ICT-317756

TRILOGY 2

Trilogy 2: Building the Liquid Net

Specific Targeted Research Project
FP7 ICT Objective 1.1 The Network of the Future

D2.5 - Tools for controlling liquidity

Due date of deliverable: 31st December 2015
Actual submission date: 31st December 2015

Start date of project	1 January 2013
Duration	36 months
Lead contractor for this deliverable	Universidad Carlos III de Madrid
Version	v0.1 , January 11, 2016
Confidentiality status	Public

Abstract

The goal of the Trilogy 2 project is to create resource liquidity with three type of resources, namely, compute, storage and networking. WP1 is in charge of developing the tools to create liquidity and WP2 develops the tools for controlling the created liquidity. This deliverable describes the work done in the third and final year of the Trilogy 2 project in terms of liquidity control tools. This deliverable first presents the Trilogy 2 Information model to describe the resources being pooled. It then details the tools developed to control liquidity from the end user's perspective. Such tools include the use of end to end encryption to modulate the effects of operator's liquidity tools in the end user's traffic, the creation of an incentive framework to foster users to create liquidity in the edges of the network and the development of a MPTCP subflow manager to assist the end user with the creation of MPTCP subflows. This deliverable also contains a set of tools for liquidity control from the operator's perspective. Such tools include the control tools for the Federated Market and Cloud.net and the means for operators to affect the liquidity created by MPTCP. Finally, we describe a few tools to understand liquidity, in order to exert informed control over resources.

Target Audience

The ultimate target audience for this deliverable is the community of knowledge engineers who define the structure of ICT systems, and those who define the standards and frameworks that are necessary for these ICT systems to interwork across the industry. In addition, this deliverable is also targeted at a) the project participants to ensure the whole is understood to be greater than the parts and b) the project's scientific advisory board and reviewers to articulate the approach being taken across the project in order to elicit useful feedback and criticism.

Impressum

Full project title	TRILOGY 2: Building the Liquid Net
Title of the workpackage	WP2 - Tussle over Liquidity
Editor	Francisco Valera, UC3M
Project Co-ordinator	Marcelo Bagnulo Braun, UC3M
Copyright notice	© 2016 Participants in project TRILOGY 2

Executive Summary

The Trilogy 2 project is set to increase liquidity in the network. A liquid network is one where its resources are pooled together behaving as one more capable resource, bringing higher flexibility and efficiency in the usage of resources. The Trilogy 2 project is targeting three types of resources, namely, networking, compute and storage resources. The task of WP1 is to develop the tools for creating liquidity and the task of WP2 is to define the tools for controlling the created liquidity. Control of liquidity is paramount to guarantee of proper use of the pooled resources, to avoid abuses and unfair distribution of resources. This deliverable describes the work on the third and final year of the Trilogy 2 WP2 on Controlling liquidity.

This deliverable structures the presentation of the different control tools as follows: it first presents the Trilogy 2 information model, then it covers the tools for controlling liquidity from the end user perspective, then it moves on to describe the liquidity control tools for the operator and finally it presents tools to understand the liquidity.

The first contribution presented in this deliverable is the Trilogy 2 Information Model. The proposed information model describes the resources to be pooled. This information model allows for a uniform description of the resources and enables the exchange of information of the resources involved. This is a fundamental piece to enable control of resources, since the first step to control the amount and type of resources each user is allowed to consume is to properly quantify them. Using the proposed information model, it is then possible to create specific data model instances that describe the resources at hand. This information model is used to describe the resources traded in the cloud Federated Market, that we describe below.

The second contribution contained in this deliverable is the set of tools for liquidity control from the user‘s perspective. Network users and network providers have in many cases different perspectives, due to the differences in goals, requirements, network usage, etc. It is then natural to provide different tools for controlling liquidity for users and for providers. Users control the fringes of the network and in particular the end systems. Operators control the core of the network. User’s tools for controlling liquidity are mainly end to end. In this deliverable we present three main tools for user control of liquidity, namely, end to end encryption, incentives for the creation of liquidity at the edges and selection of available paths, which we summarize next.

