server based on detailed requirements [104], and are already available from vendors such as NetApp and Dell. To test the scalability of in-network server allocation we used an appliance collocated with a network aggregation point of ADSL users which consists of 8 CPUs (16 cores), 24 GByte RAM, Terabytes of solid state disks, and a 10 Gbps network interface. A management tool that follows the VMware, Cisco, and EMC (VCE) consortium industrial standard [48] is also installed. We tested different server configurations and our results show that VM boot up times are on the order of tens of seconds while virtualization overhead during runtime is negligible. To that end we confirm that it is possible to even fully saturate a 10 Gbps link. It was also possible to add, remove, and migrate live servers on demand in less than a minute. To reduce the cache warm-up time when allocating a new server, the requests to an already operational cache are duplicated and fed to the new one for around ten minutes.

4.4.3.4 Privacy

During the exchange of messages, none of the parties is revealing sensitive operational information. In user-server assignment, CDNs only reveal the candidate servers that can respond to a given request without any additional operational information (e.g., CDN server load, cost of delivery). On the other side, the ISP does not reveal any operational information or the preference weights it uses for the ranking. In fact, the ISP only re-orders a list of candidate servers provided by the CDN. This approach differs from [175], where partial or complete ISP network information, routing weights, or ranking scores are publicly available. During the server allocation a CDN can decide either to request a total demand or demand in a region (e.g., city, country), thus it does not unveil the demand of an end-user.

4.4.4 Datasets

To evaluate the NetPaaS system, we use traces from the largest commercial CDN and a large European tier-1 ISP.

Commercial CDN Dataset: The CDN dataset covers a two-week period from 7th to 21st March 2011. All

entries in the log we use relate to the tier-1 ISP. This means that either the server or the end-user is using an IP address that belongs to the address space of the tier-1 ISP. The CDN operates a number of server clusters located inside the ISP and uses IPs in the IP address space of the ISP (see Section 4.4.5.1). The log contains detailed records of about 62 million sampled (uniformly at random) valid TCP connections between the CDN's servers and end-users. For each reported connection, it contains the time it was recorded, the server IP address, the cluster the server belongs to, the anonymized client IP address, and various connection statistics such as bytes sent/received, duration, packet count and RTT. The CDN operates a number of services, utilizing the same infrastructure, such as dynamic and static web pages delivery, cloud acceleration, and video streaming.

ISP Dataset: The ISP dataset contains two parts. First, detailed network information about the tier-1 ISP, including the backbone topology, with interfaces and link annotations such as routing weights, as well as nominal bandwidth and delay. It also contains the full internal routing table which includes all subnets propagated inside the ISP either from internal routers or learned from peerings. The ISP operates more than 650 routers in about 500 locations (PoPs), and 30 peering points worldwide. We analyzed more than 5 million routing entries to derive a detailed ISP network view.

The second part of the ISP dataset is an anonymized packet-level trace of residential DSL connections. Our monitor, using Endace monitoring cards [46], observes the traffic of around $20,000$ DSL lines to the Internet. We capture HTTP and DNS traffic using the Bro IDS [125]. We observe 720 million DNS messages and more than 1 billion HTTP requests involving about 1.4 million unique hostnames. Analyzing the HTTP traffic in detail reveals that a large fraction it is due to a small number of CDNs, including the considered CDN, hyper-giants and one-click-hosters [95, 67, 106] and that more than 65% of the traffic volume is due to HTTP.

To derive the needed traffic matrices, on an origin-destination flow granularity, we compute from the DSL traces (on a 10-minute time bin granularity) the demands for the captured location in the ISP network. This demand is then scaled according to the load imposed by users of the CDN to the other locations in the ISP network. For CDNs without available connection logs, we first identify their infrastructure locations using the infrastructure aggregation approach as proposed by Poese et al. [127] and then scale the traffic demands according to the available CDN connection logs.

4.4.5 Evaluation

In this section we quantify the benefits of using NetPaaS. For our evaluation we rely on traces from the largest commercial CDN and the tier-1 ISP described in Section 4.4.4. We start by presenting the traffic characteristics of the CDN inside the ISP and discuss the rationale for NetPaaS. We then evaluate the benefits of NetPaaS in the emulation environment described in [128].

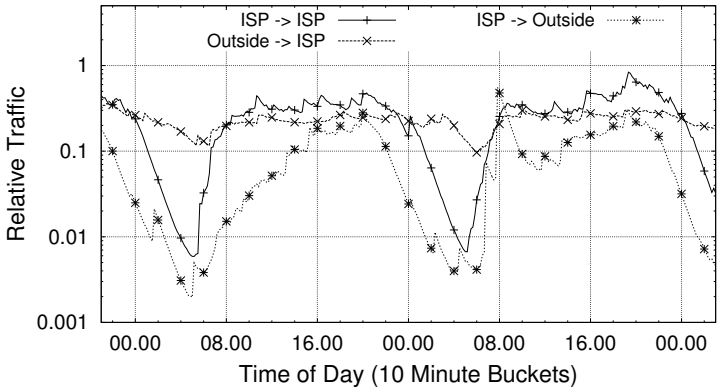


Figure 4.10: Activity of CDN in two days.

4.4.5.1 Collaboration Potential

We first describe our observations on the traffic and deployment of the large commercial CDN inside the tier-1 ISP and analyze the potential benefits of CDN-ISP collaboration. In Figure 4.10, we plot the normalized traffic (in log scale) from CDN clusters over time. We classify the traffic into three categories: *a)* from CDN servers inside the ISP to end-users inside the ISP (annotated ISP $\rightarrow$ ISP), *b)* from servers outside the ISP to end-users inside the ISP (annotated outside $\rightarrow$ ISP), and *c)* from CDN servers inside the ISP to end-users outside the ISP (annotated ISP $\rightarrow$ outside).

