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

# D4.5 – Network Architecture

Due date of deliverable: September 15, 2013

Actual submission date: October 15th, 2013

| | |
|---|---|
| Start date of project | October 1, 2010 |
| Duration | 39 months |
| Lead contractor for this deliverable | Université catholique de Louvain |
| Version | Final, April 30, 2014 |
| Confidentiality status | Public |

## Abstract

The CHANGE architecture, which is based around the notion of a flow processing platform, aims to re-enable innovation in the Internet. The processing that is done on the platforms is requested by agents. Before accepting to host the requested processing, the platform needs to make sure that the requested processing is secure with respect to the platform and other agents. Furthermore the platform can accept a request for processing only if it has currently sufficient resources to satisfy the requirements of the processing. This document, first presents a distributed operating system to be executed on the platforms. It notably provides a means to verify whether a requested processing respects a set of safety rules. Next, an approach to decide whether the platform can host a requested processing is presented. For each incoming request a new constraint satisfaction problem is solved. Finally this document also describes an update to the inter-platform signaling framework, to which inter-platform tunnel provisioning functionalities have been added.

## Target Audience

For the project participants this document describes mechanisms that can be used to ensure safety on CHANGE platforms. Moreover this document describes a technique to allocate new incoming network services individually. Finally the document also describes an update of the inter-signaling platform software to which inter-platform tunnel provisioning has been added. The readers are expected to be familiar with Internet protocols and earlier delivrables.

**Impressum**

| | |
|---|---|
| Full project title | CHANGE: Enabling Innovation in the Internet Architecture through Flexible Flow-Processing Extensions |
| Title of the workpackage | D4.5 – Network architecture |
| Editor | Florence Massen, Université catholique de Louvain |
| Project Co-ordinator | Adam Kapovits, Eurescom |
| Technical Manager | Felipe Huici, NEC |

This project is co-funded by the European Union through the ICT programme under FP7.

# Executive Summary

The CHANGE vision introduces the common concept of a flow processing platform, instantiated at critical points in the network. Although the platform and its interfaces are common, the processing performed must be programmable, allowing the network to evolve and support the needs of rapidly changing applications. Flow owners can then request processing on the platforms.

Platform owners may want to forbid some types of processing on their platform. On the other hand, the flow owners requesting the processing may want to ensure no unexpected behavior arises due to transparent middleboxes between platforms. Thus, platform owners may require a set of properties be enforced on requested processing, whereas the flow owner may require a set of properties be enforced on path segments connecting platforms. We address the problem of enforcing such demanded properties in the first part of this document. We show how heavy-weight mechanisms can be circumvented by using static analysis. The idea is that the requested processing, as well as the processing done by middle-boxes is specified. This specification corresponds to a configuration based on categorized Click elements. Using our Symbolic Execution tool we can analyze such configurations individually, but also the composition of configurations, to perform a reachability analysis and detect loops. All of this is wrapped into the CHANGE architecture, which is the main focus of this deliverable. More specifically, we describe the architecture, its implementation and evaluation results from several real-life use cases we implemented.

In the second part of this document we tackle the online problem of handling incoming processing requests from a resource point of view. A specification of the needed resources is associated with each new service request received at the platform. These resources are typically expressed in terms of CPU cores, memory, bandwidth etc. The system must then decide, given the services already running on the platform, whether sufficient resources are available on the platform to perform the requested service while guaranteeing required performances. Thus for each new service request a different problem must be solved. To do this we adapt a constraint satisfaction procedure we developed for the offline problem of assigning a known set of services to a platform. We also present different alternative components to handle a specific part of the procedure. We evaluate several configurations of the procedure on different variations of a same online scenario. The results show that all configurations perform similarly but also that the computational overhead of some of the configurations does not necessarily pay off in all cases.

The third part of this document presents a software update to the inter-platform signaling framework. We describe one of the more important latest additions to the framework, the inter-platform tunnel provisioning functionalities.

# List of Authors

| | |
|---|---|
| Authors | Costin Raiciu, Vladimir Olteanu, Matei Popovici, Radu Stoenescu, Felipe Huici, Mohamed Ahmed, Joao Martins, Georgios Smaragdakis, Mark Handley, Francesco Salvestrini, Nicola Ciulli, Pham Quang Dung, Florence Massen, Yves Deville |
| Participants | Polytechnic University of Bucharest, NEC Europe Ltd., Technische Universität Berlin, University College London, Nextworks s.r.l., Université catholique de Louvain |
| Work-package | WP4 – Network Architecture Implementation |
| Security | PUBLIC (PU) |
| Nature | Report (R) |
| Version | 1.0 |
| Total number of pages | 55 |

# Contents

**Executive Summary** 4

**List of Authors** 5

**List of Figures** 8

**List of Tables** 9

**1 Introduction** 10

**2 The CHANGE Architecture** 12

2.1 Introduction . . . . . 12

2.2 The CHANGE Architecture . . . . . 14

2.2.1 Overview . . . . . 14

2.2.2 The API . . . . . 15

2.2.3 Traffic Attraction . . . . . 17

2.2.4 Security . . . . . 18

2.2.5 A Unifying Example . . . . . 20

2.3 The Controller: Static Analysis . . . . . 21

2.3.1 SymNet: Symbolic Execution for Networks . . . . . 22

2.4 The CHANGE Platform . . . . . 23

2.5 Evaluation . . . . . 27

2.5.1 Platform Scalability . . . . . 27

2.5.2 Controller Scalability . . . . . 30

2.6 Use-cases . . . . . 33

3 Online Resource Allocation 37
3.1 Problem Description . . . 37
3.2 Notations . . . 38
3.3 Adaptation to the Online Problem . . . 38
3.3.1 Computing communication paths . . . 39
3.3.2 Online constraint satisfaction procedure . . . 39
3.4 Simulation experiments . . . 42
3.4.1 Experimental setup . . . 43
3.4.2 Experimental results . . . 43
3.5 Conclusion . . . 45

4 Updates to the inter-platform signaling framework 46
4.1 Consolidated inter-platform signaling framework software . . . 46
4.1.1 The inter-platform tunnels provisioning . . . 47
4.1.2 Signaling Manager updates . . . 48
4.1.3 Service Manager updates . . . 49
4.2 Tunnel setup example . . . 50
4.3 Conclusions . . . 52

Conclusion 53

References 54

# List of Figures

1.1 CHANGE architecture showing deployment at an edge ISP with three platforms and a common controller containing Symnet and the inter-platform signaling framework presented in this deliverable. Colored rectangles inside platforms denote instantiated processing modules. 10

1.2 CHANGE Platform showing where the different pieces of work developed in this project fit. 11

2.1 CHANGE architecture: access operators deploy processing platforms where they run their own middleboxes and processing for third parties. A controller deployed by the operator knows the network topology, router and middlebox configurations. Client requests are deployed by the controller after they are statically checked for safety. . . . . . . . . . . . . . . 14

2.2 Client request with a single processing module for simple UDP port forwarding. Packets are exchanged through custom To/FromNetfront elements. . . . . . . . . . . . . . . . . . . . . . 16

2.3 Testing the safety of processing modules. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.4 A simple example of static checking a Click configuration. . . . . . . . . . . . . . . . . . . . 21

2.5 A CHANGE platform running ClickOS VMs as well as more general (but sandboxed) Linux VMs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

2.6 ClickOS reactive architecture. A controller connected to the back-end software switch instantiates on-the-fly ClickOS VMs for each new flow. . . . . . . . . . . . . . . . . . . . . . 25

2.7 ClickOS reaction time for the first 15 packets of 100 concurrent flows. . . . . . . . . . . . . 25

2.8 100 concurrent HTTP clients retrieving a 50MB file through a CHANGE platform at 25Mb/s each. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

2.9 The X axis shows the number of VMs active when one VM is suspended or resumed. . . . . 25

2.10 Cumulative throughput when having a single ClickOS VM handle Click configurations for multiple clients. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

2.11 Throughput of a CHANGE platform servicing up to 1,000 clients with different number of VMs and clients per VM. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

2.12 Static analysis checking scales linearly with the size of the operator's network. . . . . . . . . 28

2.13 Cost of sandboxing in a CHANGE platform. . . . . . . . . . . . . . . . . . . . . . . . . . . 30

2.14 CHANGE platforms run many middleboxes on a single core with high aggregate throughput. 31

2.15 Mobile phones save energy when a CHANGE platform batches push traffic into larger intervals. 31

2.16 SCTP performance when tunneling over TCP and UDP, with different measured loss rates. . 33

2.17 Defending against a Slowloris attack with CHANGE . . . . . . . . . . . . . . . . . . . . . 33

2.18 75 PlanetLab clients downloading a 1KB file from the origin server or a 3-server CHANGE CDN. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

3.1 Network service templates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

# List of Tables

2.1 CHANGE security rules limit the type of processing allowed by third-party users and clients. 18

3.1 Description of constraint satisfaction procedure configurations . . . . . . . . . . . . . . . . 42

3.2 Description of network service samples . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

3.3 Sequence number of first rejected network service . . . . . . . . . . . . . . . . . . . . . . . 44

3.4 Total number of accepted network services . . . . . . . . . . . . . . . . . . . . . . . . . . 45

# Acronyms

**BGP** Border Gateway Protocol.

**CDN** Content Distribution Network.

**CLI** Command Line Interface.

**DAG** Directed Acyclic Graph.

**DDoS** Distributed denial-of-service.

**DNS** Domain Name System.

**DNSSEC** Domain Name System Security Extensions.

**DPI** Deep Packet Inspection.

**FPR** Flow Processing Route.

**FPRO** Flow Processing Route Object.

**FSM** Finite State Machine.

**HSA** Header Space Analysis.

**I&T** Integration & Testing.

**IDS** Intrusion Detection System.

**IP** Internet Protocol.

**IPS** Intrusion Prevention System.

**ISP** Internet Service Provider.

**MAC** Message Authentication Code.

**NAT** Network Address Translation.

**NIC** Network interface controller.

**NS** Network service.

**OS** Operating System.

| | |
|---|---|
| **P2P** | Peer-to-peer. |
| **PID** | Platform IDentifier. |
| **PM** | Processing module. |
| **RTT** | Round-Trip delay Time. |
| **SCM** | Software Configuration Management. |
| **SID** | Service IDentifier. |
| **STA** | State Transition Action. |
| **SW** | Software. |
| **TCP** | Transmission Control Protocol. |
| **TEP** | Tunnel End-Point. |
| **TEPD** | Tunnel End-Point Data. |
| **TID** | Tunnel IDentifier. |
| **ToR** | Top-of-Rack. |
| **TPID** | Tunnel Pair IDentifier. |
| **UDP** | User Datagram Protocol. |
| **VM** | Virtual Machine. |
| **XML** | Extensible Markup Language. |

# 1 Introduction

The main focus of this deliverable is to present the CHANGE architecture, its implementation, and evaluation results arising from it. The CHANGE architecture (see figure 1.1) is built around the notion of flow processing platforms. These platforms are instantiated at critical points in the network and allow to perform programmable processing. Flow owners can then request processing of their flows on these platforms and, if the request is accepted, processing modules are instantiated at one or several platforms. Figure 1.1 further shows where two of the components developed in the project fit into the architecture, namely Symnet (chapter 2) and the inter-platform signaling framework (chapter 4).

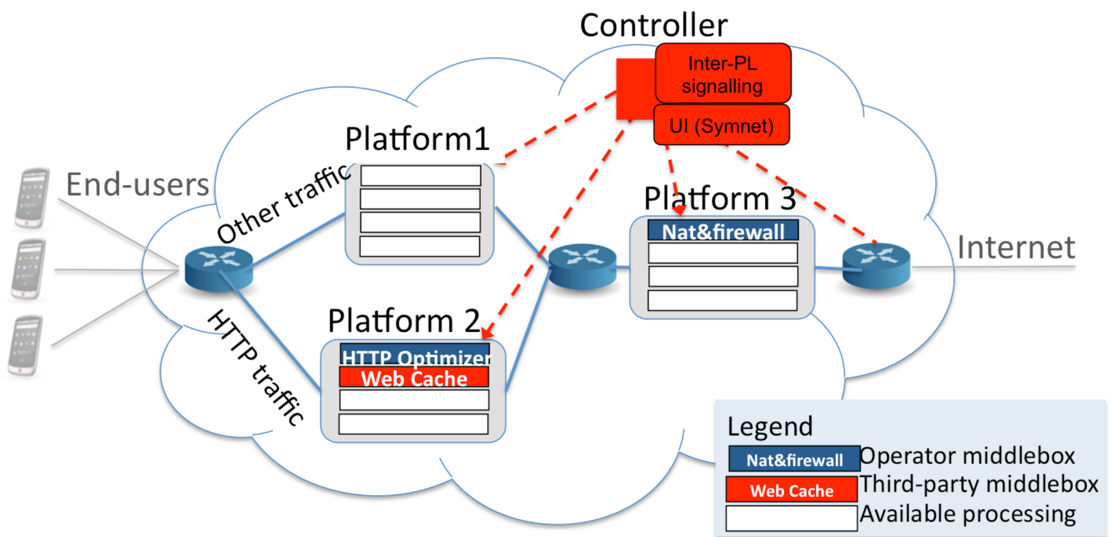


Figure 1.1: CHANGE architecture showing deployment at an edge ISP with three platforms and a common controller containing Symnet and the inter-platform signaling framework presented in this deliverable. Colored rectangles inside platforms denote instantiated processing modules.

Many of the remaining work developed within this project fit within the platform themselves (figure 1.2). First, the VALE high-speed software switch is used as a demux back-end between the processing modules (be them kernel threads as in FlowOS or virtual machines in the case of ClickOS) and the network cards; part of its performance is derived from the fact that it uses the netmap packet I/O framework. Second, FlowOS acts as one of the processing module implementations developed within CHANGE, with emphasis on easing development for flow-based processing (e.g., removing all ads in a particular HTTP flow). Third, ClickOS is another processing module implementation based around tiny, specialized virtual machines; it uses both the netmap API and the VALE software switch as network back-ends.

In addition to these data plane components, this deliverable covers, in chapter 3, the platform's online resource allocation mechanism, which is hosted within the platform's controller (again, shown in figure 1.2). Processing requests arriving at a platform provide information about the resources (CPU cores, memory and bandwidth) that are needed by the requested service. With each new request the platform then needs to decide whether its current available resources are sufficient to host the service. In a previous deliverable (D3.2, [1])

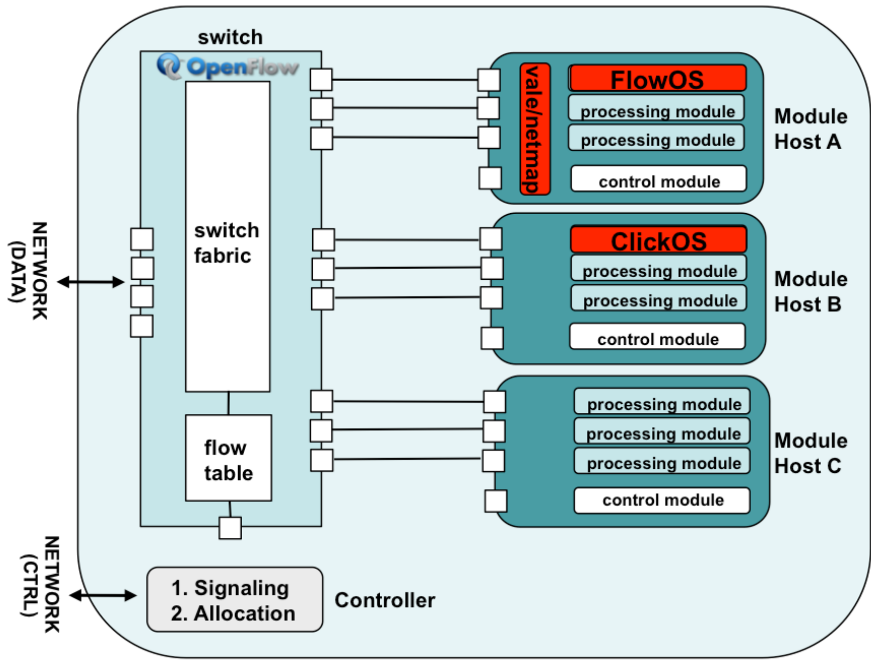


Figure 1.2: CHANGE Platform showing where the different pieces of work developed in this project fit.

a constraint satisfaction procedure, allowing to allocate resources to a set of such services known beforehand, was presented. In this deliverable we explain how this procedure can be adapted to an online context. In the online problem, the processing requests arrive over time, and nothing is known about future incoming requests. We extend the procedure previously described with further alternative components. Different configurations of the proposed procedure are tested on variations of an online scenario.