In the Trilogy 2 architecture presented in deliverable D.2.3 we presented end to end encryption as a fundamental tool to enable user control of liquidity. Liquidity tools deployed by the operator, such as Network Function Virtualization, makes the network much more flexible, allowing the operator to deploy network functions in very short time scales. While in many cases this benefits the users, many of the functions deployed by the operator affect user’s traffic in ways the users may prefer to avoid or at least modulate. For example, operators may deploy Deep Packet Inspection capabilities to rate limit some types of traffic. End users can use encryption to regain control of their traffic and determine which part of their traffic is affected by these type of operator network functions. In deliverable D2.4 we described several tools for encrypting end to end traffic, in particular we presented two approaches to secure MPTCP traffic. In this deliverable, we present two contributions related to end to end encryption. First, we performed a large scale feasibility study of the deployment and adoption of end to end encryption in the Internet. Proposed tools are only useful if they can be deployed in the Internet. By doing a large scale deployment study, we can assess if the proposed tools will work in the Internet and if not, how they need to changed in order to be deployable. We find that in most ports except for port 80, end to end encryption is deployable. Building on this conclusion from the feasibility study, we deploy a specific tool to encrypt web traffic, called HTTPCrypt. HPPTCrypt provides opportunistic encryption (like TCPINC, described in deliverable D2.4) but specific for web traffic. By making the encrypted content to look like regular HTTP traffic, HTTPCrypt manages to overcome the deployment limitations faced by TCP opportunistic encryption.

As described in the Trilogy 2 project Description of Work, incentives play a fundamental role in controlling liquidity. Rational players behavior is determined by the available incentives, so setting incentives right is one fundamental step towards achieving the desired behavior. In this deliverable, we present Kadupul, a tool for creating the incentives for end users to make liquidity available in the fringes of the network. Current network possess a rich interconnection in the fringes thanks for a myriad of available wireless technologies, such as wifi, bluetooth etc. This mesh of interconnection could enable direct user to user communication, or at least a larger networking pool to interconnect end users with a low latency. However, this connectivity remains largely unused due to lack of incentives to make it available for other users. Kadupul introduces

micro-payments in the form of Bitcoins to allow users to pay other users who are willing to transport their packets towards their destination, unlocking the connectivity potential at the edge of the network.
MPTCP makes multiple paths available to the end user. It is then critical for the end user to determine how these path are going to be used to control the liquidity created by MPTCP. The next contribution contained in the deliverable is the MPTCP subflow manager. When a MPTCP connection is started, MPTCP has the possibility to initiate many subflows e.g. using all the possible source and destination IP addresses available. However, creating all possible subflows may or may not be the optimal strategy, depending on the specific user needs. The MPTCP subflow manager controls which and how subflows are created. We present different strategies for different use cases.
The third contribution contained in this deliverable is a set of tools for control of liquidity from the operator's perspective. The Federated Market is a platform created in the Trilogy 2 project to enable cloud providers and client to trade cloud resources. Cloud providers make available compute, storage and networking resources for clients to hire. It is fundamental then for all the stakeholders to have the means to control the usage of resources. The Federated Market relies on the Information Model presented earlier to describe the resources traded. It also develops incentive mechanisms based on payment mechanisms to ensure proper usage of resources. It also provides built in enforcement mechanisms to prevent inappropriate use of resources in the market place. All these mechanisms are described in this deliverable.
We have presented MPTCP as a core technology to create network liquidity. MPTCP is inherently an end user technology, as TCP resides in the end system. However, operator will carry the MPTCP traffic, so it is natural for them to try to exercise some form of control over the network resource pools created by MPTCP. We explore how the operators can use packet drop as a control signal to affect MPTCP behaviour. We provide a game theory model that allow us to explore different ways the operator can use packet drops to divert or attract MPTCP capable traffic.
The fourth and final contribution in this deliverable is a set of tools for understanding liquidity. In order to properly control liquidity it is important first to understand liquidity. We present two tools for understanding liquidity, namely a Web dependency graph analyser and Symnet, a symbolic execution environment for network functions.
The Web dependency graph analyzer allow us to understand the structure of modern web pages and to explore how web content can benefit from the available liquidity tools, such as multipath transport. Indeed, as modern web pages contain several components, it is possible that it is beneficial to use different paths to transport different web objects composing a given web page. This and other relevant question can be addressed using the web dependency graph analyser, helping then determine the number of subflows that would be optimal to have in a given MPTCP connection (feeding into the MPTCP subflow manager discussed earlier).
As described in Deliverable D2.3, Network Function Virtualization is a core technology to create liquidity in the network. It allows to dynamically deploy network functions in very short time scales. However, due to the rapid deployment and large number of network functions, it also makes the network harder to reason about. It is then more challenging to understand how the traffic is processed in an NFV based network. In this deliverable we describe Symnet, a symbolic execution framework for network functions. Symnet explores all possible execution paths, that different packets can take through a network (i.e. all possible combinations of network functions) to expose how packets are treated when deploying a number of network functions.
Through the described tools, we believe that the different stakeholders will have a rich tool set to understand and control the liquid resources in the network.