We observe the typical diurnal traffic pattern and a daily stability of the traffic pattern. Over the two week measurement period, 45.6% of the traffic belongs to the ISP $\rightarrow$ ISP category. 16.8% of the traffic belongs to the outside $\rightarrow$ ISP category. During peak hours, outside $\rightarrow$ ISP traffic can grow up to 40%. Finally, 37.6% of the traffic is served by inside clusters to outside end-users. Our first important observation is that a significant fraction of the CDN traffic is served from servers outside the ISP despite the presence of many servers inside the ISP that would be able to serve this traffic.

Figure 4.11 shows the re-allocation of traffic that would be possible using user-server assignment. Each full bar shows the fraction of traffic currently traversing a given number of router hops within the ISP network. In this evaluation, we only consider the end-users inside the ISP. The bar labeled "N/A" is the traffic of the outside $\rightarrow$ ISP category. The different shaded regions in each bar correspond to the different router hop distances after re-allocation of the traffic. Almost half of the traffic currently experiencing 3 hops can be served from a closer-by server. Overall, a significant fraction of the traffic can be mapped to closer servers inside the ISP. Note that the tiny amount of traffic for router hop count 0 and 1 is due to the topology design of the ISP network: either the traffic stays within a PoP or it has to traverse at least two links to reach another PoP.

In Figure 4.12, we show the traffic demand towards the CDN generated by each PoP. We observe that some PoPs originate high demand while others have limited demand, if any. Manual inspection reveals that some of the PoPs with high demand cannot be served by a close-by CDN server, while other low demand PoPs have a cluster near by. Variations in the demand over time exhibit even more significant mismatches between demand and CDN locations. With such a time-varying demand and the timescales at which CDN deployments

Figure 4.11: Potential hop reduction by using NetPaaS.

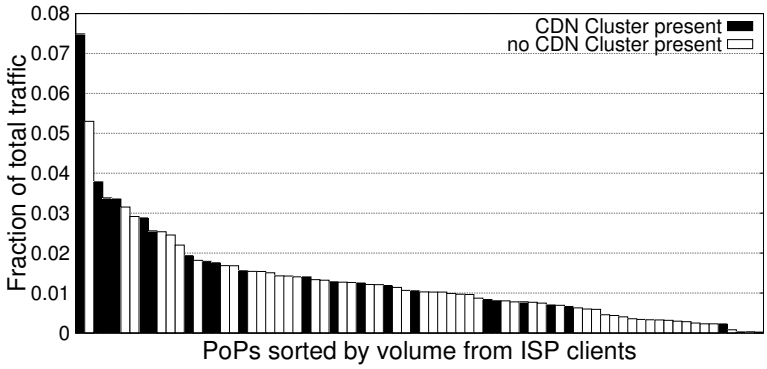


Figure 4.12: Traffic demand by ISP network position.

take place today, such mismatches should be expected.

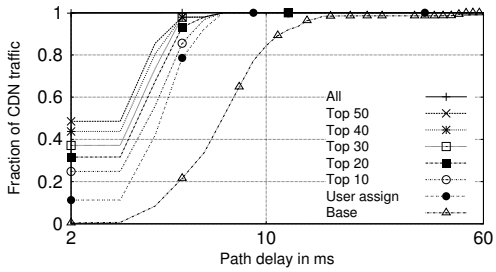
We conclude that there are ample opportunities for CDNs to benefit from collaboration with ISPs to re-arrange or expand their footprint. Also, these observations support the use of NetPaaS to improve the operation of both the CDN and the ISP in light of the new CDN-ISP strategic alliances [1, 16, 17, 14].

4.4.5.2 Improvements with NetPaaS

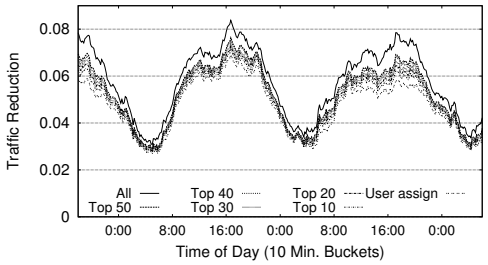
In this section we quantify the benefit of NetPaaS for the large commercial CDN inside the tier-1 ISP. First we show the benefits of user-server assignment for the existing CDN infrastructure and continue with the additional benefit of server allocation. In our evaluation we ensure that NetPaaS respects the available CDN server capacities and specifications in different locations. In the rest of the section, unless otherwise mentioned, we optimize the delay between end-user and CDN server [119]. Moreover, as we will show in our evaluation, by optimizing the delay between end-user and CDN server other traffic engineering goals are achieved.

4.4.5.2.1 Informed End-user to Server Assignment We first evaluate the benefits NetPaaS can offer when using user-server assignment only for the already deployed infrastructure of the large commercial CDN. In Figure 4.13(a) we show the current path delay between end-user and CDN servers, annotated as "Base". When using user-server assignment, annotated as "User assign", the delay is reduced by 2–6 msecs for most of the CDN traffic and another 12% of all traffic can be fetched from nearby CDN servers, a significant performance gain. To achieve similar gains CDNs have to rely on complicated routing tweaks [94].

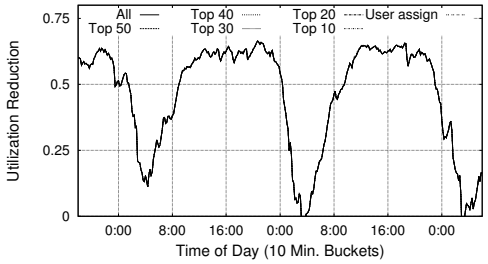
When utilizing NetPaaS for user-server assignment the traffic traverses a shorter path within the network.