Finally, in chapter 4 we present an update to the signaling framework described in deliverable D4.4[3]. We describe how the framework, and more specifically the Service Manager and Signaling Manager, have been updated to provision inter-platform tunnels. An example is provided to show how newly introduced commands are used to set up a tunnel. The framework is meant to be run on the architecture's controller, as shown in figure 1.1, as well as on the platforms' controllers (figure 1.2).

# 2 The CHANGE Architecture

## 2.1 Introduction

Middleboxes have long been scorned by end-to-end purists for stifling innovation in the Internet and making networks difficult to debug. Meanwhile, they have proliferated to the point where most connections in the Internet pass through two types of middleboxes: *transparent* middleboxes are deployed at the first hop close to the client, in corporate networks and access networks, in order to optimize traffic and provide security for their clients; and *explicit* middleboxes such as front-end servers and load-balancers are deployed by content providers or third parties to enhance the client's connection performance (e.g., [12, 8]).

It is plain to see that innovation has not died in the Internet: networks have been constantly evolving despite protocol stagnation. Further, there has been a strong push from content providers to bring the processing closer to the user, either by relying on CDNs (e.g., Akamai [19, 16]) or by deploying hardware in access networks (e.g., Google Global Cache [5, 10] or Netflix Open Connect [6]). These explicit middleboxes split connections into a hop using traditional HTTP over TCP that passes most transparent middleboxes, and a backbone hop where any IP-based transport can be used, allowing innovation. Even end-user applications deploy middleboxes to bypass NATs, such as Skype supernodes or STUN servers.

While all of these examples show that in-network processing can help evolve the Internet, today's status quo also has a few major problems. Middleboxes are complex software that patch specific problems, and there is no coherent framework to develop or test them. Unfortunately, even operators do not always know exactly what each of their middleboxes does, and find it difficult to reason about how different middlebox processing composes when applied to the same traffic. This leads to **huge complexity in network management**: there is anecdotal evidence of certain operators being reluctant to remove old middleboxes from their network for fear of breaking it.

Secondly, **innovation is a select club**: middlebox deployments are only available to network operators and CDNs/major content providers, and out of the reach of the small/average content providers and most end-user applications. This limits the type of changes that can be made, and tilts the balance of power in the tussle between the end-points on one hand, and the network operators on the other.

Finally, **transparent middleboxes make the Internet unpredictable**. Connectivity is often limited to only a few protocols—mostly TCP, and then maybe only ports 80/443—and application optimizers sometimes enforce that all traffic look like today's traffic, going as far as even changing packets if needed. The end-systems' answer has been to tunnel traffic to get it through the network, and to encrypt it to evade application optimizers. This is a blow to operators, who cannot apply their security policies anymore, and hurts end-systems too because tunneling leads to poor performance in many cases (for example, using encryption is costly, with HTTPS downloads drawing 20% more energy compared to HTTP on a Samsung Galaxy Nexus device in some of our tests). None of these problems are critical today, but they are only bound to get worse as more and more middleboxes are deployed that look deeper into traffic to do their work, forcing clients to tunnel deeper and deeper to evade them.

The high-level question we ask in this paper is: **what changes are needed in access networks to spur Internet innovation, rather than stifle it?** Changing the Internet is notoriously difficult—as testified by the really slow adoption of IPv6 for example.

We take a step back and observe that all these problems would be feasible to solve if we could **reason about middlebox processing** at configuration time. If all parties involved knew a-priori what middleboxes would do when presented with certain traffic, network management would be a lot easier, operators could run in-network processing for third parties without security fears, and end-systems could enjoy better service and performance from the network they are connected to. Counter-intuitively, there are several potentially win-win scenarios that go unexploited because of this inability by the different parties involved to communicate and reason about the middleboxes' emergent behavior.

In this paper we introduce CHANGE , a novel architecture aimed at embracing software middleboxes in order to invigorate innovation in the Internet. CHANGE (1) allows **safe in-network processing for third-parties**, not only the rich few and (2) allows **end-users to query the network about changes to their traffic**, so that they may use, in collaboration with their operator, the most effective protocols for their applications, leading to win-win situations. We accomplish these high-level goals through the following contributions:

- The architecture's controller and a static checking tool called SymNet which together allow safe, in-network instantiation of network processing by third parties using an extended version of the Click modular router language (section 2.3). SymNet can perform stateful checking of large topologies consisting of over 1,000 nodes in seconds.
- An API and primitives through which different parties can request processing from the operator network, as well as a set of stock packet processing functionality so that clients can easily create middlebox configurations (section 2.2.2).
- Security rules to ensure that third-party instantiated processing cannot harm other parties, the network, or the Internet at large (section 2.2.4).
- The architecture's platform, which can efficiently run isolated middlebox processing on behalf of third-parties (section 2.4). The platform can handle as many as 1,000 concurrent clients, can instantiate processing in as little as 30 milliseconds, and can achieve cumulative throughputs of Gigabits per second while running up to 100 virtualized middleboxes.

We have built and tested the CHANGE architecture's components in isolation in order to understand their limits (section 2.5). In addition, we have deployed CHANGE in a wide-area testbed across several countries, and implemented seven different use cases that showcase the architecture's potential (section 2.6). Our experiments show that CHANGE scales to large numbers of users and brings performance benefits to end users, operators and content providers.

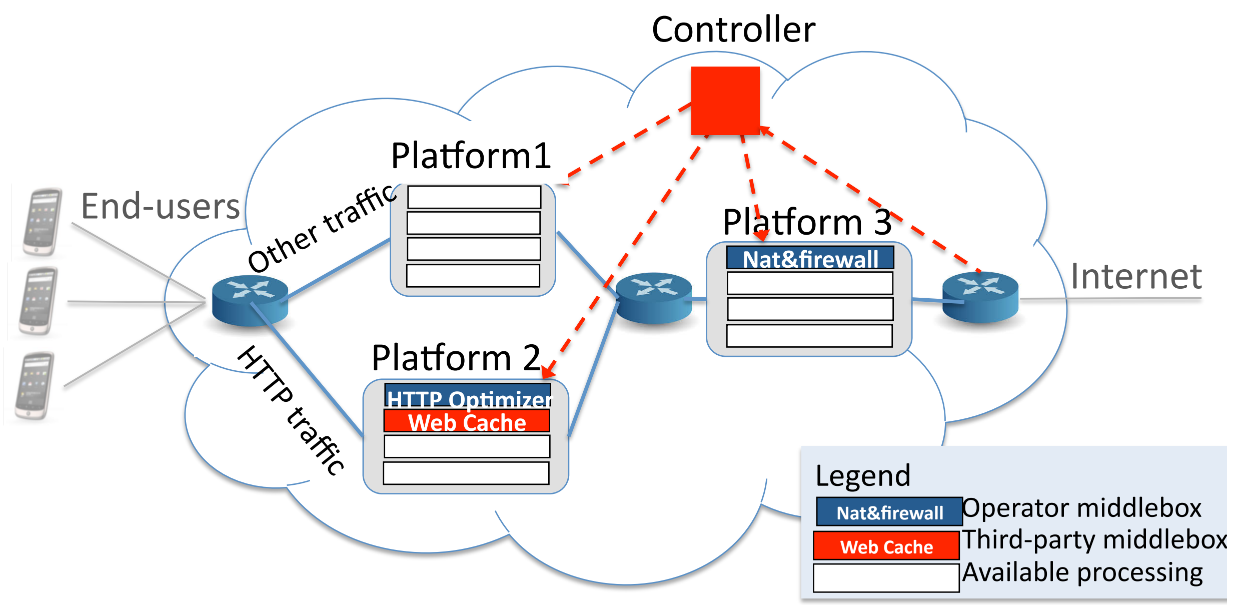


Figure 2.1: CHANGE architecture: access operators deploy processing platforms where they run their own middleboxes and processing for third parties. A controller deployed by the operator knows the network topology, router and middlebox configurations. Client requests are deployed by the controller after they are statically checked for safety.

## 2.2 The CHANGE Architecture

Our focus is on making pragmatic changes that are deployable. Consequently, we restrict our focus on access ("eyeball") networks, intentionally leaving out the core of the network. Focusing on access networks is a good place to start, as they concentrate most middleboxes processing today, and are the main focus of network function virtualization (NFV) [7]—a recent trend that aims to make middleboxes software running on commodity hardware.

At first sight, the shift to NFV might make transparent middlebox problems even worse, because we will have more such middleboxes that are dynamically instantiated and terminated. We believe the opposite: the trend towards re-architecturing middleboxes as software running on commodity hardware can be used to mitigate many of the problems transparent middleboxes create.

We do not want to eliminate transparent middleboxes, but to make the network more predictable such that end-points can take the right decisions when deploying new application or transport-level functionality. Counter-intuitively, throughout the paper and especially in section 2.6 where we describe our use cases, we show that both operators and endpoints can benefit from such a solution. **We propose that access network operators become miniature cloud providers, with a focus on running in-network functionality for themselves, their clients, or paying third parties.**

### 2.2.1 Overview

What are the basic components needed to make such an architecture viable? At the center of it is a set of general-purpose, software-based platforms deployed at the operator and able to carry out the necessary middlebox functionality (see figure 2.1); crucially, we show that such a platform can be built on inexpensive commodity hardware while still being able to service 1,000 isolated clients (section 2.5.1).

In addition, we need a controller in order to receive requests from clients and instantiate processing. One major obstacle is reassuring operators that third-party processing is safe to run—it does not harm other users or the operator itself—and does not break the operator's own policies. Our architecture's controller includes a static checking tool called SymNet that ensures this is the case; we cover its design and implementation in detail in section 2.3.

Beyond this, we need primitives and an API so that clients can express the type of processing desired (e.g., a firewall blocking all incoming traffic from a set of IPs representing spam servers, or ensuring that client-initiated TCP traffic to a certain port goes through the operator network unmodified). We further need a way for client packets to reach platforms, and certain security rules to regulate issues such as to whom a third-party instantiated middlebox should be allowed to send packets to. In the rest of this section we discuss each of these items in detail, and wrap up with an example that shows how all of the architectural components come together to provide safe, in-network processing.

### 2.2.2 The API

In-network processing is rather domain-specific - both input and output are network traffic. Most operations applied to packets are simple and include NATs, filters, tunneling, proxies, shaping, forwarding and so on, and arbitrarily complex functionality can be built by combining these basic building blocks and creating new ones.

That is why CHANGE clients express processing requests by using an extended version of the configuration language used by the Click modular router software [15]. The Click language has the notion of *elements*, small units of packet processing such as `DecIPTTL`, `Classifier`, `IPFilter` (and hundreds of others) which users interconnect into acyclical graphs called *configurations*.

In a CHANGE platform a *processing module* consists of an instantiation of such a configuration (e.g., a firewall or NAT). This can essentially take two forms: either a Click configuration using well-known Click elements; or, if such functionality is not sufficient, a pre-defined "stock" processing module offered by the platform and implemented with a software package other than Click, such as the content cache in our use cases. CHANGE further extends the language to allow it to specify processing spanning multiple boxes.

Besides easing client-side development of functionality, the great advantage of using the Click language is that, as long as a client's processing request is composed of known elements, we can statically model the processing done by such processing modules, which allows us to understand a-priori how a composed configuration behaves.

**Client Requests.** A client request contains two parts: (1) the *processing modules* to be instantiated and the *links* connecting them and (2) the client *requirements* that the configuration must satisfy (figure 2.2). A configuration contains any number of processing modules, links and requirements.

In addition, an operator can offer any number of custom processing modules to improve the service offered to

```
FromNetfront() ->
IPFilter(allow udp port 1500) ->
IPRewriter(pattern - - 172.16.15.133 - 0 0)
-> TimedUnqueue(120,100)
-> ToNetfront()

REQUIRE reach 0/0 172.16.15.133 udp 1500
```

Figure 2.2: Client request with a single processing module for simple UDP port forwarding. Packets are exchanged through custom To/FromNetfront elements.

clients. The controller implements a $listModules$ call that prints all the stock processing modules available to the client. Such modules can be implemented in various ways, from Click configurations to software running on commodity OSes. Our prototype controller, for instance, offers, in addition to modules based on Click elements, a reverse-HTTP proxy appliance, an explicit proxy (both based on squid), a DNS server that uses geolocation to resolve customer queries to nearby replicas and an arbitrary x86 VM where customers can run any processing. The latter offers great flexibility, at the cost of security and understanding of the processing; such VMs are sandboxed at runtime and are more expensive to run (see section 2.2.4 for a discussion of sandboxing).

Once the client has specified its processing modules, it can add links between these modules to obtain the desired processing logic. The links are also specified using Click syntax: $pm_1$ : $out_1$ -¿ $pm2$ : $in_1$ specifies that traffic leaving interface 1 on $pm_1$ will reach interface 1 on $pm_2$.

**Requirements** are of two kinds: *reachability* and *invariants*. The reachability requirement has the following syntax:

**REACH** $prefix|port\ prefix|port\ (tcpdump - rule)*$.

A reachability constraint states that the client expects traffic from a given prefix or port reaching another port or prefix to conform to the set of *tcpdump*-like rules enumerated. For instance, the client in figure 2.2 expects that Internet UDP traffic can reach its private IP address on port 1500.

Reachability requirements help the operator identify the appropriate platform to instantiate the client's processing modules, and to understand whether the requests are feasible given its own requirements. By analyzing client requirements over time, the operator can evolve its network by optimizing it to meet its clients needs.

Invariant requirements specify that the packet header fields (given in *tcpdump*-like syntax) do not change between the two given ports. For instance, a client may require that the TCP payload does not change by specifying *invariant 0/0 172.16.15.133 tcp payload*. The operator runs the invariant check on the boundaries of its network—thus the answer does not capture changes outside the operator's domain:

**INVARIANT** {PREFIX—PORT} {PREFIX—PORT} {HEADER}*

Reachability and invariant requirements are sufficient to test if the desired processing can take place and also to avoid unwanted processing from occurring along a given path.

**Operator configuration.** The operator describes its router-level network topology and routing tables as well as platform locations and addresses using similar syntax. This information is used by the controller to perform static analysis and to instantiate processing. Each network box (e.g., platforms and routers) is specified as a separate processing module, and a "legacy" flag is added to tell the controller not to instantiate this processing. Physical links are specified between these boxes using the Click syntax. When the operator wants to instantiate its own middleboxes, it uses the same API as regular clients, omitting the legacy flag.

The operator uses reachability and invariant statements to express its policy. In the example in figure 2.1 the operator specifies that all traffic reaching its HTTP optimizer must be HTTP: *reach 0/0 platform2:if allow tcp port 80 deny all* .

**Client configuration.** The client installs CHANGE software locally and configures it with the address of the controller it wishes to use. For end-users, this will be their operator's controller, and may be configured manually or automatically via DHCP extensions. For content-providers, we assume an out-of-band dissemination mechanism will be used instead. The client software authenticates itself to the controller through a public/private key-pair registered a-priori, when the end-user starts its relationship with the operator.

The client sends a request including processing modules, links and requirements. To implement the links that connect different processing modules, our controller instantiates all the processing modules requested by one user on a single platform, using a software switch to implement the links.

The controller assigns each processing request a client-unique identifier. For each processing module instantiated, the client is given an IP address/protocol/port combination that can be used to reach that module, useful for traffic redirection. For each requirement, a boolean answer is given. Clients can disable processing requests by quoting their identifier, thus stopping any processing modules that are active.

### 2.2.3 Traffic Attraction

CHANGE processing modules are **explicitly addressed**: an IP address (and maybe port) is assigned when processing is instantiated. For in-network processing to occur, traffic first needs to be attracted to a processing module running on a platform. The traffic attraction technique depends on the entity requesting processing. A **traffic source** (i.e., the initiator of a connection) that wants in-network processing can simply tunnel its traffic to the corresponding interface of a processing module. In contrast, a **traffic destination** will attract traffic to a platform by configuring dynamic DNS to give the platform's IP address in response to queries for its domain name.