List of Authors

Authors	B. Hesmans, R. Bauduin, O. Bonaventure, C. Raiciu, M. Popovici, R. Stoenescu, V. Olteanu, J. Thomson, T. Moncaster, I. Sidorov, J. Chesterfield, M. Bagnulo, Jon Crowcroft, Anil Madhavapeddy
Participants	UCL-BE, UPB, ONAPP, UC3M, UCAM
Work Package	D.2.5 - Tools for controlling liquidity
Security	Public (PU)
Nature	R
Version	v0.1
Total number of pages	89

Contents

List of Figures

List of Tables

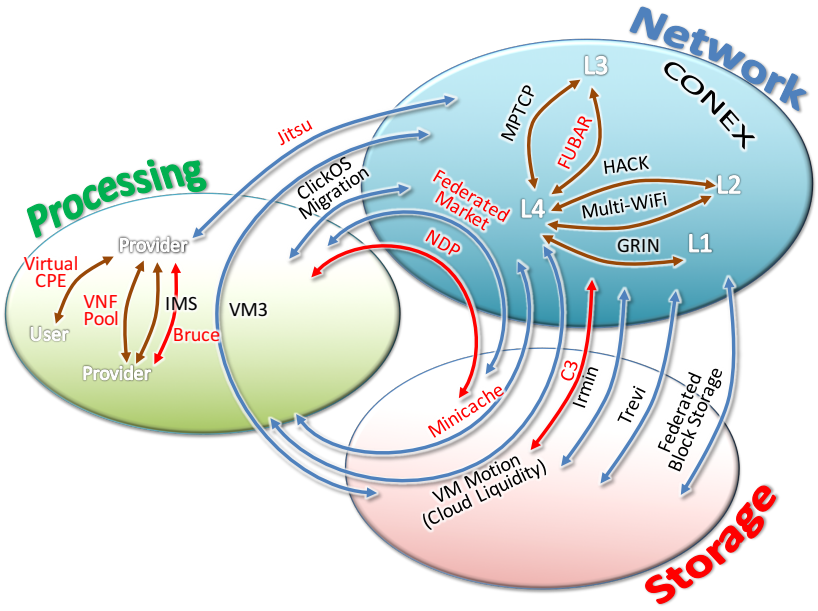


Figure 1.1: Trilogy 2 use-cases and architecture

1 Introduction

The goal of the Trilogy 2 project is the creation of a more liquid Internet. In order to achieve that, the project has produced over the last three years a number of tools aimed to create liquidity from the available network resources. Liquidity is created by pooling available resources. A resource pool is a set of resources that behaves as a single more potent resource. The three main types of resources that the project has focus its effort on are networking, processing and storage resources. Some of the tools created by the Trilogy 2 project also allow for trading resources, so that it is possible for different parties to trade one type of resource for another (e.g. to use more processing to reduce the use of storage). The tools for creating and trading liquidity are developed in WP1 and in particular the tools developed in this third year of the project are described in D1.3. The different tools produced during the whole project lifetime are depicted in Figure 1.1. The figure shows the different resources pooled by the different tools.

While the creation of resource liquidity is in general beneficial as it allows a more efficient use of resources, it can be negative if not properly controlled. For example, by pooling all available resources, it is possible for a single user to (ab-)use all the resource pool, consuming all available resources. Other forms of detrimental effects can also be identified. In order to avoid such negative effects, it is then necessary to provide the tools to control the extent of liquidity. In this deliverable, the describe the different tools produced in the third and final year of the Trilogy 2 project for controlling the resource pools.

In order to control liquidity, it is first necessary to uniformly describe the available resources, so that information characterizing the resources can be exposed and exchanged among the different stakeholders involved. We achieve that by defining an Information Model for the resources being pooled. The first contribution contained in this deliverable is the Trilogy 2 Information Model for the different types of resources being pooled by the different tools created in the project. The Trilogy 2 Information Model is described in Section 2). Using the Information Model, we can then exchange information and reason about the different resources being pooled. We can use the information model to properly define service level agreements or other forms of definitions involving the amount of resources that each user is entitled to.