(a) Improvements in user to server delay.



(b) Total traffic reduction within the network.



(c) Maximum link utilization reduction.

Figure 4.13: Utilizing NetPaaS for user-server assignment and server allocation.

This yields an overall traffic reduction in the network. In Figure 4.13(b) we plot the reductions in the overall traffic within the network, labeled "User-assign". The reduction can be as high as 7% during the peak hour. This is a significant traffic volume that is on the scale of tens to hundreds of Terabytes per day in large ISPs [3, 11]. As a consequence, the most congested paths are circumvented, as the full server and path diversity is utilized [129]. Our evaluation shows that user-server assignment significantly improves CDN operation with the already deployed infrastructure and capacity. Moreover, the ISP does not need to change its routing, thus reducing the possibility of introducing oscillations [63].

In Figure 4.13(c) we plot the reduction in utilization for the most congested link at any point of time. We observe that during the peak time the utilization of the most congested link can be reduced by up to 60%. This is possible as traffic is better balanced and the link is utilized to serve mainly the local demand. Such a reduction in utilization can postpone link capacity upgrades.

4.4.5.2.2 In-network Server Allocation We next evaluate the benefits of NetPaaS when server allocation is used in addition to user-server assignment. For short term CDN server deployments virtualized servers offer flexibility. For long term deployments, especially in light of the CDN-ISP alliances [1, 16, 17, 14], bare metal servers offer better performance. As our evaluation shows, the optimized placement of servers improves end-user performance as well as server and path diversity in the network, and enables ISPs to achieve traffic engineering goals.

To estimate the locations for installing new servers, we use the local search heuristic to approximate the solution of CFL (see Section 4.4.3.2.3). Figure 4.14 shows the accuracy of server allocation in terms of delay reduction when deploying 30 and 50 additional servers, labeled "Top 30" and "Top 50" respectively (similar observations are made for other numbers of servers). Notice that these 30 or 50 servers are not necessarily in the same PoP. It can be the case that more than one server is in the same PoP. For the optimal cases we

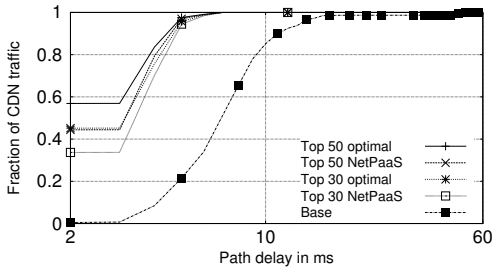


Figure 4.14: NetPaaS accuracy in selecting server location.

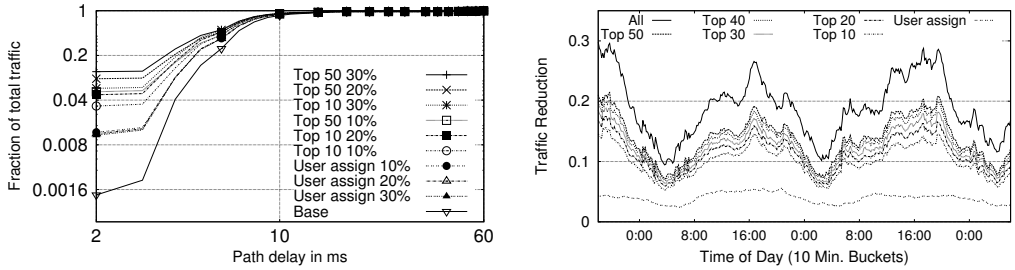
pre-computed the best server locations based on the full knowledge of our 14-days dataset, while NetPaaS calculates the placement by utilizing past traffic demands and the current network activity during runtime. Our results show that NetPaaS achieves gains close to those of the optimal placement.

In Figure 4.13(a) we show the delay improvements of NetPaaS when less than 10% of the servers are utilized, thus we range the number of servers between 10 to 50 servers that are allocated in any of the about 500 locations within the ISP, labeled "Top 10" to "Top 50". We also include a case where servers are allocated in all possible locations, labelled "All". As expected, in this case, nearly all traffic can be served from the same PoP as the end-user. Yet, with only 10 additional servers around 25% of the CDN demand can be satisfied in the same PoP. With 50 additional servers it is possible to satisfy more than 48% of the CDN demand by a server located in the same PoP as the end-users. This shows that a relatively small number of servers can reduce the user to server delay significantly. It also shows the impact that the placement of a server plays in reducing the delay between end-user and content server. Note, that we report on the reduction of the backbone delay, the reduction of the end-to-end delay is expected to be even higher as the server is now located in the same network.

We next turn our attention to the possible traffic reduction in the network when NetPaaS is used. In Figure 4.13(b) we show the possible network wide traffic reduction with server allocation when 10 to 50 servers can be allocated by the CDN. The traffic reduction especially during the peak hour ranges from 7% with 10 additional servers and reaches up to 7.5% when 50 additional servers can be utilized. Again, this is a significant traffic volume that is on the scale of tens to hundreds of Terabytes per day in large ISPs. Note that the primary goal of NetPaaS was to reduce the end-user to server delay, not network traffic. If all available locations (about 500) are utilized by the CDN, then the total traffic reduction is around 8% during peak time. This shows that a small number of additional servers significantly reduces the total traffic inside the network. We also notice that our algorithm places servers in a way that the activity of the most congested link is not increased, see Figure 4.13(c). In our setting, further reduction of the utilization of the most congested link by adding more servers was not possible due to routing configuration.