A processing module will not usually be the communication's end-point, so it is necessary to steer the traffic onto the final destination. Such redirection can be implemented by the processing module either by rewriting the destination address, or by tunneling the traffic.

| Functionality | Third-party | Client | Operator |
|---|---|---|---|
| DPI | ✗ | ✗ | ✓ |
| NAT | ✗ | ✗ | ✓ |
| Transcoder | ✗ | ✗ | ✓ |
| Implicit Proxy | ✗ | ✗ | ✓ |
| Explicit Proxy | ✗ | ✓ | ✓ |
| Sendmail | ✗ | ✓ | ✓ |
| Rate limiter | ✓ | ✓ | ✓ |
| Firewall | ✓ | ✓ | ✓ |
| IDS | ✓ | ✓ | ✓ |
| IPS | ✓ | ✓ | ✓ |
| Scrubber | ✓ | ✓ | ✓ |
| Tunnel | ✓ | ✓ | ✓ |
| Multicast | ✓ | ✓ | ✓ |
| CDN | ✓ | ✓ | ✓ |
| DNS Server | ✓ | ✓ | ✓ |

Table 2.1: CHANGE security rules limit the type of processing allowed by third-party users and clients.

### 2.2.4 Security

In-network processing allows untrusted users to run custom software inside processing platforms operated by third parties. Security issues inside a processing platform are similar to those in public clouds, and the solutions we use are similar. Virtualization offers both performance and security isolation, and accountability ensures that users are charged for the resources they use, discouraging resource exhaustion attacks against platforms. However, to be certain that the processing cannot harm itself, other users of the platform, or the Internet at large, an additional set of security rules apply depending on the three types of users of CHANGE platforms: external third-parties (e.g., content providers), the operator's own customers, and the operator itself.

**1. External third-parties** can only process traffic destined to their IP addresses: our explicit addressing ensures that traffic reaching a user's processing module is destined to that user. What about outgoing traffic? Should third-party users be allowed to generate traffic to any destination? The answer is obviously *no*. So what should be allowed, then?

We cannot easily change the Internet to be default-off [11] and spoofing free[1] - but we can make sure that traffic generated by CHANGE platforms obeys these principles. In doing so we ensure that deploying in-network processing does not add to the long list of Internet security issues. To implement default-off, CHANGE requires that any third-party generated traffic must be authorized by its destination. Authorization can be explicit or implicit:

- **Explicit authorization** occurs when either a) the destination has requested the instantiation of the processing or b) the destination is a processing module belonging to the same user. In the first case, authorization will be performed when the user registers with the provider; we expect that a limited number of addresses will be registered, and that changing them will be infrequent. In the second case, the provider needs to implement a way to disseminate to all interested platforms the IP addresses

[1] All attempts to date have failed to gain any traction.

assigned to a certain user.

- **Implicit authorization** occurs when a host $A$ sends traffic to a processing module PM. If PM is the destination, then it is implicitly authorized to use $A$ as the destination address of the response traffic; this rule is similar in spirit to existing firewall behavior that allows incoming traffic corresponding to established outgoing connections. This rule directly allows CHANGE platforms to reply to traffic coming from regular end hosts (e.g., to implement a web server). If PM forwards traffic from $A$, to prevent spoofing, CHANGE ensures that traffic leaving PM has a) PM's IP as a source address *or* b) the same address as when the traffic entered PM.

These simple rules must be obeyed by all processing modules requested by third-party customers. When the processing requested is a stock module or a Click configuration, the controller can check at instantiation time whether the rules hold, meaning that no runtime checks are needed. When the user instantiates a virtual machine or custom Click elements, the platform monitors the user's traffic at runtime and enforces these rules.

**2. The operator's own customers** are also allowed to send traffic to any destination, without the destination explicitly agreeing. This merely extends the service already offered to customers to processing modules. This implies that customers can also deploy explicit proxies. However, addressing for these middleboxes is still explicit, forbidding NATs.

**3. The operator's processing modules** are allowed to generate traffic as they wish, reflecting the trust the operator places in them. Static analysis only helps the operator decide whether the box is achieving its intended purpose (i.e., correctness, as opposed to security).

**Enforcing security rules** requires a mix of static analysis and sandboxing. The operator's routing configuration together with explicit addressing ensure that a client's processing module will only receive traffic destined to it.

The controller runs static analysis on the client configuration by using the topology in figure 2.3. It runs three checks:

- A stand-alone reachability check from the processing module, where *each possible output port* is checked to see whether generated traffic has source address 172.100.10.1 and a destination in the explicitly approved list. This covers all the traffic generated by the PM in absence of incoming traffic.
- Next, a generic client $C$ generates traffic to the processing module. This implicitly authorizes return traffic from the PM to $C$. The output is again checked to see if the rules are obeyed, in particular: the source addresses of all outgoing packets, on all possible ports, have to be either $C$ or 172.100.10.1. This covers all stateless processing done by the PM.
- Finally, we run a full TCP reachability check. If the PM forwards traffic to other servers, we have these reply with $SYN/ACK$, and then check the behavior of the PM when it receives this traffic. This last check models the per-flow state of the PM.

If the rules are satisfied, the module is guaranteed to be safe, as long as individual Click elements are bug-free. If any of these static checks fail, there is a strong possibility that the PM disobeys the security policies. It may be that the faulty behavior is not triggered in real life, thus leading to false positive alerts. In this case, the operator has two options: it may sandbox the PM or refuse to instantiate it.

In order to sandbox a processing module we implement a new Click element called `ChangeEnforcer`. If the operator determines that a processing module requires sandboxing, the controller transparently instantiates two of these elements. The elements will run in the same VM as the user when a Click configuration is being run, or in a new VM when arbitrary code is run on behalf of the client. In section 2.3 we describe the controller and its static checking tool in greater detail, and in section 2.5 we quantify the performance costs of running a sandboxed processing module.

**Security Properties.** CHANGE has been explicitly designed to provide a default-off behavior for platforms and to prevent spoofing. CHANGE prevents, by default, Denial of Service Attacks, port scanning, and even outgoing mail for third-party processing. In table 2.1 we highlight the functionality CHANGE allows different types of users to run.

### 2.2.5 A Unifying Example

To make the description of the architecture more concrete, we now work through a simple example to illustrate how to actually use the CHANGE architecture. In particular, we target push notifications for mobiles, which are useful for a wide range of applications including email, social media and instant messaging clients. Push notifications involves the mobile device opening a long-running TCP connection to a cloud server and sending periodic keep-alives (roughly every minute) to ensure that NAT state does not expire. Apps wanting to notify the mobile contact the cloud server and, after authentication, are allowed to send data to the mobile.

Unfortunately, this mechanism allows applications to send as many messages as they wish, keeping the device's cellular connection awake and thus significantly draining its battery (as much as nine times faster according to [9]).

To improve this situation we leverage the CHANGE architecture. Assume a mobile customer wants to allow incoming notifications on port 1500 for UDP traffic. To express this requirement, the customer sends a request to its provider's controller asking: $reach\ 0/0\ IP_{client} : 1500\ udp$.

Next, the controller runs the reachability request against its static model of the topology (recall figure 2.1). Since the client sits behind Platform 3's NAT, the response is negative. Instead, the client can ask the provider to install processing on its behalf. The functionality of the processing module is very simple and shown in the client request in figure 2.2: traffic received on port 1500 by the processing module is forwarded to the client's IP address. Additionally, the module only allows notifications from certain IP addresses, and batches traffic with a period of 120 seconds to save energy. Upon receiving this request from a customer, the controller:

1. Finds a suitable platform to instantiate the requested processing. At every potential platform (three of them in our example topology) it uses static analysis to see if both the provider's and the customer's

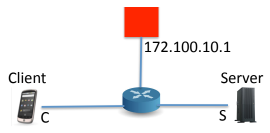


Figure 2.3: Testing the safety of processing modules.

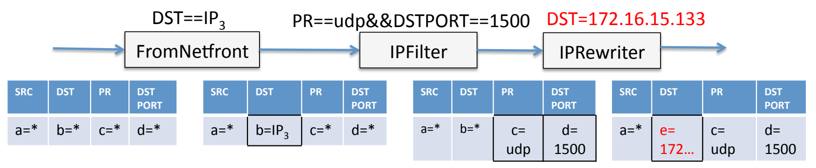


Figure 2.4: A simple example of static checking a Click configuration.

requirements can be met. In the example provided, only Platform 3 applies, since Platforms 1 and 2 are not reachable from the outside because they lie behind Platform 3's NAT.

2. If there are suitable platforms, it instantiates the processing; otherwise, it does nothing. In the example given the processing module is started on Platform 3.
3. Finally, it informs the client of the outcome (the external IP address of the processing module in this case).

We next discuss, in great detail, the controller's static checking tool (section 2.3) and our architecture's platform (section 2.4). We then provide an evaluation of their scalability when trying to service potentially thousands of concurrent clients (section 2.5), and follow that up with a set of illustrative use cases that showcase the architecture's potential (section 2.6).

## 2.3 The Controller: Static Analysis

In-network processing comes with tremendous flexibility but will be used only if both providers and users can reason about its emergent behavior. There are two sides to this problem, both related to the composition of the solution. First, providers must check whether customer processing is congruent with their own. Second, customers need to be assured that their processing is running correctly. Each of these two subproblems needs explicit support from CHANGE .

What is the best tool for this job? Statically checking network routing is a well-established topic, with reachability analysis as well as loop detection the strong candidates for verification [21]. More recently, Header Space Analysis (HSA) [14, 13] proposed a more general version of network static analysis that can also model arbitrary middleboxes as transformations of packet headers from input to output. HSA can find loops and run reachability analysis in networks with complex configurations.

Header Space Analysis could be used to check CHANGE configurations too, but it has two important shortcomings. First, HSA does not model middlebox state and therefore cannot capture the behavior of common middleboxes such as stateful firewalls or NATs. Secondly, HSA cannot discover properties that are common in practice. For example, consider analyzing reachability through a simple tunnel: in this case, HSA will create a header space representing all possible packets at tunnel ingress, after ingress the header space will be more specific (the source and destination addressed will be fixed), and finally at egress the header will revert back to showing all possible packets. A more useful analysis would output that any packet that exits the tunnel is the same as when it entered it.

### 2.3.1 SymNet: Symbolic Execution for Networks

This example suggests that symbolic execution, a technique prevalent in compilers, could be used for checking. We have designed and built SymNet, a tool that uses symbolic execution to statically analyze network configurations, including Click configurations and CHANGE routing configurations. Our tool extends Header Space Analysis – packet fields are symbolic variables that are tracked as they pass through the network. The set of possible values of a packet field, at any point on its path, is still a header space, thus we can run all the analysis that HSA enables, and more.

**Modeling packets.** We use symbolic packets to represent a set of packets with certain common constraints (e.g., all packets sourced by one host). A symbolic packet is a set of variables that model header fields and even the payload. Each variable can be free or bound to a symbolic expression. A symbolic expression is either another variable, set of variables, or set of possible values. Free variables can take any value, while bound variables express restrictions on the values of the corresponding field. For instance, a variable binding for the SYN Flag could specify that it is set to 1 or cleared.

**Modeling the topology.** The topology is given as a set of processing modules, each of which can have multiple interfaces; the provider and client configurations tell us which interfaces are connected, and what the invariants are. As in Header Space Analysis, we model processing modules as transformations on (symbolic) packets arriving on interfaces. To this end, we use *rules*. A rule is a pair of functions $(match, apply)$. The $match$ function is applied to a packet and an interface, and returns true only if that packet can arrive at the specified interface. The apply function takes as argument a packet and an input interface, and returns a list of (packet, output interface), where the packets are modified according to the processing module.

**Modeling flow state.** Certain processing such as those specific to tunnels or stateful firewalls require us to capture the *state* of the flow. Modeling state is achieved through using work variables. For instance, the entry-point of a tunnel can be modeled by: (i) pushing new, tunnel-dependent work variables in the flow, to store the content associated to certain header variables and (ii) modifying these header values appropriately. The tunnel exit point is modeled similarly: the content of the pushed variables is restored to the appropriate header variables.

Stateful firewalls receive similar treatment: a new, firewall-dependent variable is added to the packet to simulate flow state. At the receiver, this variable is copied into the return traffic. This allows the firewall to recognize the return packets as being part of an allowed outgoing connection.

**Reachability.** Computing reachability between a source $s$ and destination port $d$, amounts to: (i) taking a generic symbolic packet as seen at the source interface, (ii) applying all matching rules at a given platform and (iii) forwarding the resulting packets on all outgoing interfaces. The steps (i), (ii), and (iii) are repeated for each resulting pair of packets and outgoing interfaces until the destination is reached, on all possible paths.

The final result tells us which packets are accessible at destination port $d$ from source port $s$.

**Invariants.** To check invariants we run reachability between the source and destination ports. We then compare variables that model the invariants at the source and destination ports. The invariants hold only if the variables are bound to the same value.

**Loop detection** is accomplished by taking each interface from a given configuration and unfolding all possible paths starting at it. The exploration process will cease if no more paths exist or if an already visited interface has been reached. Loops will be reported iff: (i) a previously visited interface is reached and (ii) the current flow is more *general* or equal to the flow detected at the first visit. A flow is more general than another if all bound variables from the latter are also bound (and to the same values) in the former. Finally, we note that our loop detection mechanism is conceptually similar to the single infinite loop detection described in [14].

**Example.** Let's discuss how we can use SymNet to analyze the user-provided configuration in figure 2.2. We show how reachability is run in figure 2.4, where annotations above the Click elements show their SymNet model. The symbolic packet, shown below the elements, starts up with all header values set to unbound variables. `FromNetfront` only allows packets destined to this processing module, thus variable $b$ can only have value $IP_3$ after passing through this element. Next, `IPFilter` restricts the possible values for the proto (PR) and destination port header fields. Finally, `IPRewriter` changes the value of the destination header, setting it to a new variable $e$ whose value is bound to 172.16.15.133.

By checking the symbolic packet as it exits the configuration, the controller knows that the only possible destination address is 172.16.15.133—which is the address of the client in this case. The source address is the same as when it entered the configuration (variable $a$), so this processing module is safe to instantiate without any sandboxing.

**Implementation.** We have implemented SymNet in Haskell. Verification is achieved by performing reachability and loop detection tests on a Haskell description of the Click configurations or stock processing modules. We have modeled individual Click elements and stock processing modules manually.

The CHANGE controller is a front-end to SymNet that we have implemented in Scala. The controller parses client processing requests and automatically generates the appropriate Haskell model, runs the checker and, depending on the outcome, instantiates processing and directs traffic into the processing module by configuring the software switch running on the target platform. Finally, the controller replies to the client.

## 2.4 The CHANGE Platform

CHANGE platforms are based on Xen and so inherit the isolation, security and performance properties afforded by paravirtualization. As such, the platforms can run vanilla x86 VMs, though this is not our main target because it severely limits the scalability of the system to a few tens of users/VMs.

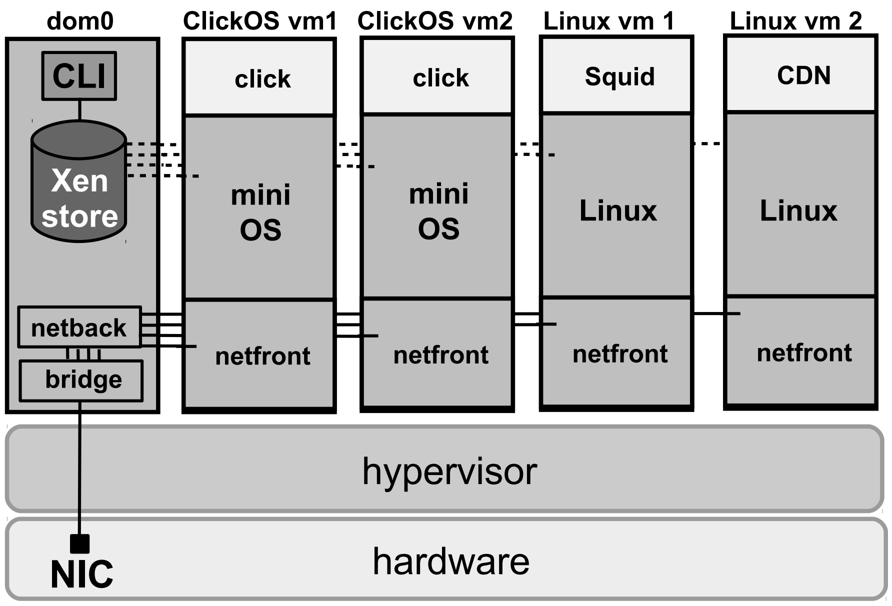


Figure 2.5: A CHANGE platform running ClickOS VMs as well as more general (but sandboxed) Linux VMs.

For the platform to be viable, it has to be able to support a potentially large number of concurrent clients while ensuring isolation between them. As a result, the CHANGE platforms rely on ClickOS, a guest operating system optimized to run Click configurations. ClickOS consists of a tiny Xen virtual machine built from (1) Mini-OS, a minimalist, single address-space, paravirtualized operating system distributed with the Xen sources, and (2) the Click modular router software [15]. We chose Xen because the split driver model affords us access to modern hardware and drivers (e.g., an Intel 10Gb card and the corresponding `ixgbe` driver) that a minimalist OS such as Mini-OS would not normally support.

Figure 2.5 gives an overview of the CHANGE platform based on ClickOS. The control plane consists of a python API and a CLI built on top of it, both running on the Linux-based dom0 domain. We rely on the XenStore, a proc-like database, to communicate control information between dom0 and the ClickOS VMs. For instance, we use it to emulate Click's element handlers, which under Linux depend on the proc filesystem or sockets to function.

ClickOS supports a large range of middleboxes such as firewalls, NATs, load balancers, and DNS proxies, with many more possible thanks to the over 200 Click elements built into the ClickOS VM image. Extending this with our own elements allowed us to implement other functionality such as a carrier-grade NAT and software BRAS.

Regarding networking, packets arrive at the NIC and are handled by the device driver in dom0. The interface is connected to a software bridge, either Open vSwitch or a modified VALE [20] switch for 10Gb/s experiments. The switch sends the packets to the "right" VM, and in particular a VM's virtual interface (vif). The vifs are then serviced by a *netback* driver residing in dom0 [2]. The netback driver exports a common, ring-based interface which *netfront* drivers in the virtual machines implement. In the case of ClickOS, a new Click

[2]Strictly speaking, the device drivers, switch, vifs and netback driver do not need to be in dom0 but rather a driver domain; in practice however, the dom0 often acts as the driver domain.

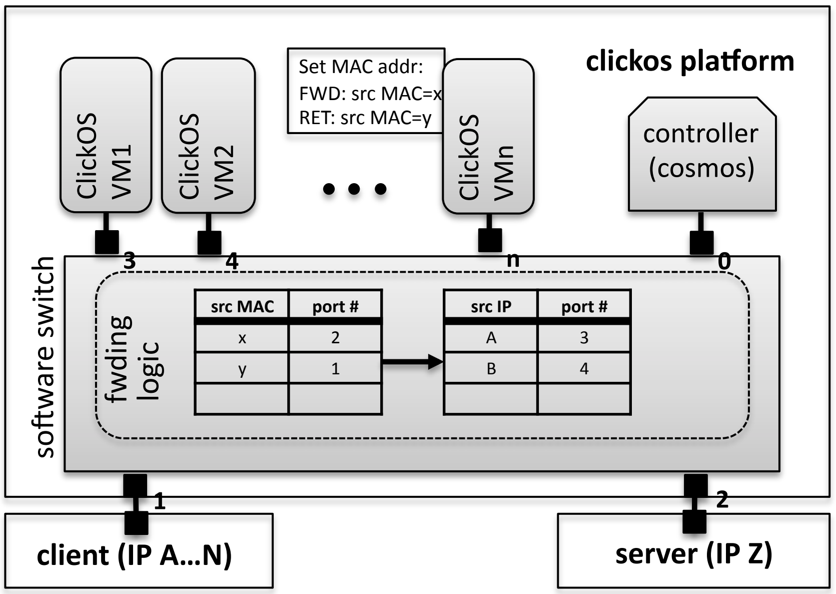


Figure 2.6: ClickOS reactive architecture. A controller connected to the back-end software switch instantiates on-the-fly ClickOS VMs for each new flow.

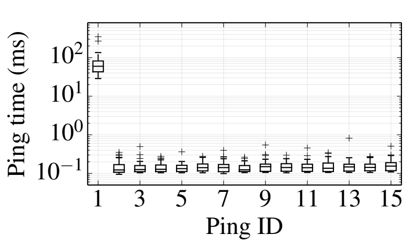


Figure 2.7: ClickOS reaction time for the first 15 packets of 100 concurrent flows.

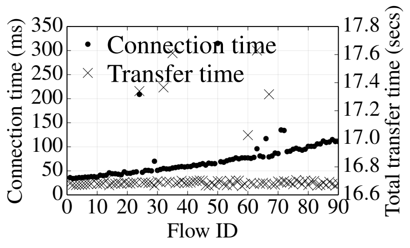


Figure 2.8: 100 concurrent HTTP clients retrieving a 50MB file through a CHANGE platform at 25Mb/s each.

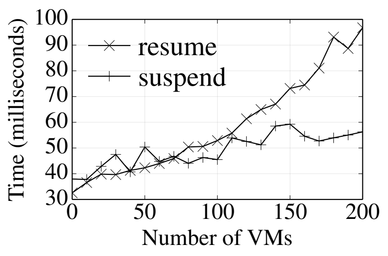


Figure 2.9: The X axis shows the number of VMs active when one VM is suspended or resumed.

element that we implemented called `FromNetfront`, takes care of receiving packets and sending them down the rest of the Click chain (sending packets follows the reverse path which ends at a `ToNetfront` element).

For the full details of ClickOS we refer the reader to [18]. In the rest of this section we focus on modifications and experimentation we carried out in order to build a viable CHANGE platform around ClickOS.

**Starting middleboxes on the fly.** By itself ClickOS goes a long way towards meeting our scalability requirements. ClickOS virtual machines are tiny (5MB when running) and this allows us to run up to 100 of them on inexpensive commodity hardware as explained below. Even so, a CHANGE platform is likely to have to service many more clients.

One key observation is that since ClickOS VMs can boot rather quickly (in about 30 milliseconds), we only have to ensure that the platform copes with the maximum number of concurrent clients at any given instant. Thus, we create ClickOS VMs *on-the-fly*, as the first packet belonging to a certain client arrives, while being transparent to the client.

To achieve this, we modify ClickOS' back-end software switch to include a switch controller connected to one of its ports (figure 2.6). The controller monitors incoming traffic and identifies new flows, where a new flow consists of a TCP SYN or UDP packet from a source IP address not yet seen [3].

In addition, we install a special module into the switch's forwarding logic that consists of a MAC table containing `<src MAC, port #>` tuples followed by another module with an IP table of `<src IP, port #>` tuples. When a packet from a new flow arrives at the switch, no entries for it exist on either table and so it gets sent to the controller. The controller then takes care of instantiating a new ClickOS VM to handle the flow and attaches the VM's virtual interface to the switch. Further, it installs an entry into the IP table directing packets coming from the source IP address to the port the new VM is attached to, and entries into the MAC table mapping a generated address in the 00:00:00:00:00:XX range to the switch port of each of the NICs on the system. Before sending packets back to the switch, each ClickOS VM sets their source MAC address to one of these addresses, depending on whether the packet came from the client or the server (figure 2.6 has an example with entries in the tables).

**Suspend and resume.** Creating VMs on the fly works great as long as the processing is stateless or only relevant to the single flow the VM was created to handle. For stateful handling, and to be able to still scale to large numbers of concurrent clients, we add support to Mini-OS to allow us to suspend and resume ClickOS VMs; we present evaluation results from this mechanism in section 2.5.

**Scalability via static checking.** The one-client-per-VM model is fundamentally limited by the maximum number of VMs a single box can run (a few hundred), and on-the-fly instantiation mitigates the problem but is no panacea. We could of course further increase capacity with servers with large numbers of CPU cores, or use additional servers, but this would just be throwing money at the problem.

Another approach is to run multiple users' configurations in the same ClickOS virtual machine, as long as we can guarantee that the users are properly isolated. To consolidate multiple users onto a single VM we create a Click configuration that contains all individual user configurations preceded by a traffic demultiplexer; no links are added between different users' configurations, and no elements instances are shared. Explicit addressing ensures that a client's module will only see traffic destined to it, and our security rules ensure that processing modules cannot spoof IP addresses.

Standard Click elements do not share memory, and they only communicate via packets. This implies that running static analysis with SymNet on individual configurations is enough to decide whether it is safe to merge them.

For Click elements that keep per-flow state, ensuring isolation is trickier: one user could force its configuration to allocate a lot of memory, DoS-ing the other users. To avoid such situations we would need to monitor and limit the amount of memory each configuration uses. A simpler solution is to not consolidate stateful processing—this is what our prototype does. In the next section we experimentally quantify the gains that can be had from this approach in terms of support for larger numbers of flows/clients.

[3]The choice of source IP address as a flow identifier is unimportant to the rest of the reactive mechanism and can be easily changed.

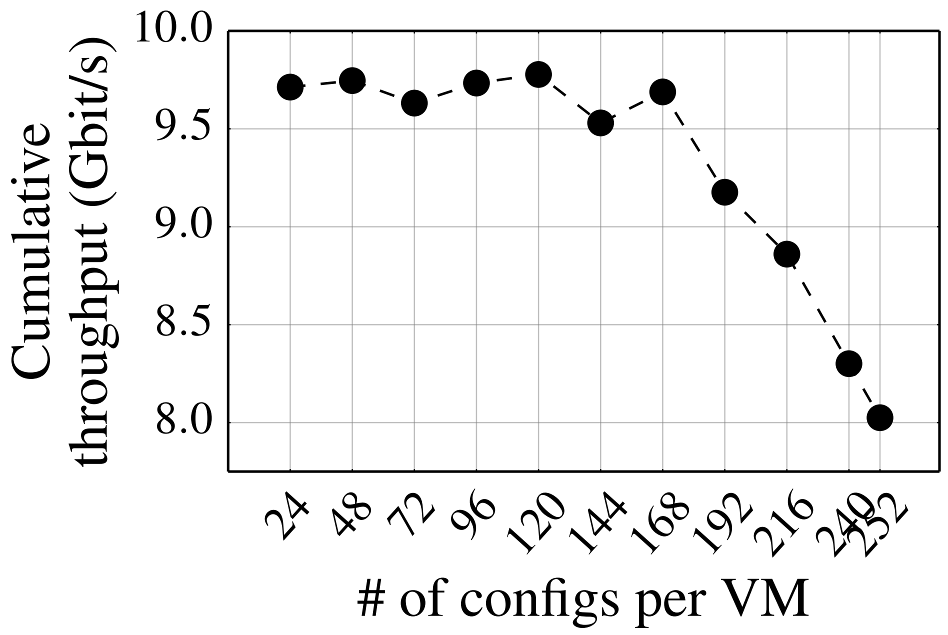


Figure 2.10: Cumulative throughput when having a single ClickOS VM handle Click configurations for multiple clients.

## 2.5 Evaluation

CHANGE relies on two main components: its scalable platform software based on ClickOS and SymNet, the static analysis tool running at the controller. In this section we evaluate the scalability of these two components.

### 2.5.1 Platform Scalability

It has been shown that a system with a few ClickOS VMs can process packet rates on the order of millions per second for basic appliances such as NATs [18]. Without any of our optimizations, around 100 ClickOS VMs can be run on a single commodity server without affecting total throughput, at least for trivial processing (packet forwarding).

Our main goal is to understand how many concurrent users can share a CHANGE platform while carrying out actual middlebox processing. First we test ClickOS's ability to quickly react to incoming traffic by instantiating on-the-fly virtualized middleboxes. We connect three x86 servers in a row: the first initiates ICMP ping requests and also acts as an HTTP client (curl), the middle one acts as the CHANGE platform and the final one as a ping responder and HTTP server (nginx). For the actual processing we install a stateless firewall in each ClickOS VM.

All measurements were conducted on an inexpensive (about $1,000), single-socket Intel Xeon E3-1220 system (4 cores at 3.1 GHz) with 16 GB of DDR3-ECC RAM running Xen 4.2.0. To obtain the best performance, we pin interrupts for the NIC connected to the client to core 0, interrupts for the other NIC to core 1, the switch controller to cores 1 and 2 and all ClickOS virtual machines to core 3.

In our first experiment we start 100 pings in parallel, with each ping sending 15 probes. Each ping is treated

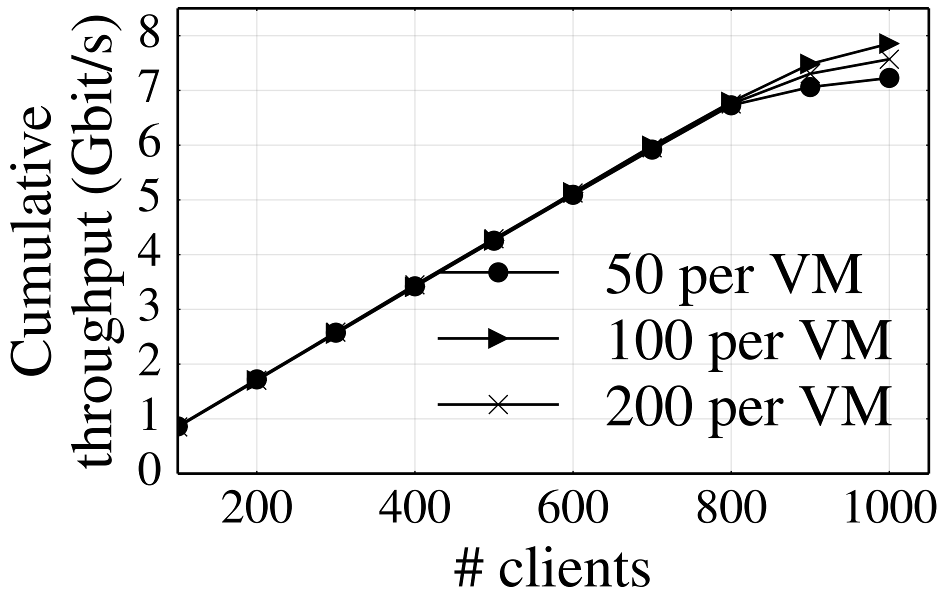


Figure 2.11: Throughput of a CHANGE platform servicing up to 1,000 clients with different number of VMs and clients per VM.

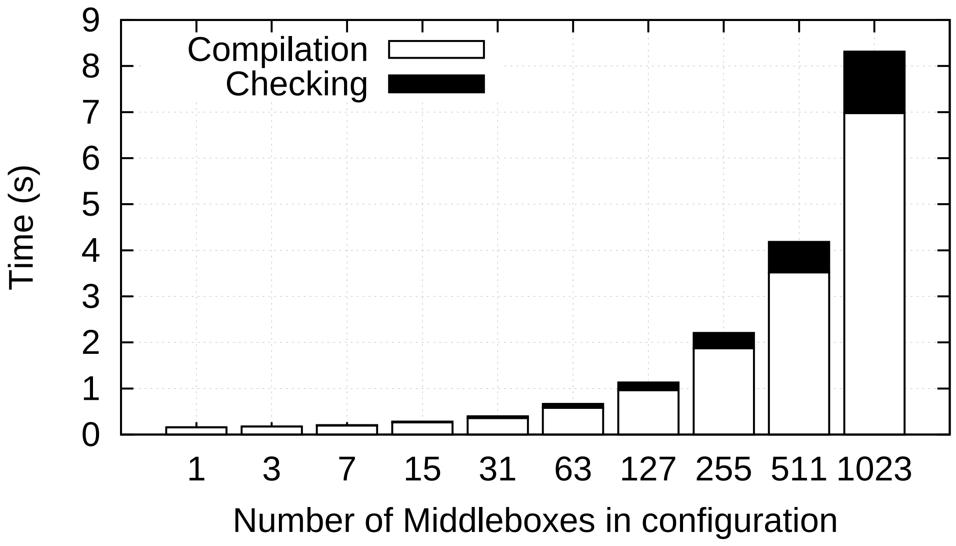


Figure 2.12: Static analysis checking scales linearly with the size of the operator's network.

by the platform as a separate flow, and a new VM is started when the first packet is seen. We expect that the first packet's delay should be reasonable despite VM creation, and that successive packets in the flow experience much lower delay. Figure 2.7 shows the results of this experiment. Clearly, the first packet in a flow is more costly than the ones that come after, but its round-trip time is still 50 milliseconds on average. For packets 2-15, the ClickOS VM is already in place, and so the RTT drops down significantly. The RTT increases as more and more ClickOS VMs are running on the system, but even with 100 VMs the ping time for the first packet of the 100th flow is 100 ms.