4.4.5.3 Joint Service Deployment with NetPaaS

We next consider the case of a CDN or an application that is launched within an ISP by exclusively utilizing NetPaaS. Examples include ISP-operated CDNs, licensed CDNs, or application-based CDNs. The latter is already happening with Google Global Cache [12] and with Netflix Open Connect in North America and



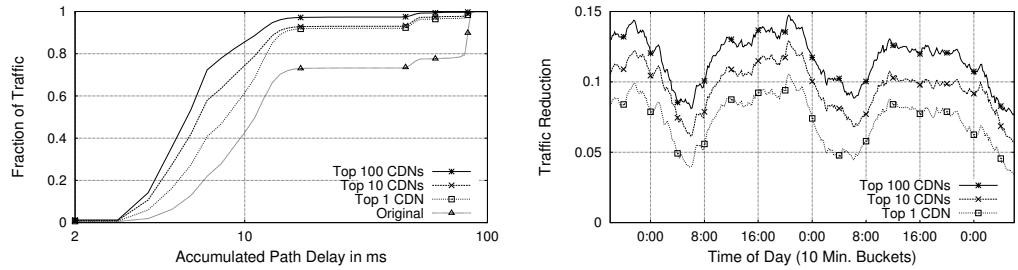
(a) Reduction in user-server delay by NetPaaS.

(b) Total traffic reductions by NetPaaS (30% CDN traffic).

Figure 4.15: Joint service deployment with NetPaaS.

North Europe [15]. Today, Netflix is responsible for around 30% of the total traffic in the peak hour in major US-based carriers [78]. In this section, we evaluate the performance of NetPaaS when such a service is launched using a CDN-ISP collaborative deployment scheme. In Figure 4.15 we show the benefits of a joint CDN-ISP server deployment within the network. For our evaluation, we use the large commercial CDN, for which we know the sources of the demand and the server specifications and locations, and scale its traffic to reach 10%, 20%, or 30% of the total traffic of the ISP. As previously, with NetPaaS and using only user-server assignment, it is possible to satisfy a significant fraction of the total traffic from close-by servers, see Figure 4.15(a). This can be even increased further when additional locations are available via server allocation. Our results also show that while increasing the traffic demand for the CDN, NetPaaS manages to keep the delay between users and servers low, as well as to reduce the total network traffic.

Figure 4.15(b) shows the total traffic reduction when the CDN traffic accounts for 30% of the total traffic. With user-server assignment only, NetPaaS is able to reduce the total traffic inside the network by up to 5%. When assigning additional servers, NetPaaS is able to reduce the total traffic from 15% with 10 servers to 20% with 50 servers. A traffic reduction of up to 30% is possible when additional servers can be allocated in all ISP PoPs.



(a) Reduction in user-server delay for top 1, 10, and 100 CDNs.

(b) Total network traffic reduction for top 1, 10, and 100 CDNs.

Figure 4.16: Improvements with NetPaaS when considering the top 1, 10, and 100 CDNs.

We also tested NetPaaS with multiple CDNs to evaluate the scalability of the system as well as the potential benefit of the system. For this, only user-server assignment was used as no information about the server requirements and the capacity of the other CDNs is available. We consider the top 1, 10, and 100 CDNs

by traffic volume in the ISP. The largest CDN accounts for 19% of the total traffic, the top 10 CDNs are responsible for more than 40% and the top 100 CDNs for more than 67% respectively. Most of the large CDNs have deployed distributed infrastructure, located in a number of networks [129]. Figure 4.16 shows the improvements in user-server delay as well as the total traffic reduction achieved by NetPaaS. For the largest CDN most of the traffic can be served from close-by servers and as a result the total traffic can be reduced by up to 10%. When turning our attention to the top 10 and top 100 CDNs, we observe that NetPaaS is able to further increase the improvements, but with diminishing returns. With the top 10 CDNs the traffic is reduced by up to 13% and with the top 100 CDNs 15% respectively. We conclude that by utilizing NetPaaS for the top 10 CDNs, it is possible to achieve most of the reduction in user-server delay and total traffic. We present a larger set of results for the top CDNs and an evaluation for a number of optimization goals under various network topologies in [64].

5 Conclusions

In this deliverable we presented the final results from the CHANGE project. We began by covering work on the CHANGE platform, whose final version concentrated on using Flowstream as its software controller, mSwitch as its high-speed software switch, and ClickOS as its virtualized, data plane implementation technology. We believe the performance results from this working prototype to be rather auspicious; for instance, mSwitch can switch packets on an inexpensive x86 server at rates of hundreds of Gigabits per second while ClickOS can saturate a 10Gb/s link while running over 100 concurrent virtualized middleboxes. We argue that such results show the feasibility of the concepts put forth in the Network Function Virtualization trend.

In addition, this document introduced Symnet, a novel static checking tool able to verify the correctness and security properties of networks containing stateful middleboxes. Because Symnet understands the Click configuration language, we are able to use it to check the properties of configurations installed on CHANGE platforms and on ClickOS. Further, this deliverable discussed Tracebox, a tool developed within the project to detect middleboxes in network paths and so useful for discovering troublesome inter-platform connections. We also presented an update on the inter-platform signaling framework which has been integrated with the Flowstream controller software introduced in previous deliverables.

Moreover, we described a number of applications developed for the CHANGE platform, including, but not limited to, firewalls, IDSes, DPIs, load balancers, carrier-grade NATs and software BRASes. For some of these we even went as far as testing their performance when running within a ClickOS virtual machine. The results were encouraging, with ClickOS being able to process packets at rates of 2-5 million packets per second on a single CPU core.