Next, we conducted a more realistic experiment consisting of an HTTP client initiating 100 concurrent HTTP requests capped at 25 Mb/s that go through the ClickOS box in order to reach an HTTP server hosting a 50MB file. Once again, this implies the reactive creation of 100 ClickOS VMs. We measured the time it takes for the connection to be set up (i.e., including VM creation) as well as the total time to transfer the file, and plot the results in figure 2.8.

Starting and terminating ClickOS VMs is ideally suited for stateless network processing, such as a plain firewall. At the extreme, we could instantiate a VM per packet, process it and terminate immediately. This would allow us to support, at least in theory, any number of users, at the cost of increased packet delays. The right granularity of VM creation can be optimized for different types of traffic, and is subject of our future work.

**Suspend and resume.** When VMs hold per-flow state, however, terminating a VM would effectively terminate the end-to-end traffic, which is unacceptable. The solution in this case is to use suspend/resume instead of terminate/boot. To this end we have implemented suspend and resume support in Mini-OS and performed experiments varying the number of active VMs when we suspend and resume a single VM. The results are shown in figure 2.9 and are comparable to the ClickOS boot times.

**Aggregating multiple users onto a single virtual machine.** Another way to scale the number of clients served by a CHANGE box is by *consolidating* the processing modules of different clients onto a single ClickOS virtual machine, with proper demultiplexing and multiplexing of traffic (recall that we had already proven that consolidation is safe as long as the processing is stateless). To understand the scalability of this optimization, we create a single ClickOS VM and install a Click configuration containing an `IPClassifier` element to act as a destination IP demuxer. For each client we run individual firewall elements, and then traffic is again multiplexed onto the outgoing interface of the VM.

We start up an increasing number of HTTP flows, one per client, and measure the cumulative throughput through our platform (figure 2.10). As shown, we can sustain essentially 10Gb/s line rate for up to over 150 clients or so, after which the single CPU core handling the processing becomes overloaded and the rate begins to drop. While the exact point of inflexion will depend on the type of processing (in this case firewalling) and the CPU frequency, these results hint that aggregating configurations onto a single VM is an easy way to increase the number of clients supported by a platform.

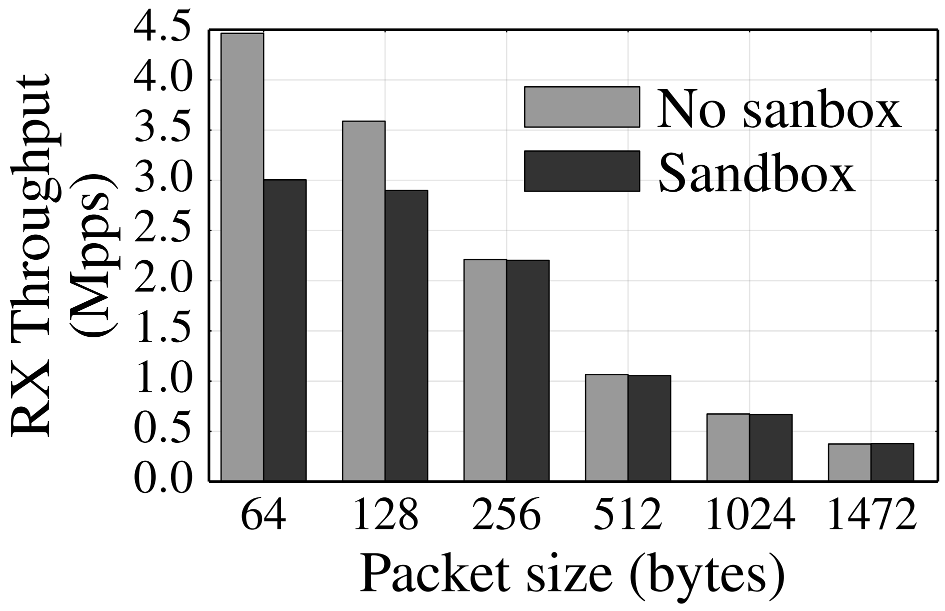


Figure 2.13: Cost of sandboxing in a CHANGE platform.

Finally, we gradually increase the number of clients up to 1,000 by leveraging SymNet's ability to verify that running configurations from different clients on shared VMs is safe. We try different numbers of clients per VM ($n$=50, 100 or 200), where each client is downloading a web-file at a speed of 8Mbps and the $n$'th client triggers the creation of a new VM (for instance, with $n$=50 and 1,000 clients we create 20 VMs). We then measure the cumulative throughput at the CHANGE platform and plot the results in figure 2.11. Even with all VMs pinned to the same CPU core, ClickOS gives 10Gbps line rate throughput in this experiment, highlighting its scalability to large numbers of clients.

Is 1,000 clients, however, a realistic target? To understand whether CHANGE can scale to real-world workloads, we downloaded and processed MAWI traces taken between the 13th and 17th of January; these are packet traces from the WIDE backbone in Japan. Each trace covers 15 minutes of traffic, and we eliminate all connections for which we do not see the setup and teardown messages. Most of the traffic we ignore is background radiation (instances of port scanning), but some of it is due to longer connections intersecting the 15-minute trace period. The results show that, at any moment, there are at most 1,600 to 4,000 active TCP connections, and between 400 to 840 active TCP clients (i.e., active openers). The exact thresholds varies with the day of the week, but the main take-away is that a single CHANGE platform running on commodity hardware could run personalized firewalls for all active sources.

### 2.5.2 Controller Scalability

We turn our attention to the CHANGE controller: how long does it take to respond to a user's processing request? We ran an experiment where the client was connected to a WiFi hotspot, and the WiFi hop dominated the round-trip time to the controller (around 10ms). The client issued the request shown in our push example (figure 2.2). The provider runs the static checking alone, without instantiating the VM, and we measure the wall-clock request execution time at the client. For the provider topology shown in figure 2.1, the whole

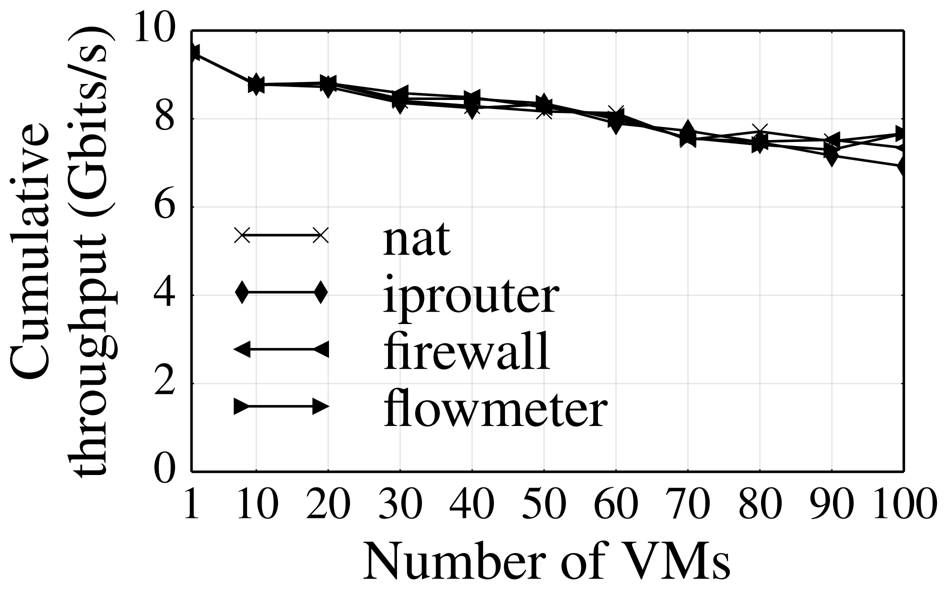


Figure 2.14: CHANGE platforms run many middleboxes on a single core with high aggregate throughput.

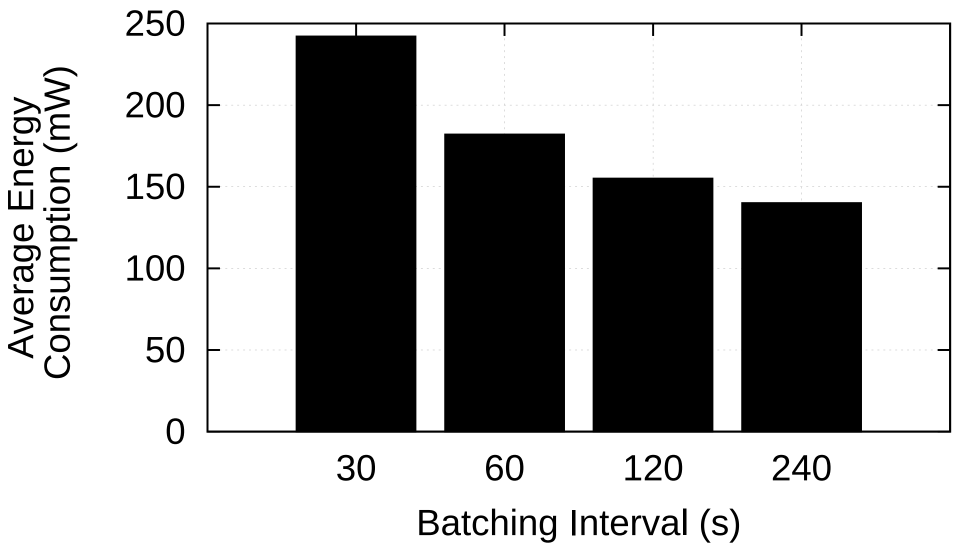


Figure 2.15: Mobile phones save energy when a CHANGE platform batches push traffic into larger intervals.

request takes 250ms; out of this, the server needs 101ms to compile the Haskell rules our front-end generates, and just 5ms to run the analysis itself. This result implies that static checking could be used for interactive traffic, such as deciding the proper way to tunnel traffic.

To understand how this number scales to a larger, more realistic operator topology we randomly add more routers and platforms to the topology shown and measure the request processing time. The results in Figure 2.12 show that static processing scales linearly with the size of the network. The biggest contributor to the runtime is the time needed to compile the Haskell configuration. In practice, the operator will have a compiled version of its configuration, and we only need to compile the client's configuration and load it dynamically. In short, SymNet scales well: checking reachability on a network with 1,000 boxes takes 1.3 seconds.

**Stateful checking.** SymNet models basic TCP functionality and can model middleboxes that hold per-flow state. To model TCP connectivity we run reachability between the two endpoints as follows: the active opener injects a symbolic packet whose protocol is set to TCP, and this is traced all the way to the destination (passive opener). At the destination, the same symbolic packet is echoed back by reversing the source and destination addresses and ports, but keeping all the variable bindings.

Next, to model stateful middleboxes—such as a stateful firewall—we attach the state of the box to the symbolic packet. In particular, the firewall adds a state variable to the symbolic packet on its way out, and only allows return packets through that carry this variable. The time needed for checking is the same as in figure 2.12. Checking stateful middleboxes is really cheap with SymNet: the memory costs grow linearly with the number of stateful boxes encountered; there is no state explosion as is the case with model-checking based techniques (e.g., [17]).

**Sandboxing.** We use a single ClickOS VM to receive traffic directly via a 10 Gbps NIC or through our `ChangeEnforcer` sandboxing element (recall section 2.2.4). Figure 2.13 shows that the throughput drops by a third for 64B packets, and by a fifth for 128B packets. For other packet sizes there is no measurable degradation. We conclude that sandboxing is relatively expensive for small packet sizes; luckily, it is not needed in the first place if we can statically check whether the processing is safe. We have seen in our previous results that static checking is very fast—in particular, security checking takes a few hundred milliseconds, out of which Haskell compilation dominates.

Another question is how many false positives appear in practice, which will force us to resort to sandboxing. To answer this question we ran security checks on processing modules implementing the middlebox functionalities shown in Table 2.1 and found that static analysis accurately characterizes safety in these cases. Of course, more complex processing may yield more false positives—more experience is needed to understand how prevalent false positives are.

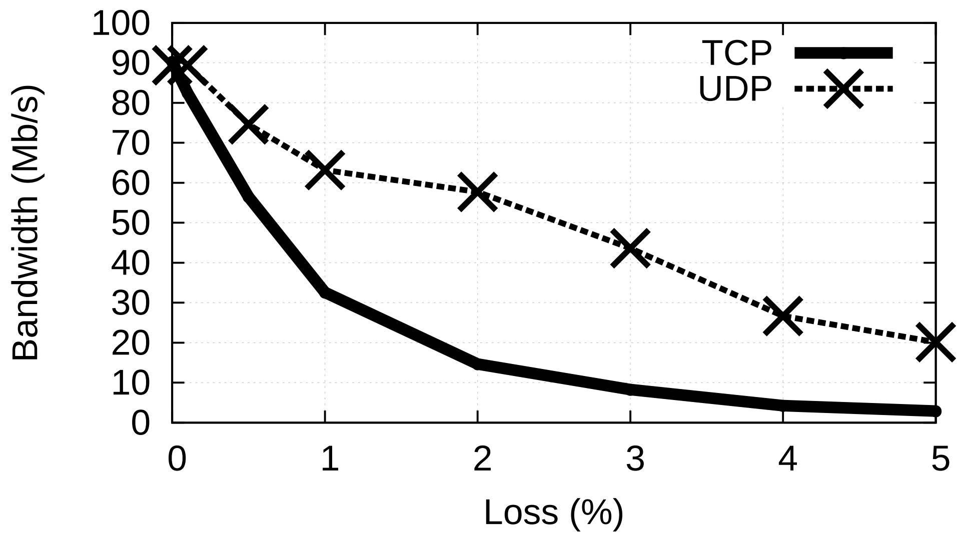


Figure 2.16: SCTP performance when tunneling over TCP and UDP, with different measured loss rates.

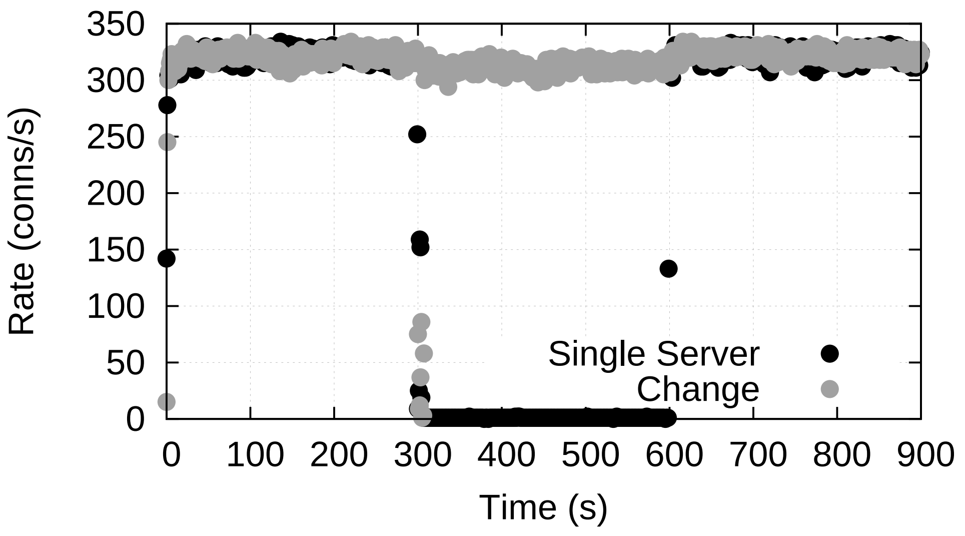


Figure 2.17: Defending against a Slowloris attack with CHANGE

## 2.6 Use-cases

We now discuss a variety of use-cases showcasing the benefits our architecture for the main actors of the Internet ecosystem: operators, end-users and content-providers.

**Software Middleboxes** are a great lure for operators and have become prominent with the advent of NFV. CHANGE offers a scalable way to run many middleboxes on low-end commodity hardware. In our first use case, we deploy a number of different middleboxes on a CHANGE platform and measure the aggregate throughput of the box. Traffic is generated by a client running *curl* connected via a 10 Gigabit pipe to the platform, itself connected via another 10 Gigabit pipe to a server running `nginx` and serving content from a ramdisk. We vary the number of middleboxes (one per VM) that we run on a single core of our server, split the client traffic evenly between the middleboxes and plot the aggregate throughput through the platform. Figure 2.14 shows that the CHANGE platform manages to sustain high throughput, regardless of the number of middleboxes and their type; this goes beyond the work in [18], which measured middlebox throughput

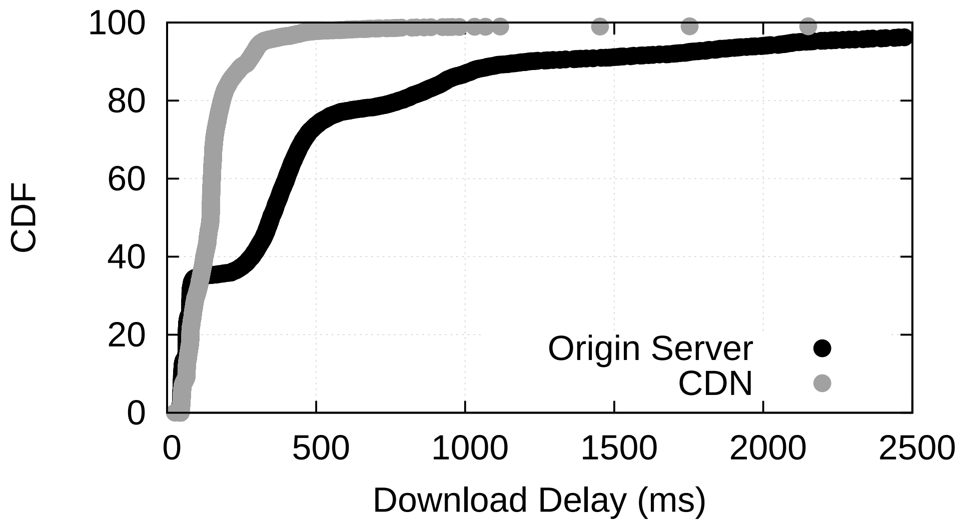


Figure 2.18: 75 PlanetLab clients downloading a 1KB file from the origin server or a 3-server CHANGE CDN.

using a single VM.

**Push Notifications.** Using CHANGE , mobiles can request their cellular operator to forward a port in the NAT to them, perhaps subject to some security clearance (e.g., only allowing notifications from a certain source address). Additionally, the clients' processing can batch messages at the provider, thus reducing the number of costly "energy tails"—a nickname for the many seconds cellular radios wait before switching back to the idle state. We instantiated the configuration presented in figure 2.2 using as client a Samsung Galaxy Nexus mobile phone connected via 3G to the Internet. It takes around 3s for the whole request to be executed by the CHANGE controller, which finds the proper placement for the processing module, and checks its security, the clients' requirements and the operator's. This time is dominated by the time needed to wake up the 3G interface. The reply specifies the IP address of the newly allocated processing module. From one of our servers, we send one UDP message with 1KB payload every 30s to the given address and port; the platform batches the messages in the processing module and delivers them at different intervals.
We measured the mobile device's energy consumption with the Monsoon power monitor, and show the results in figure 2.15. Batching has a massive effect on average energy consumption, reducing it from 240mW to 140mW. In this use-case, CHANGE brings benefits for both the client and the cellular operator: the client can trade increased delay in receiving notifications for lower energy consumption, while the operator gets the opportunity to monitor and perhaps filter malicious messages.

**Protocol Tunneling.** Consider the task of running SCTP (or any other new protocol) over the Internet. Deploying it natively is impossible because middleboxes block all traffic that is not TCP or UDP. Thus SCTP must be tunneled, but which tunnel should we use? UDP is the best choice, but it may not work because of firewalls that just drop non-DNS UDP packets. In such cases, TCP should be used, but we expect poorer performance because of bad interactions between SCTP's congestion control loop and TCP's.
To understand the effect of such interactions, we use iperf to measure bandwidth between two servers con-

nected via an emulated wide-area link with capacity 100Mbps and a 20ms RTT. We also introduce random losses to understand how the protocol fares under congestion scenarios. The results in figure 2.16 show how SCTP over TCP encapsulation dramatically reduces the throughput achieved: at 3% loss rate, TCP tunneling gives 5 times less throughput than UDP.

SCTP has to be adaptive about the tunnel it uses: first try UDP and fall back to TCP if UDP does not work, but to make the decision we need at least one timeout to elapse at the sender—three seconds according to the spec. Instead, the sender could use the CHANGE API to send a UDP reachability requirement to the network. This request takes around 200ms, after which the client can make the optimal tunnel choice, drastically reducing connection setup time.

**HTTP vs. HTTPS.** Mobile apps heavily rely on HTTP to communicate to their servers because it just works, and in many cases they are even tunneling other traffic over HTTP. Application optimizers deployed by the network may alter HTTP headers, breaking the application's own protocol. Should the applications use HTTPS instead to bypass such optimizers? We have measured the energy consumption of a Samsung Galaxy Nexus phone while downloading a file over WiFi at 8Mbps. The download times are almost identical, while the energy consumption over HTTP was 570mW and 650mW over HTTPS, 15% higher. The added cost of HTTPS comes from the CPU cycles needed to decrypt the traffic.

Smaller energy consumption is a strong incentive for mobiles to use HTTP, but this may break apps, so we are stuck with the suboptimal solution of using HTTPS. Instead, the client should use CHANGE to send an invariant request to the operator asking that its TCP payload not be modified. The operator has incentives to comply with the invariant, because seeing plaintext traffic improves its network's security.

The provider should turn its HTTP optimizations off for requesting apps, resulting in a win-win situation for both parties. In our scenario, the operator simply instantiates a processing module on Platform 2, bypassing the optimizer for the target client's traffic.

**TLS Offloading.** A small website wants to enable HTTPS to increase security, but this imposes a burden on its server's CPU and increases web page load times because of the extra RTTs required by the TLS handshake. We deploy a client running `wget` and a server running `apache` on different continents, with a round-trip time between them of around 100ms. With HTTPS, the average download time for 10 requests is 960ms. We next use CHANGE to deploy a stock reverse-proxy processing module close to the client (10ms RTT) and to redirect the client to use this proxy via DNS. As a result, the client talks HTTPS to the processing module, and the module communicates over HTTP with the server. With this setup, the average page download time goes down to 640ms. The decrease is not as big as one might expect because the CHANGE reverse proxy is not optimized at all: it opens a new HTTP request to the web server for each connection, which adds almost 100ms to the request.

**DoS Protection.** We use the same stock processing module to defend against an HTTP attack tool called

Slowloris that attempts to starve all valid clients of a server by maintaining as many open connections to that server as possible. Slowloris opens many connections and trickles the HTTP request bytes at a very slow rate, which prevents the server from timing out the connection. The best known defense is to just ramp up the number of servers, and this is what we do with CHANGE . When under attack, the web server starts a number of processing modules at remote operators, and redirects new connections via geolocation to those servers. In figure 2.17 we plot the number of valid requests serviced per second before, during, and after the attack. We can see that CHANGE is able to quickly instantiate processing and divert traffic, thus reducing the load on the origin server.

**Content Distribution Network.** CHANGE can safely run legacy code in sandboxed processing modules. As our final use case we run a small-scale content-distribution network as follows. The origin server is located in Italy, and there are three content caches (located in Romania, Germany and Italy) instantiated on CHANGE platforms. Each content cache is an x86 virtual machine running Linux and Squid, the latter configured as a reverse-proxy. The CHANGE controller instantiates sandboxing for such machines using the mechanism previously described in section 2.2.4.

Traffic is redirected to the caches via DNS. CHANGE also offers as a processing module a simple DNS server that we have coded which takes a list of platform IP addresses and services queries using geolocation information [4], directing each client to its closest cache. As clients we used 75 PlanetLab nodes scattered around Europe – geolocation spreads these clients to the caches approximately evenly. We ran repeated downloads of 1KB files from all these clients, and we report the results in figure 2.18. We see that the CDN is behaving as expected: the median download time is halved, and the 90% percentile is four times lower.

[4]We use the MaxMind geolocation database [4].

# 3 Online Resource Allocation

In the CHANGE vision Flowstream platforms must be able to host processing requested by agents. These requests for processing come in the form of allocation requests for network services. In a previous deliverable ([1]) we gave algorithms to allocate a given set of network services on a Flowstream platform such as to satisfy some performance criteria. In this deliverable we explain how these methods can be adapted to the online problem, where the allocation demands from new services arrive over time. Experimental results for this online problem are provided. In the following, we first recall the different problem components and then differentiate between the online and the offline problem. Next, the adaptation of the offline constraint satisfaction procedure to the online problem is described. Some parts of the constraint satisfaction procedure that are new w.r.t Deliverable 3.2 ([1]) are then highlighted. Finally we test different possible configurations of the proposed procedure on a number of simulated online scenarios.

## 3.1 Problem Description

As for the offline problem, the online problem has two main components: a Flowstream platform and a (previously unknown in the case of the online problem) set of network services .

The Flowstream platform is composed of programmable switches and commodity servers. The switches interconnect the servers. These servers are characterized by their number of CPU cores, their memory capacity and their internal communication bandwidth. The servers are located in racks. Each rack is associated with a Top-of-Rack (ToR) switch, and all communication between servers (whatever their rack) goes through their corresponding ToR(s). The Flowstream platform is connected to the Internet by a given number of I/O-nodes. The different nodes (switches, servers and I/O nodes) in the Flowstream platform are connected via links. With each link is associated a bandwidth and a delay. The Flowstream platform can thus be seen as a directed graph.

A network service (NS) is a service that must be processed by the Flowstream platform. Each NS is represented by a directed acylic graph (DAG) containing three types of nodes: source nodes, sink nodes and Processing Modules (PM). The source and sink nodes represent the entry from and the exit to the outside world. The processing modules correspond to a computation that needs to be carried out. Each module requires a given number of CPU cores and a given amount of memory. Some of the modules need to communicate, this is represented by a link in the DAG. Since the source (sink) nodes are the entry (exit) point from (to) the Internet, there are no incoming (outgoing) links for these nodes in the DAG. All links in the DAG are associated with a bandwidth requirement. Furthermore, for each network service, a bound is imposed on the total delay between the reception of data at a source node and the reception of the processed data at a sink node.

In the offline problem, a set of NSs must be deployed on the Flowstream platform. This means that for each NS, its PMs must be attributed to servers and its sink nodes to I/O nodes (source nodes are fixed to one I/O node known beforehand). As a server only has a limited number of CPU nodes, limited memory capacity and a limited internal communication bandwidth, it can only handle a restricted set of PMs at once. The server's resources may not be exceeded by the PMs that are allocated to it.
Furthermore communication paths between the PMs and between the PMs and the source and sink nodes must be computed. These communication paths are paths in the graph representing the Flowstream platform. Since the links in this graph are associated with a limited bandwidth, care has to be taken to guarantee that this bandwidth limit is not exceeded on any link. Also the total delay on the paths must allow to respect the bound on the total delay of the NSs.
A complete model for the offline problem, along with detailed constraints and objectives was given in deliverable[1].

The online problem is the same as the offline problem, with the difference that the set of NSs is not known beforehand. The NSs to distribute over the Flowstream platform appear gradually, and servers, I/O nodes and communication paths need to be chosen for the new NSs while taking into account the resources already allocated to the NSs currently in execution. At no point is any information about the future known. The goal of the online problem is to accommodate as many of the incoming NSs as possible while respecting the problem constraints.

## 3.2 Notations

In this section we quickly review some notations that will be used in the remainder of this chapter.
A Flowstream platform is modeled as a directed graph $\mathcal{D} = (F, E)$, The nodes in $F$ correspond to different types of switches (I/O nodes, Openflow switches) or servers.
The set of I/O nodes is denoted by $F_{\text{I/O}}$ ($F_{\text{I/O}} \subset F$) and the set of servers by $F_{\text{SERVER}}$ ($F_{\text{SERVER}} \subset F$).
A network service is modeled as a directed acyclic graph $\mathcal{S} = (G, H)$. The nodes in $G$ are partitioned into three sets: the set of source nodes $G_{\text{SOURCE}}$, the set of sink nodes $G_{\text{SINK}}$ and the processing modules $G_{\text{PM}}$. We have thus $G = G_{\text{SOURCE}} \cup G_{\text{SINK}} \cup G_{\text{PM}}$.
A solution $sol$ assigns each node from each NS graph to a node in the Flowstream platform graph $\mathcal{D}$. Given a solution $sol$, we denote $v(sol)$ the total number of violations of all the constraints of the problem, and $h(sol)$ the number of servers used by solution $sol$. Solution $sol$ can be modified by (re-)assigning some node $g \in G$ to some node $f \in F$, the resulting solution is denoted by $sol[g, f]$.

## 3.3 Adaptation to the Online Problem

In the offline problem, the set of network services that need to be handled by the Flowstream platform is known in advance. In the online version of this problem, the networks services appear over time, and for each

NS, servers must be allocated to its PMs, I/O nodes to its sink nodes and communication paths need to be determined, all of this given the current allocations of servers and I/O nodes to already running NSs.

The idea is to reuse the approach presented in [1]. The pseudo-code for the online scenario is given in Algorithm 1. At initialization an empty solution $sol$ is created and the communication paths to be used on the Flowstream platform are precomputed. This is done to avoid computational overhead during the constraint satisfaction procedure and is described in detail in section 3.3.1. The system then listens for new incoming network services. Each time a new service $\mathcal{S}$ is received the constraint satisfaction procedure will try to update the current solution $sol$ w.r.t. the new network service. If the constraint satisfaction procedure is able to find a mapping between the nodes in $\mathcal{S}$ and the nodes in $\mathcal{D}$ such that all constraints are respected, service $\mathcal{S}$ is accepted and the current solution $sol$ is updated. In the other case, the network service $\mathcal{S}$ is rejected by the system. The constraint satisfaction procedure is described in section 3.3.2. Note that it is not allowed to modify the network services already in execution on the Flowstream platform.

**Algorithm 1:** OnlineScenario

```
1 $sol \leftarrow \emptyset$;
2 $P \leftarrow$ precompute communication paths;
3 while $true$ do
4     $\mathcal{S} \leftarrow$ get incoming network service;
5     $sol' \leftarrow$ ConstraintSatisfactionProcedure($sol$,$P$,$\mathcal{S}$,$\mathcal{D}$);
6     if $sol' \neq \emptyset$ then
7         $sol \leftarrow sol'$;
8     else
9         Reject $\mathcal{S}$;
```

### 3.3.1 Computing communication paths

In order to avoid the computation overhead, the communication paths between servers and between servers and I/O nodes of the Flowstream platform are precomputed. This is done in such a way that the number of paths that pass a certain link are balanced. That means we try to avoid situations where a link is shared by many paths while some other link is used by only a few or no paths. The algorithm for computing paths is depicted in Algorithm 2. It starts by computing $F_p$, the set of node pairs between which a path needs to be computed (line 1). Then a cost associated with each link in graph $\mathcal{D}$ is initialized to 1 (lines 2 – 3). Next, node pairs are randomly selected from $F_p$ (line 4-5). For each node pair, the shortest path in graph $\mathcal{D}$, taking into account the cost $c(l)$ of each link $l \in E$ is computed (line 6). The cost of the links used by this path is then incremented. The rationale behind this is, that this will deter from the use of these links in the computation of the shortest path for a different pair of nodes (lines 8–9). This procedure is repeated for every pair of nodes in $F_p$.