Finally, it is worth pointing out that many of the technologies (if not most) developed within the CHANGE project either already are or will be soon released as open source; the details of these releases are given in this document's appendix.

A Software Releases

In this brief appendix we list information about the open source releases for the various technologies described in this deliverable. For those that are not yet available, we explain what the release plan is.

(i) **netmap**: The netmap API provides high speed packet I/O and is freely available as open source at http://info.iet.unipi.it/ luigi/vale/ .

(ii) **mSwitch**: mSwitch is already available as open source software from the UNIPI website at http://info.iet.unipi.it/ luigi/vale/ .

(iii) **ClickOS**: We are in the process of cleaning up the sources and documenting code in order to release the code. This will be done in stages: first the Xen I/O acceleration pipe (relevant to the Xen and virtualization communities) and then the ClicKOS virtual machine. The release is planned for the first half of 2014.

(iv) **Tracebox:** Tracebox is already open sourced and available at http://www.tracebox.org.

(v) **Symnet:** We plan to release the code as open source sometime in 2014 (we have not done so yet as this code is relatively new and needs further testing).

(vi) **FlowOS:** The plan is to make the code available as open source in the first half of 2014.

(vii) **Signaling:** The plan is to make the code available as open source on github in the first quarter of 2014.

(viii) **ClickOS applications: (IDS, FW, DPI, FlowPinner)** The plan is to make the code available as open source in the first quarter of 2014.

(ix) **Suricata with Netmap support:** the source is already available in github at https://github.com/decanio/suricata-np waiting to be included into the Suricata's upstream code base.

Bibliography

[1] Akamai and AT&T Forge Global Strategic Alliance to Provide Content Delivery Network Solutions. http://www.akamai.com/html/about/press/releases/2012/press_120612.html.

[2] Akamai Aura Licensed CDN. http://www.akamai.com/html/solutions/aura_licensed_cdn.html.

[3] AT&T Company Information. http://www.att.com/gen/investor-relations?pid=5711.

[4] D2.4.: Flow processing architecture specification (final version).

[5] D3.1.: Platform controller design.

[6] D3.3: Flow processing platform: Main implementation.

[7] D4.2.: Inter-platform signaling (revised version).

[8] D4.3: Protocols and mechanisms to combine flow processing platform.

[9] D4.4: Network architecture: inter-platform communication software.

[10] D4.5: Network architecture.

[11] Deutsche Telekom ICSS. http://ghs-internet.telekom.de/dtag/cms/content/ICSS/en/1222498.

[12] GoogleCache. http://ggcadmin.google.com/ggc.

[13] The ini file format.

[14] KT and Akamai Expand Strategic Partnership. http://www.akamai.com/html/about/press/releases/2013/press_032713.html.

[15] Netflix Open Connect. https://signup.netflix.com/openconnect.

[16] Orange and Akamai form Content Delivery Strategic Alliance. http://www.akamai.com/html/about/press/releases/2012/press_112012_1.html.

[17] Swisscom and Akamai Enter Into a Strategic Partnership. http://www.akamai.com/html/about/press/releases/2013/press_031413.html.

[18] T-Systems to offer customers VMware vCloud Datacenter Services. http://www.telekom.com/media/enterprise-solutions/129772.

[19] Network Functions Virtualisation. SDN and OpenFlow World Congress, October 2012.

[20] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *USENIX Conference*, pages 93–112, 1986.

[21] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable Client Accounting for Hybrid Content-Distribution Networks. In *NSDI*, 2012.

[22] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated Data Placement for Geo-Distributed Cloud Services. In *NSDI*, 2010.

[23] B. Ager, N. Chatzis, A. Feldmann, N. Sarrar, S. Uhlig, and W. Willinger. Anatomy of a Large European IXP. In *SIGCOMM*, 2012.

[24] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig. Comparing DNS Resolvers in the Wild. In *ACM IMC*, 2010.

[25] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig. Web Content Cartography. In *ACM IMC*, 2011.

[26] Ole Agesen, Jim Mattson, Radu Rugina, and Jeffrey Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[27] K. Andreev, C. Garrod, B. Maggs, and A. Meyerson. Simultaneous Source Location. *ACM Trans. on Algorithms*, 6(1):1–17, 2009.

[28] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H.. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. UC Berkeley Technical Report EECS-2009-28, 2009.

[29] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit. Local Search Heuristics for k-Median and Facility Location Problems. *SIAM J. on Computing*, 2004.

[30] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proc. ACM/USENIX Internet Measurement Conference (IMC)*, October 2006.

[31] M. Axelrod. The Value of Content Distribution Networks. AfNOG 2008.

[32] F. Baker. Requirements for IP Version 4 routers. *Request for Comments 1812*, June 1995. http://ftp.ietf.org/rfc/rfc1812.txt.

[33] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[34] 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.

[35] R. Beverly, A. Berger, and G. Xie. Primitives for active Internet topology mapping: Toward high-frequency characterization. In *Proc. ACM/USENIX Internet Measurement Conference (IMC)*, November 2010.

[36] Bob Jenkins' Web Site. SpookyHash: a 128-bit noncryptographic hash. http://www.burtleburtle.net/bob/hash/spooky.html, October 2013.

[37] T. Bourgeau and T. Friedman. Efficient IP-level network topology capture. In *Proc. Passive and Active Measurement Conference (PAM)*, March 2013.

[38] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live Wide-Area Migration of Virtual Machines Including Local Persistent State. In *VEE*, 2007.

[39] Marco Canini, Wei Li, Andrew W. Moore, and Raffaele Bolla. GTVS: Boosting the Collection of Application Traffic Ground Truth. In *Proceedings of the First International Workshop on Traffic Monitoring and Analysis (TMA'09)*, May 2009.

[40] Marta Carbone and Luigi Rizzo. Dummynet revisited. *SIGCOMM Comput. Commun. Rev.*, 40(2):12–20, Apr 2010.

[41] Marta Carbone and Luigi Rizzo. An emulation tool for planetlab. *Comput. Commun.*, 34(16):1980–1990, Oct 2011.

[42] Alfredo Cardigliano, Luca Deri, Joseph Gasparakis, and Francesco Fusco. vpf_ring: towards wire-speed network monitoring using virtual machines. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 533–548, New York, NY, USA, 2011. ACM.

[43] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. *RFC 3234*, Feb. 2002.

[44] K. Church, A. Greenberg, and J. Hamilton. On Delivering Embarrasingly Distributed Cloud Services. In *HotNets*, 2008.

[45] Cisco. Cisco Cloud Services Router 1000v Data Sheet. http://www.cisco.com/en/US/prod/collateral/routers/ps12558/ps12559/data_sheet_c78-705395.html, July 2012.

[46] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design Principles for Accurate Passive Measurement. In *PAM*, 2000.

[47] B. Cohen. Incentives Build Robustness in BitTorrent. In *P2PEcon Workshop*, 2003.

[48] Virtual Computing Environment Consortium. http://www.vce.com.

[49] C. Contavalli, W. van der Gaast, S. Leach, and E. Lewis. Client subnet in DNS requests. draft-vandergaast-edns-client-subnet-01.

[50] E. Cronin, S. Jamin, C. Jin, A. Kurc, D. Raz, and Y. Shavitt. Constraint Mirror Placement on the Internet. *JSAC*, 2002.

[51] Luca Deri. Direct NIC Access. http://www.ntop.org/products/pf_ring/dna/, December 2011.

[52] G. Detal. tracebox, July 2013. See http://www.tracebox.org.

[53] D. DiPalantino and R. Johari. Traffic Engineering versus Content Distribution: A Game-theoretic Perspective. In *INFOCOM*, 2009.

[54] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[55] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[56] F. Dobrian, A. Awan, I. Stoica, V. Sekar, A. Ganjam, D. Joseph, J. Zhan, and H. Zhang. Understanding the Impact of Video Quality on User Engagement. In *SIGCOMM*, 2011.

[57] B. Donnet and T. Friedman. Internet topology discovery: a survey. *IEEE Communications Surveys and Tutorials*, 9(4), December 2007.

[58] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. In *Proc. ACM SIGMETRICS*, June 2005.

[59] D. Engler and M. Kaashoek. DPF: fast, flexible message demultiplexing using dynamic code generation. In *Proc. ACM SIGCOMM*, pages 53–59, 1996.

[60] ETSI. Leading operators create ETSI standards group for network functions virtualization. http://www.etsi.org/index.php/news-events/news/644-2013-01-isg-nfv-created, September 2013.

[61] ETSI Portal. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. http://portal.etsi.org/NFV/NFV_White_Paper.pdf, October 2012.

[62] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), Jan 2013.

[63] B. Fortz and M. Thorup. Optimizing OSPF/IS-IS Weights in a Changing World. *IEEE J. Sel. Areas in Commun.*, 2002.

[64] B. Frank, I. Poese, G. Smaragdakis, S. Uhlig, and A. Feldmann. Content-aware Traffic Engineering. *CoRR arXiv*, 1202.1464, 2012.

[65] FreeBSD-net Mailing List. ipfw meets netmap. http://lists.freebsd.org/pipermail/freebsd-net/2012-August/032972.html, 2013.

[66] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *Proc. ACM IMC*, pages 218–224, 2010.

[67] A. Gerber and R. Doverspike. Traffic Types and Growth in Backbone Networks. In *OFC/NFOEC*, 2011.

[68] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[69] H. Haddadi, G. Iannaccone, A. Moore, R. Mortier, and M. Rio. Network topologies: Inference, modeling and generation. *IEEE Communications Surveys and Tutorials*, 10(2):48–69, April 2008.

[70] 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.

[71] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[72] A. Heffernan. Protection of BGP Sessions via the TCP MD5 Signature Option. *RFC 2385*, Aug. 1998.

[73] B. Hesmans. Mbclick, July 2013. See https://bitbucket.org/bhesmans/mbclick.

[74] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it Still Possible to Extend TCP? In *Proc. ACM IMC*, pages 181–192, 2011.

[75] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP. In *Proc. ACM/USENIX Internet Measurement Conference (IMC)*, November 2011.

[76] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *Proc. ACM IMC*, 2011.

[77] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. LUA, an extensible extension language. *Software: Pactice & Experience*, 26(6):635–652, June 1996.

[78] Sandvine Inc. Global broadband phenomena. Research Report http://www.sandvine.com/news/global_broadband_trends.asp.

[79] Intel. Intel DPDK: Data Plane Development Kit. http://dpdk.org, September 2013.

[80] Intel. Intel Virtualization Technology for Connectivity. http://www.intel.com/content/www/us/en/network-adapters/virtualization.html, September 2013.

[81] Intel Corporation. Intel Turbo Boost Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html, September 2013.

[82] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. *RFC 1323*, May. 1992.

[83] W. Jiang, R. Zhang-Shen, J. Rexford, and M. Chiang. Cooperative Content Distribution and Traffic Engineering in an ISP Network. In *SIGMETRICS*, 2009.

[84] E. Katz-Bassett, H. Madhyastha, V. Adhikari, C. Scott, J. Sherry, P. van Wesep, A. Krishnamurthy, and T. Anderson. Reverse traceroute. In *Proc. USENIX Symposium on Networked Systems Design and Implementations (NSDI)*, June 2010.