### 3.3.2 Online constraint satisfaction procedure

The constraint satisfaction procedure consists of at most two steps:

**Algorithm 2:** ComputePaths($\mathcal{D}$)

```
Input: Flowstream platform $\mathcal{D} = (F, E)$
Output: $P$ the set of paths between nodes in set $F_{\text{SERVER}} \cup F_{\text{I/O}}$
1 $F_p = (F_{\text{SERVER}} \times F_{\text{SERVER}}) \cup (F_{\text{I/O}} \times F_{\text{SERVER}}) \cup (F_{\text{SERVER}} \times F_{\text{I/O}})$;
2 foreach $l \in E$ do
3     $c(l) \leftarrow 1$;
4 while $F_p \neq \emptyset$ do
5     $\langle v_1, v_2 \rangle \leftarrow$ Pop randomly from $F_p$;
6     $p(v_1, v_2) \leftarrow$ shortest path from $v_1$ to $v_2$;
7     $P \leftarrow P \cup p(v_1, v_2)$;
8     foreach $l \in p(v_1, v_2)$ do
9         $c(l) \leftarrow c(l) + 1$;
```

1. The computation of an initial solution (possibly not respecting all constraints)

2. A Tabu Search to render the initial solution (from step 1) feasible (if necessary)

In the online mode these steps are executed for each new incoming service. The first step consists in modifying the current solution $sol$ (which assigns elements from the Flowstream platform to the network services that are currently in execution), in order to accommodate the elements of the new network service on the Flowstream platform. The resulting, modified, solution $sol'$ is possibly infeasible (does not respect all the problem constraints), in that case the Tabu Search can be executed on it. The goal of the Tabu Search is to further modify solution $sol'$ in order to render it feasible. If this fails, the original solution $sol$ is kept (and the network service is rejected, see Algorithm 1). In the experiments we considered both using only Step 1 (the computation of an initial solution) and Steps 1 & 2 (computation of initial solution on which Tabu Search is applied).

The Tabu Search has been fully described in [1]. The only modification that has to be made for the online problem, is that the search may only modify the allocation of the new network service's elements. This means that the assignment of already running network services' elements to elements of the Flowstream platform must remain untouched.

In [1] we described a round-robin based approach to compute an initial solution for the offline problem. For the online problem we consider two additional methods. The three methods are described in the following section.

**Computing an initial solution**

The job of the algorithms presented in this section is to take a current solution $sol$, a new network service $\mathcal{S} = (G, H)$ and to assign the network service's elements to the Flowstream platform $\mathcal{D} = (F, E)$ (while taking into account the network services already running on the platform). The source and sink nodes of the network service are accommodated randomly on their optional locations. The proposed algorithms thus only need to map nodes $G_{\text{PM}}$ to nodes $F_{\text{SERVER}}$. Again the mapping of already running network services may not be modified.

We consider three algorithms to achieve this: Random Initialization, Minimization of #Servers Initialization and Round Robin Initialization.

*Random Initialization*

The algorithm is depicted in Algorithm 3. For each processing module $g_i \in G_{\text{PM}}$ we first try to find a server node $j \in F_{\text{SERVER}}$, such that mapping $g_i$ to $j$ does not result in (possibly further) violations of the problem constraints (line 2). If several such servers are available, one of them is chosen randomly. If no such server is available (line 3), some server is chosen at random (line 4). Finally the current solution $sol$ is updated, assigning processing module $g_i$ to server $j$. Note that this solution may be infeasible.

**Algorithm 3:** RandomInit($sol, P, \mathcal{S}, \mathcal{D}$)

```
Input: Current solution $sol$, Flowstream platform $\mathcal{D} = (F, E)$, incoming network service $\mathcal{S} = (G, H)$
Output: Possibly modified solution $sol$
1 foreach processing module $g_i \in G_{PM}$ do
2     $j \leftarrow$ select $j \in F_{\text{SERVER}}$ such that $v(sol[g_i, j]) - v(sol) = 0$;
3     if $j = \emptyset$ then
4         $j \leftarrow$ select randomly from $F_{\text{SERVER}}$;
5     $sol \leftarrow sol[g_i, j]$;
```

*Minimization of #Servers Initialization*

The second initialization algorithm is depicted in Algorithm 4. It is similar to the random initialization except that it first tries to select a server $j$ for the current processing module $g_i$, such that the assignment of $g_i$ to $j$ does not result in (possibly further) violations of the problem constraints and such that the number of servers used is minimal (line 2). If no such server can be found, we use the server that causes the least constraint violations when $g_i$ is assigned to it. Note that the resulting solution may be infeasible.

**Algorithm 4:** MinimumServersUsedInit($sol, \mathcal{S}, \mathcal{D}$)

```
Input: Current solution $sol$, Flowstream platform $\mathcal{D} = (F, E)$, incoming network service $\mathcal{S} = (G, H)$
Output: Possibly modified solution $sol$
1 foreach processing module $g_i$ of $G_{PM}$ do
2     $j \leftarrow$ select $j \in F_{\text{SERVER}}$ such that $v(sol[g_i, j]) - v(sol) = 0$ and $h(sol[g_i, j])$ is minimal;
3     if $j = \emptyset$ then
4         $j \leftarrow$ select from $F_{\text{SERVER}}$ such that $v(sol[g_i, j])$ is minimal;
5     $sol \leftarrow sol[g_i, j]$;
```

*Round Robin Initialization*

The third algorithm is based on a Round Robin scan. It was already presented in [1]. The idea behind this algorithm is to favor the placement of processing modules of a same network service on one or more servers connected to a same ToR. This allows to ease the respect of the limit on the total delay between reception of data at a source node and the reception of the processed data at a sink node. The algorithm assumes nodes corresponding to servers located on a same rack (connected to a ToR) are numbered in a consecutive way.

For completeness's sake the complete algorithm is depicted in Algorithm 5.

```
Algorithm 5: RoundRobinInit($sol, \mathcal{S}, \mathcal{D}$)
Input: Current solution $sol$, Flowstream platform $\mathcal{D} = (F, E)$, incoming network service $\mathcal{S} = (G, H)$
Output: Possibly modified solution $sol$
1  $j_0 \leftarrow currentServer$;
2  foreach processing module $g_i \in G_{PM}$ do
3      $j \leftarrow j_0$;
4      if $v(sol[g_i, j])) - v(sol) > 0$ then
5          $j \leftarrow j + 1$;
6          if $j > nbServers$ then
7              $j \leftarrow 1$;
8          $found \leftarrow$ FALSE;
9          while $\neg found$ do
10             if $j = currentServer$ then
11                 break;
12             if $v(sol[g_i, j])) - v(sol) = 0$ then
13                 $found \leftarrow$ TRUE;
14             else
15                 $j \leftarrow j + 1$;
16                 if $j > nbServers$ then
17                     $j \leftarrow 1$;
18         if $\neg found$ then
19             $j \leftarrow$ select from $F_{\text{SERVER}}$ such that $v(sol[g_i, j])$ is minimal;
20         $sol \leftarrow sol[g_i, j]$;
21         $j_0 \leftarrow j$;
```

## 3.4 Simulation experiments

We implemented our constraint satisfaction procedure in the Comet language and it is available to all project partners in the CHANGE SCM repository. We tested different configurations of the constraint satisfaction procedure on an online scenario under different constraints. The different configurations are obtained by using algorithms 3, 4 and 5 individually and in combination with the Tabu Search. The configurations are summarized in Table 3.1.

| Name | Description |
|---|---|
| R | Random Initialization (Algorithm 3) |
| R+TS | Tabu Search with R procedure for the initial solution |
| RR | Round Robin Initialization (Algorithm 5) |
| RR+TS | Tabu Search with RR procedure for the initial solution |
| S | Minimization of #Servers Initialization (Algorithm 4) |
| S+TS | Tabu Search with S procedure for the initial solution |

Table 3.1: Description of constraint satisfaction procedure configurations

### 3.4.1 Experimental setup

For the experimental setup we need to simulate both a Flowstream platform and a sequence of incoming network services.

**Simulating a Flowstream platform** To test our algorithms we view a Flowstream platform as a large data center including 32 core switches, 32 intermediate switches, 500 ToR switches and 10000 servers. Each core switch is furthermore connected to 5 I/O switches. Each server has 8 CPU cores and 2048MB of memory. The internal bandwidth of servers is assumed to be 10Gbps. The links connecting servers to their ToR have a bandwidth of 1Gbps. Each remaining link of the data center has a bandwidth of 10Gbps. The delay on each link is assumed to take value 1.

**Simulating a sequence of network services** We consider four different types of network service templates. The properties for each are given in Table 3.2, their topologies are illustrated in Figure 3.1. We assume that the size of packets transmitted between network service elements is of 1024 bits.
Using the properties from Table 3.2 we generated $k = 4000$ network service instances for each of the four types. Thus, in total, we have 16000 network services. A random sequence is then generated from these 16000 network services. Based on this sequence each network service is assigned a sequence number. To simulate the online character of the problem, these network services are then assumed to appear over time, respecting the sequence. We assume that all the network services remain in execution until the last network service in the sequence has arrived. At no point is any information about the upcoming network services available.

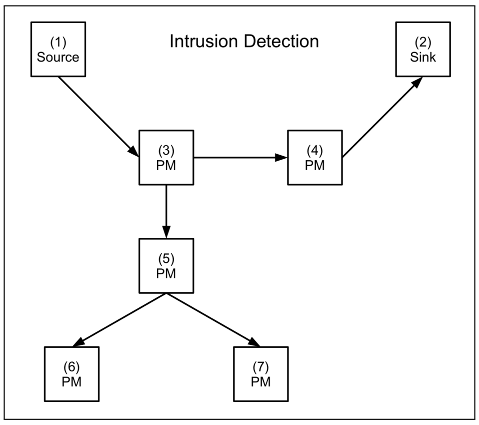
| Name | #I/O | #PM | CPU cores | Memory (MB) | Traffic (pps) |
|---|---|---|---|---|---|
| Intrusion Detection | 2 | 5 | $\{0.5, 2\}$ | 100..1024 | 38800..100000 |
| Load Balancing | 5 | 7 | $\{0.7, 1, 1.5\}$ | 100..200 | 25000..100000 |
| Network Access Control | 2 | 3 | $\{0.5, 1\}$ | 100..200 | 96000..100000 |
| Tunneling | 3 | 3 | $\{1.5, 2.5\}$ | 200..1024 | 50000..100000 |

Table 3.2: Description of network service samples

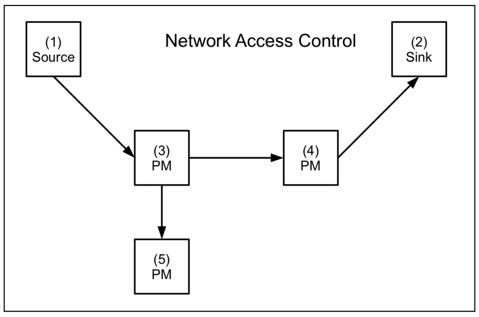
### 3.4.2 Experimental results

For our experiments we always keep the same sequence of network services but vary the delay bound for which we try values $\{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20\}$. We thus obtain 11 instances of the online problem, on which we test each of the algorithm configurations from Table 3.1.

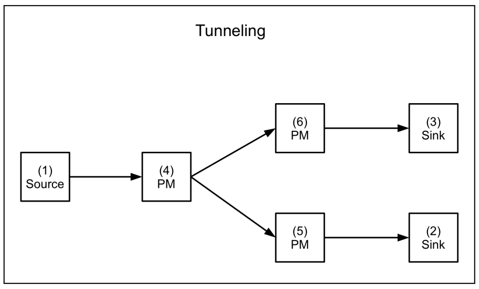
The constraint satisfaction procedure configurations are compared in terms of two criteria: the sequence number of the first network service to be rejected and the total number of accepted network services.



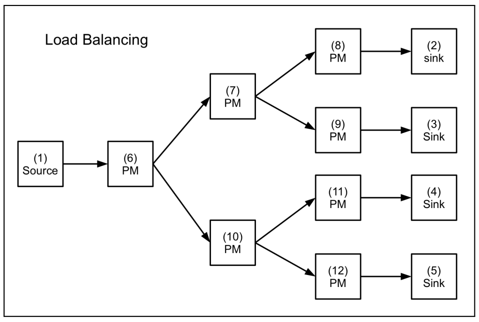
(a)



(b)



(c)



(d)

Figure 3.1: Network service templates

Table 3.3 indicates the sequence number of the first network service to be rejected by the six algorithm configurations under different values for the delay bound. Note that when a network service is rejected, this does not mean that the following network services will be rejected as well. We see that when the delay bound value is less than 16, the configuration RR+TS is the best one: with a delay bound of 11, RR+TS is able to allocate the 1193 first network services, while configuration RR can allocate only the first 326 incoming network services. The remaining configurations already fail to allocate the second incoming network service. With a delay bound value greater than 15, the configuration S+TS performs best.

| Delay bound | R | R+TS | RR | RR+TS | S | S+TS |
|---|---|---|---|---|---|---|
| 10 | 2 | 2 | 2 | 2 | 2 | 2 |
| 11 | 2 | 2 | 327 | 1194 | 2 | 2 |
| 12 | 2 | 2 | 429 | 2938 | 4 | 2 |
| 13 | 2 | 8 | 429 | 10157 | 2 | 172 |
| 14 | 2 | 8 | 429 | 10157 | 8 | 113 |
| 15 | 2170 | 5081 | 10134 | 10134 | 58 | 991 |
| 16 | 10232 | 10297 | 10134 | 10134 | 8929 | 11650 |
| 17 | 10301 | 10279 | 10134 | 10134 | 10186 | 11652 |
| 18 | 10265 | 10303 | 10134 | 10134 | 11650 | 11645 |
| 19 | 10253 | 10233 | 10134 | 10134 | 11639 | 11647 |
| 20 | 10264 | 10253 | 10134 | 10134 | 11637 | 11635 |

Table 3.3: Sequence number of first rejected network service

Table 3.4 presents the total number of network services accepted and allocated on the Flowstream platform. The results are mitigated: none of the configurations outperforms the remaining configurations over all values for the delay bound. All of the configurations are able to allocate at least 9950 network services. Configuration R finds better results than the others in 3 out of the 11 instances while R+TS finds better results than the others in 4 instances. Each of the configurations RR+TS and S+TS finds better results than the others in 1 out of 11 instances, and configuration RR finds better results than the others in 2 out of 11 instances. Another observation that we make is that the initialization procedures (R, S and RR) do not always perform worse than their combination with Tabu Search (R+TS, S+TS, RR+TS) in terms of total number of accepted network services. This may be due to the fact, that the initial solution procedures reject a network service more easily than the combined procedures. It is possible that rejecting a network service allows to allocate a number of the following network services, which would not have been possible otherwise, if the first network service would have been allocated. It is for this reason as well, that a given configuration will not always be able to allocate a higher total number of network services when the delay bound increases. The same behavior (in lesser magnitude) can be observed in terms of the first rejected network service, here the variations are most likely due to random decisions in the constraint satisfaction procedure.

| Delay bound | R | R+TS | RR | RR+TS | S | S+TS |
|---|---|---|---|---|---|---|
| 10 | 9950 | 9986 | 10775 | 10918 | 10942 | 11141 |
| 11 | 9957 | 9975 | 11894 | 12500 | 11142 | 11659 |
| 12 | 9987 | 9988 | 12606 | 12491 | 11146 | 11708 |
| 13 | 10009 | 12460 | 12627 | 12510 | 11200 | 11882 |
| 14 | 10039 | 12657 | 12622 | 12510 | 11215 | 11879 |
| 15 | 11909 | 13202 | 12623 | 12519 | 11505 | 11709 |
| 16 | 13131 | 13167 | 12506 | 12505 | 11640 | 11649 |
| 17 | 13166 | 13156 | 12506 | 12506 | 11648 | 11651 |
| 18 | 13152 | 13159 | 12505 | 12505 | 11652 | 11644 |
| 19 | 13146 | 13138 | 12505 | 12505 | 11646 | 11647 |
| 20 | 13151 | 13146 | 12506 | 12506 | 11637 | 11634 |

Table 3.4: Total number of accepted network services

## 3.5 Conclusion

This chapter describes how the constraint satisfaction procedure to allocate a set of network services on a Flowstream platform can be adapted to the online problem. In this problem a sequence of network services appear over time, and the system has to decide whether they can feasibly handled, given the current load of the platform. Our constraint satisfaction procedure consists of two steps: the generation of an initial solution and the improvement of this solution should it be infeasible. For the initial solution generation we propose three different strategies. We tested different configurations of our constraint satisfaction procedure on an online problem under varying delay bounds. The results show that none of the configurations clearly outperforms the others in terms of total number of accepted network services. However two of the configurations perform better than the others in terms of the delaying the first rejection of a network service. Future work focuses on other search heuristics exploiting the dynamic path computation during the allocation of network services: communication paths between processing modules are neither precomputed nor fixed.