[85] Mehul Chadha Kaushik Kumar Ram, Alan L. Cox and Scott Rixner. Hyper-switch: A scalable software virtual switching architecture. In *Proc. of USENIX Annual Technical Conference*, 2013.

[86] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, nsdi'13, pages 99–112, Berkeley, CA, USA, 2013. USENIX Association.

[87] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[88] A. Kivity, Y. Kamay, K. Laor, U. Lublin, and A. Liguori. Kvm: The linux virtual machine monitor. In *Proc. of the Linux Symposium*, 2007.

[89] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers not to the HiPPO. In *KDD*, 2007.

[90] 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.

[91] M. Korupolu, C. Plaxton, and R. Rajaraman. Analysis of a Local Search Heuristic for Facility Location Problems. *J. Algorithms*, 37:146–188, 2000.

[92] M. Korupolu, A. Singh, and B. Bamba. Coupled Placement in Modern Data Centers. In *IPDPS*, 2009.

[93] P. Krishnan, D. Raz, and Y. Shavitt. The Cache Location Problem. *IEEE/ACM Trans. Networking*, 8(5), 2000.

[94] R. Krishnan, H. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao. Moving Beyond End-to-end Path Information to Optimize CDN Performance. In *ACM IMC*, 2009.

[95] C. Labovitz, S. Lekel-Johnson, D. McPherson, J. Oberheide, and F. Jahanian. Internet Inter-Domain Traffic. In *ACM SIGCOMM*, 2010.

[96] N. Laoutaris, P. Rodriguez, and L. Massoulie. ECHOS: Edge Capacity Hosting Overlays of Nano Data Centers. *ACM CCR*, 38(1), 2008.

[97] N. Laoutaris, G. Smaragdakis, K. Oikonomou, I. Stavrakakis, and A. Bestavros. Distributed Placement of Service Facilities in Large-Scale Networks. In *INFOCOM*, 2007.

[98] T. Leighton. Improving Performance on the Internet. *CACM*, 2009.

[99] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *IMC*, 2010.

[100] H. H. Liu, Y. Wang, Y. Yang, H. Wang, and C. Tian. Optimizing Cost and Performance for Content Multihoming. In *SIGCOMM*, 2012.

[101] M. Luckie, Y. Hyun, and B. Huffaker. Traceroute probe methode and forward IP path inference. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, October 2008.

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

[103] Anil Madhavapeddy, Richard Mortier, Ripduman Sohan, Thomas Gazagnaire, Steven Hand, Tim Deegan, Derek McAuley, and Jon Crowcroft. Turning down the lamp: software specialisation for the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[104] H. Madhyastha, J. C. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, and A. Vahdat. scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs. In *FAST*, 2012.

[105] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[106] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On Dominant Characteristics of Residential Broadband Internet Traffic. In *IMC*, 2009.

[107] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. Enabling fast, dynamic network processing with clickos. In *HotSDN*, pages 67–72, 2013.

[108] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. *RFC 2018*, Oct. 1996.

[109] A. Medina, M. Allman, and S. Floyd. Measuring interactions between transport protocols and middleboxes. In *Proc. ACM SIGCOMM Internet Measurement Conference (IMC)*, October 2004.

[110] Microsoft. Patch available to improve TCP initial sequence number randomness. Microsoft Security Bulletin MS99-066, Microsoft, October 1999. See http://technet.microsoft.com/en-us/security/bulletin/ms99-046.

[111] Microsoft Corporation. Microsoft Hyper-V Server 2012. http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx, September 2013.

[112] Minix3. Minix3. http://www.minix3.org/, July 2012.

[113] J.C. Mogul and S.E. Deering. Path MTU discovery. RFC 1191 (Draft Standard), Nov 1990.

[114] R. Morris, E. Kohler, J. Jannotti, and M. Kaashoek. The Click modular router. In *Proc. ACM SOSP*, pages 217–231, 1999.

[115] D. R. Morrison. Practical algorithm to retrieve information coded in alphanumeric. *J. of the ACM*, 1968.

[116] Nadav HarEl and Abel Gordon and Alex Landau and Muli Ben-Yehuda and Avishay Traeger and Razya Ladelsky. Efcient and Scalable Paravirtual I/O System. In *Proc. of USENIX Annual Technical Conference*, 2013.

[117] N.Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. On multi–gigabit packet capturing with multi–core commodity hardware. In *Passive and Active Measurement conference (PAM)*, 2012.

[118] CISCO Global Visual Networking and Cloud Index. Forecast and Methodology, 2011-2016. http://www.cisco.com.

[119] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network. *SIGOPS Rev.*, 44, 2010.

[120] Open vSwitch. Production Quality, Multilayer Open Virtual Switch. http://openvswitch.org/, March 2013.

[121] OpenVZ. Welcome to OpenVZ Wiki. http://wiki.openvz.org/Main_Page, July 2012.

[122] J. S. Otto, M A. Sánchez, J. P. Rula, and F. E. Bustamante. Content Delivery and the Natural Evolution of DNS - Remote DNS Trends, Performance Issues and Alternative Solutions. In *IMC*, 2012.

[123] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. *RFC 2663*, Aug. 1999.

[124] C. Paasch. Presentation ietf 87, July 2013. See http://tools.ietf.org/agenda/87/slides/slides-87-tcpm-11.pdf.

[125] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Com. Networks*, 1999.

[126] pfSense Forum. Benchmark of pf / ipfw / forwarding on FreeBSD-HEAD. http://forum.pfsense.org/index.php?topic=61572.0, 2013.

[127] I. Poese, B. Frank, B. Ager, G. Smaragdakis, and A. Feldmann. Improving Content Delivery using Provider-Aided Distance Information. In *ACM IMC*, 2010.

[128] I. Poese, B. Frank, S. Knight, N. Semmler, and G. Smaragdakis. PaDIS Emulator: An Emulator to Evaluate CDN-ISP Collaboration. In *SIGCOMM demo*, 2012.

[129] I. Poese, B. Frank, G. Smaragdakis, S. Uhlig, A. Feldmann, and B. Maggs. Enabling Content-aware Traffic Engineering. *ACM CCR*, 42(5), 2012.

[130] Gergely Pongracz, Lszl Molnr, and Zoltn Lajos Kis. Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK. In *Proc. of the Second European Workshop on Software Defined Networks*, 2013.

[131] J. Postel. Internet Control Message Protocol. RFC 792 (INTERNET STANDARD), Sep 1981. Updated by RFCs 950, 4884, 6633.

[132] J. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The Little Engine(s) That Could: Scaling Online Social Networks. In *SIGCOMM*, 2010.

[133] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the Electric Bill for Internet-scale Systems. In *ACM SIGCOMM*, 2009.

[134] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[135] K. K. Ram, A. L. Cox, and S. Rixner. Hyper-switch: A scalable software virtual switching architecture. In *Proc. USENIX 2013*, 2013.

[136] Kaushik Kumar Ram, Jose Renato Santos, Yoshio Turner, Alan L. Cox, and Scott Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 61–70, New York, NY, USA, 2009. ACM.

[137] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), Sep 2001. Updated by RFCs 4301, 6040.

[138] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proc. USENIX ATC*, 2012.

[139] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. USENIX Annual Technical Conference*, 2012.

[140] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet i/o in virtual machines. In *Proc. ACM/IEEE ANCS*, 2013.

[141] Luigi Rizzo, Marta Carbone, and Gaetano Catalli. Transparent acceleration of software packet forwarding using netmap. In Albert G. Greenberg and Kazem Sohraby, editors, *INFOCOM*, pages 2471–2479. IEEE, 2012.

[142] Luigi Rizzo and Giuseppe Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[143] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, Jul 2008.

[144] Pedro M. Santiago del Rio, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil. Wire-speed statistical classification of network traffic on commodity hardware. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, IMC '12, pages 65–72, New York, NY, USA, 2012. ACM.

[145] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[146] Scapy. URL http://www.secdev.org/projects/scapy/.

[147] G. Schaffrath, C. Werle, P. Papadimitriou, A. Feldmann, R. Bless, A. Greenhalgh, A. Wundsam, M. Kind, O. Maennel, and L. Mathy. Network Virtualization Architecture: Proposal and Initial Prototype. In *SIGCOMM VISA*, 2009.

[148] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[149] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM*, August 2012.

[150] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, 2012.

[151] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S Ratsanamy, and V Sekarl. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM*, 2012.

[152] Sourceware.org. The Newlib Homepage. http://sourceware.org/newlib/, September 2013.

[153] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), Oct 2000. Obsoleted by RFC 4960, updated by RFC 3309.

[154] A. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind Akamai (travelocity-based detouring). In *SIGCOMM*, 2006.

[155] Swig.org. Simplified Wrapper and Interface Generator. http://swig.org, September 2013.

[156] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar. Answering What-if Deployment and Configuration Questions with Wise. In *ACM SIGCOMM*, 2009.

[157] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.

[158] M. Torren. tcptraceroute - a traceroute implementation using TCP packets. man page, UNIX, 2001. See source code: http://michael.toren.net/code/tcptraceroute/.

[159] S. Triukose, Z. Al-Qudah, and M. Rabinovich. Content Delivery Networks: Protection or Threat? In *ESORICS*, 2009.

[160] V. Jacobson et al. traceroute. man page, UNIX, 1989. See source code: ftp://ftp.ee.lbl.gov/traceroute.tar.gz.

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

[162] VMware. VMware Virtualization Software for Desktops, Servers and Virtual Machines for Public and Private Cloud Solutions. http://www.vmware.com, July 2012.

[163] Vyatta. The Open Source Networking Community. http://www.vyatta.org/, July 2012.

[164] Y. A. Wang, C. Huang, J. Li, and K. W. Ross. Estimating the Performance of Hypothetical Cloud Service Deployments: A Measurement-based Approach. In *INFOCOM*, 2011.

[165] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *Proc. ACM SIGCOMM*, August 2011.

[166] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, Dec 2002.

[167] J. Whiteaker, F. Schneider, and R. Teixeira. Explaining Packet Delays under Virtualization. *ACM CCR*, 41(1), 2011.

[168] Wikipedia. Exokernel. http://en.wikipedia.org/wiki/Exokernel, July 2012.

[169] Wikipedia. L4 microkernel family. http://en.wikipedia.org/wiki/L4_microkernel_family, July 2012.

[170] Wikipedia. FreeBSD Jail. http://en.wikipedia.org/wiki/FreeBSD_jail, September 2013.

[171] Wikipedia. Solaris Containers. http://en.wikipedia.org/wiki/Solaris_Containers, September 2013.

[172] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never then Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[173] Xen Blog. Xen Network: The Future Plan. http://blog.xen.org/index.php/2013/06/28/xen-network-the-future-plan/, September 2013.

[174] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On static reachability analysis of ip networks. In *Proceedings of Infocom*, 2005.

[175] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. G. Liu, and A. Silberschatz. P4P: Provider Portal for Applications. In *SIGCOMM*, 2008.

[176] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Proc. of USENIX Annual Technical Conference*, 2013.

[177] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *CONEXT*, CoNEXT '12, 2012.

[178] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proc. ACM CoNEXT*, December 2013.