# 4 Updates to the inter-platform signaling framework

## 4.1 Consolidated inter-platform signaling framework software

The inter-platform signaling framework introduced in deliverable [2] (high level architecture) and in deliverable [3] (software design) has proven to be solid during the integration and testing phases (I&T). Still the design fulfills the initial requirements identified for the CHANGE flow processing architecture, and the overall signaling architecture from [2] is reflected into the inter-platform signaling prototype. Consequently, the work described in this deliverable primarily focuses on the improvements and addition of functionalities on the software.

The inter-platform signaling prototype has been released as a GPL2 licensed package, and is accessible to all the project partners via the CHANGE SCM repository. It passes the unit tests contained into its automated harness test suite[1][2] as well as the in-house tests which have been executed under varying conditions in multi-platform simulated and virtualized environments. The description of tests is contained in the software package and is not included in this document as it does not add a real value to the description of the actual improvements implemented in the prototype. I&T activities are in progress in the WP5 scope on the CHANGE testbeds, and may originate further refinements or fixes to the inter-platform signalling software. These final refinements will be surely included on the final software releases, and detailed in WP5 reports.

The updates to the signaling framework software can be briefly summarized in:

1. improvements to the the package building infrastructure which have been applied in order to build the software components in less prerequisites demanding Linux/OS based systems;
2. integration of the interplatform signaling software within the CHANGE platform;
3. improvements on the development framework (i.e. the common libraries, shared among all the signaling SW components) to enhance the support to the newly introduced functionalities;
4. implementation of the interplatforms tunnels provisioning.

The activities listed under 1) – 3) are mostly software refinements that do not change the framework design and major functionalities. On the contrary, the tunnel provisioning function represents an important add-on

[1]A *test harness* is a collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs. It has two main parts: the test execution engine and the test script repository. Test harnesses allow for automation testing: they can call functions with supplied parameters, print out the outputs and finally compare the obtained results to the expected values.

[2]The signaling package *test execution engine* is implemented with ad-hoc code embedded into each software module while the *test script repository* is represented by the package *src/test* directory. This repository provides test suites for the Signaling Manager, the Service Manager and the Service Broker. The package building framework provides an automation testing framework that can be used to test each component as well as their shared libraries.

to the prototype and therefore it will be described in more details in the following.

### 4.1.1 The inter-platform tunnels provisioning

The lifetime of an inter-platform tunnel can be partitioned into three sequential phases: a) the tunnel setup, b) its (eventual) modification and c) the final teardown.

The information exchanged among the signaling functional entities varies depending on the specific phase under consideration.

During the setup phase, the provisioning of a tunnel between two signaling-adjacent flow processing platforms can be implemented by the means of two separate FPRO actions, one for each platform/tunnel endpoint (TEP). Each FPRO action must contain:

- the platform local tunnel identifier (TID)
- the platform local information required for the tunnel instantiation (Tunnel End-Point Data, TEPD)
- a peer scoped FPRO-to-FPRO binding required for the correlation of the FPRO tunnel related actions contained into the FPR (Tunnel Pair Identifier, TPID).

Therefore, the FPR can be described as follows:

```
<FPR>         ::= <SID> ... <FPRO-1> ... <FPRO-N> <FPRO-(N+1)>
<FPRO-N>      ::= <PID> ... <FPRO-ACTION> ... [ <TID> <TEPD> <TPID> ]
<FPRO-(N+1)> ::= ... [ <TID> <TEPD> <TPID> ] ...
```

The TEPD contents depend on the tunnel type. By restricting to only the nonencrypted and TLS/SSL encrypted case, the TEPDs can be described as follows:

```
<TEPDupstream> ::=<addrus> [ <dh><ca><scert><skey> ]
<TEPDdownstream> ::= <addrds> [ <ca> <ccert> <ckey> ]
```

Where the addr-us, addr-ds, dh, ca, s-cert/c-cert and s-key/c-key represent the upstream and downstream tunnel addresses, the Diffie Hellman parameters, the Certification-Authority, the Certificate and Key respectively.

Once the setup phase is complete, either the (PID, TPID) pair or the (PID, TID, TPID) tuple can be used to uniquely identify each tunnel endpoint, depending on the scope of the identification: the former, i.e. (PID, TPID), is used between two adjacent platforms; the latter, i.e. (PID, TID, TPID), is used in the service-wide end-to-end scope.

In order to implement the inter-platform tunnel provisioning functionalities, the following incremental changes have been applied to the Signaling Manager and the Service Manager, also including their shared libraries:

- The FPRO serialization/deserialization has been updated to better support the new object for tunnels specification (e.g. TEPD, TPID, TID)
- The tunnels lifetime management functionalities have been introduced, as point solutions, into the Signaling Manager Protocol FSM (see [3])
- The Service Manager CLI has been upgraded with additional tunnel-related commands

The aforementioned changes are further detailed in the following sub-sections and, to better frame them, a Service Manager CLI based example is also presented.

### 4.1.2 Signaling Manager updates

In order to support the tunnels related functionalities, the Tunnel Manager has been introduced into the Signaling Manager software architecture. This single-instance component is in charge of tracking all the resources required by the (platform-local) tunnels management functions as well as hiding their low-level implementation to all the Service FSM[3]. The Tunnel Manager is therefore a software component only interfaced to the low level CHANGE platform tunneling facilities on the one hand, and to the Service FSM on the other hand. While the former interaction can vary depending on the low-level tunnel implementation (e.g. OpenVPN), the latter can be summarized with the following high-level interface description:

- `allocate`: looks for and allocates the resources required for the tunnel provisioning
- `setup_upstream`: implements the tunnel upstream part
- `setup_downstream`: implements the tunnel downstream part
- `teardown`: shutdowns the tunnel either on the upstream or downstream case (it can be assumed the opposite of the `setup` operations)
- `release`: disposes the tunnel resources (it can be assumed the opposite of `allocate` operation)

The Service FSM remains unchanged w.r.t. the description presented in [3]. Only point solutions have been introduced to hookup the aforementioned actions to the FSM transitions, as described in the following:

- `MSG_SSREQ`: The state-transition action (STA) examines the FPRO contents, collects all the TEPs information (e.g. TEPD, TID, TPID) and repeatedly calls the Tunnel Manager allocate method in order to allocate the resources required by each TEP. All the TEPs are handled the same way, regardless of their type (i.e. upstream or downstream)
- `RES_LOCKED`: The STA repeatedly calls the Tunnel Manager `setup_upstream` method for each upstream TEP

[3]A service instantiated into a CHANGE platform, by the means of the signaling framework, is supported by a *Protocol FSM* instance, as described in [3] in section 4.2.1. A Protocol FSM is also referred as a *Service FSM*.

- MSG_SSALLOC: The STA repeatedly calls the Tunnel Manager setup_downstream method for each downstream TEP
- SERVICE_DOWN: The STA calls the Tunnel Manager release method, for each previously provisioned tunnel
- ASSOC_ERR, RES_ERR, SSALLOC_ERR, SSCONF_ERR, ACK_ERR and SSREQ_ERR: The teardown and release methods are called (for each tunnel provisioned for the service the FSM instance is referring to) upon unrecoverable errors detection, in order to torn down all the tunnels instantiated in either the RES_LOCKED or MSG_SSALLOC states.

### 4.1.3 Service Manager updates

The tunnel related commands, introduced into the Service Manager CLI, are the following:

- action-tun-upstream: used to declare the upstream TEPD and TID
- action-tun-downstream: used to declare the downstram TEPD and TID

The two commands have the following syntax:

```
action-tun-upstream <lcl-id> <rmt-id>
                    <intf-id-from> <intf-id-to>
                    [ <dh> <ca> <cert> <key> ]


action-tun-downstream <lcl-id> <rmt-id>
                      <intf-id-from> <intf-id-to>
                      [ <ca> <cert> <key> ]
```

With:

- command: either action-tun-upstream or action-tun-downstream
- lcl-id: a platform local tunnel identifier
- rmt-id: a platform-local tunnel identifier
- intf-id-from: an interface identifier (on the local platform)
- intf-id-to: an interface identifier (on the remote platform)
- dh, ca, cert and key: the Diffie-Hellman, Certificate-Authority, Certificate and Key parameters required for a SSL/TLS based configuration

The `dh`, `ca`, `s-cert` and `s-key` parameters are optional and their presence is used to discriminate between an encrypted and a non-encrypted tunnel setup.
The data duplication that can be noticed in the commands parameters allows for decoupling the two TEP specification while keeping the commands syntax simple (in terms of either grammar complexity and parameters count). The Service Manager performs the parameters mangling, removes the duplications and finally obtains the minimum (TEPD-upstream, TPID, TID) and (TEPD-downstream, TPID, TID) data that will be instantiated into their corresponding FPRO actions.

The introduced commands must be used in conjunction with the `fpr-add-actions` command which binds each action to its specific FPRO, as described in the following example:

```
1. action-tun-upstream  tun1up   tun1down intfA intfB
2. action-tun-downstream tun1down tun1up  intfB intfA
3. fpr-add-actions fprX platform0 tun1down
4. fpr-add-actions frpX platform1 tun1up
```

The first two steps define the upstream and downstream (TEPD, TID) couples while the latter two steps bind them to their respective FPROs.
The Service Manager automatically builds and sets up the correct sequence of FPRO objects. Moreover, it also automatically generates the TPIDs that will be injected into the corresponding FPRO actions.

## 4.2 Tunnel setup example

The following example depicts the Service Manager CLI commands sequence that can be used to create (and then remove) a service across three different platforms and with two tunnels (one tunnel for each signaling adjacent platform pair). The configuration details used in the example (e.g. the interfaces Addresses, the PMs configurations) do not depict a real testbed example and are used only to frame the aforementioned commands into a simple setup.

```
# Interfaces definitions
interface-add intf1 eth0 real 192.168.0.1
interface-add intf2 eth1 real 192.168.0.2
interface-add intf3 eth2 real 192.168.0.3
interface-add intf4 eth4 real 192.168.0.4
interface-add intf5 eth5 real 192.168.0.5
interface-add intf6 eth6 real 192.168.0.6

# PMs definitions
pm-add fw1 image1 Raw True constraints intf1 intf2
```

```
pm-add fw2 image2 Raw True constraints intf3 intf4
pm-add fw3 image3 Raw True constraints intf5 intf6

# PMs configurations
pm-config-file fw1 /home/admin/change/sig/test/svcmgr/file_conf1
pm-config-file fw2 /home/admin/change/sig/test/svcmgr/file_conf2
pm-config-file fw3 /home/admin/change/sig/test/svcmgr/file_conf3

# PMs bindings to their respective platform
platform-add    platform0 10.0.2.187 50002
platform-config platform0 fw1

platform-add    platform1 10.0.2.136 50002
platform-config platform1 fw2

platform-add    platform2 10.0.2.146 50002
platform-config platform2 fw3

# FPR instantiation
fpr-create test

# Platforms binding to the FPR
fpr-build test platform0 platform1 platform2

# Actions definition (between platform0 and platform1)
action-tun-upstream   tun1up   tun1down intf3 intf2
action-tun-downstream tun1down tun1up   intf2 intf3

# Actions definition (between platform1 and platform2)
action-tun-upstream   tun2up   tun2down intf5 intf4
action-tun-downstream tun2down tun2up   intf4 intf5

# Actions binding to the FPR
fpr-add-actions test platform0 tun1down
fpr-add-actions test platform1 tun1up
```

```
fpr-add-actions test platform1 tun2down
fpr-add-actions test platform2 tun2up

# Service Setup Request (creates the service)
ssreq-create ssr1
ssreq-build  ssr1 1 99 fid1 10.0.2.187 test
ssreq-send   ssr1 name1 10.0.2.187 50001 60

# Service Deletion Request (destroys the previous service)
sdreq-create sdr1
sdreq-build  sdr1 1 99 fid1 10.0.2.187
sdreq-send   sdr1 name2 10.0.2.187 50001 500
```

## 4.3 Conclusions

The inter-platform signaling framework introduced in deliverables [2] and [3] has proven to be solid during the I&T phases and this chapter consequently presents only the most relevant improvements and additions. The signaling software has been updated in order to perform the provisioning of the inter-platform tunnels and these changes have been reflected into updates to either the Signaling Manager and the Service Manager, as well as to the framework common libraries. The software has been released in the CHANGE project SCM repository and future works will focus on the integration and validation activities in the WP5 scopes. The signaling framework will be installed into the CHANGE testbeds and it will be integrated with the applications, in order to provide the functionalities required for the validation of the distributed use cases. These activities, and the refinements or fixes they might generate, will be finally detailed in the WP5 reports.

# Conclusion

This document has tackled several important mechanisms that are required to enable CHANGE platforms to process flows in real networks and the global Internet.

Middleboxes tend to carry a social stigma, where even if they are not considered to be the root cause of many of the Internet's afflictions, they are at least seen as a contributing factor. In this paper we have tried to show that while they certainly do not simplify the network, given the right architectural framework middle-boxes, and especially software ones, can be leveraged as a means to invigorate innovation and derive win-win situations for all parties involved, including end clients, operators, and content providers.
The core of CHANGE is its ability to automatically reason about middlebox behavior. Our static analysis tool and the models we have built to describe existing Click elements allow CHANGE to reason automatically about the safety aspects of customer processing requests, and to scale by consolidating many users in a single virtual machine. Our platform can scale to thousands of users simultaneously, hinting that it may just be possible to deploy CHANGE in the wild.
In the second part of this document we showed which methods can be used by a Flowstream platform to decide whether resources are sufficient to handle new incoming processing requests while ensuring a set of performance constraints. This is a variant of the offline problem considered in an earlier deliverable where all requests for processing are known beforehand. We explained how a constraint satisfaction procedure used to solve the offline problem can be adapted for the online problem, where requests for processing appear over time. We also extended this procedure with several alternative components. Different configurations of the procedure were then tested on an online scenario, demonstrating the viability of our method.
Finally, in the third part of this report we described how the inter-platform signaling software has been updated to encompass inter-platform tunnels provisioning.

# Bibliography

[1] D3.2: Flow processing platform design and early implementation.

[2] D4.2: Inter-platform signaling.

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

[4] GeoLite Free Downloadable Databases. http://dev.maxmind.com/geoip/legacy/geolite/.

[5] Google Global Cache. http://ggcadmin.google.com/ggc.

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

[7] Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. SDN and OpenFlow World Congress, 2012.

[8] V. K. Adhikari, S. Jain, Y. Chen, and Z.-L. Zhang. Vivisecting YouTube: An Active Measurement Study. In *INFOCOM*, 2012.

[9] A. Aucinas, N. Vallina-Rodriguez, Y. Grunenberger, V. Erramilli, D. Papagiannaki, J. Crowcroft, and D. Wetherall. Staying Online While Mobile: The Hidden Costs. In *CoNEXT*, 2013.

[10] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: the Virtue of Gentle Aggression. In *SIGCOMM*, 2013.

[11] H. Ballani and Y. Chawathe and S. Ratnasamy and T. Roscoe and S. Shenker. Off by Default! In *HotNets*, 2005.

[12] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Holzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[13] P. Kazemian, M. Chang, H. Zeng G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking using Header Space Analysis. In *NSDI*, 2013.

[14] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: static checking for networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F.Kaashoek. The Click modular router. *ACM Trans. Computer Systems*, 18(1), 2000.

[16] T. Leighton. Improving Performance on the Internet. *Comm. of ACM*, 2009.

[17] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[18] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[19] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS Oper. Syst. Rev.*, 2010.

[20] 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.

[21] 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.