In addition to the Information Model, we have produced a number of liquidity control tools. These tools provide different means to control liquidity, including the means for creating the incentives for users to do a proper use of the available resources and the tools to enforce the proper usage of resources.

In this deliverable, we present three main types of control tools, namely, tools for user control of liquidity, tools for operator control of liquidity and tools for understanding the extent of liquidity.

In Section 3 we describe the tools for enabling the end-user to control liquidity. As we described in the Trilogy 2 architecture presented in Deliverable D2.3, end-to-end encryption is a cornerstone mechanism to enable user

control of liquidity. By enabling the end user to encrypt its traffic, we allow the end user to control in what extent its traffic is affected by the liquidity mechanisms created and controlled by the operator, in particular by the Network Function Virtualization mechanisms presented in Section 5 of Deliverable D2.3. Significant work on end to end encryption mechanisms as liquidity control tools for the end user was reported in D2.4. Section 3 presents the work on this last year of the project in this area.

In addition to end to end encryption, Section 3 describes two other mechanisms for user control of liquidity, namely a MPTCP subflow controller and Kadupul, a mechanism to create the incentives for creating liquidity. The MPTCP subflow controller provides the means for the end user to control how to use the available network resource pool by selecting which path to use to actually forward data. This work covers some of the control aspects of the MPTCP work done in the project, in particular the work reported in Deliverable D1.3. The other mechanism presented in Section 3 for user control of liquidity is Kadupul. Kadupul is a mechanism to create the incentives for end user to create liquidity. As described in the Trilogy 2 Description of Work and in particular in Task 2.3, incentives are an important mean to provide control of liquidity, because given the right incentives, the different parties involved will behave as expected (assuming they are rational). Kadupul focusses in creating the incentives for end users to contribute with their available resources to create networking pools in the very edges, where the operator has little or no resources.

In Section 4, we present the work done in this last year of the Trilogy 2 project in terms of tools for enabling the control of liquidity from the operator side. We first present the control tools for the Federated Market and Cloud.net and then we move on to describe how the ISPs can try to control the liquidity create by the end users using MPTCP. The Federated Market was introduced in Deliverables D1.3 and D2.4 as a technology to enable different cloud providers and clients to trade cloud resources. It benefits from the proposed Information model to describe the different resources traded. The tools described in Section 4 provide both the incentives (though payments) and the enforcement tools to control that the resources consumed are coherent with the resources traded. We next describe the tools that operators can use to control the liquidity created by user using MPTCP. MPTCP is an end-to-end protocol. This means that it is up to the end users (client and server) to deploy and use MPTCP without any involvement from the operator in the middle. However, MPTCP affects the traffic patterns exposed to the operators. As such, it is only natural to question how the ISPs will/can react to the MPTCP traffic. In Section 4 we explore how the ISP can use packet dropping to shape the MPTCP-enabled end user traffic.

Finally, in Section 5 we describe tools for understanding liquidity. We believe that in order to properly control liquidity, it is important to understand certain aspects of the liquidity pools available. We present tools for understanding two key aspects of liquidity, namely a Web dependency graph analyser and Symnet. The Web dependency graph analyzer is a tool that allow us to understand how the modern web protocols would interact with MPTCP and other multipath transports and in particular how much benefits can MPTCP provide when it is used to deliver Web traffic, given the nature of the modern Web pages. Symnet is a tools that enables symbolic execution of network functions. A core part of the work done in Trilogy 2 was on developing various NFV concepts. In an NFV capable network, network functions can be dynamically deployed and combined to fit the operators needs. However, such flexibility makes it much harder to understand and reason about the network. In this section we describe Symnet, a symbolic execution tool that by exploring all possible execution paths for the functions available in the network, it allows to gain deeper understanding of the behavior of the combination of deployed network functions.

2 Information Model

2.1 Introduction

In recent years the Internet has become a much more fluid system. Processing, storage and bandwidth usage from different machines and different parts of the network are being pooled together. Users draw resources from the pool, making trade-offs between their different requirements. The mechanisms that allow such liquidity all create resource pools. A resource pool is a collection of individual resources which together act as a single more capable and more robust resource. Once a resource pool has been created, it becomes a source of liquidity, as the elements of the pool are interchangeable. This does not however imply that the elements of the pool are homogeneous they will typically vary in capability and location, so moving demand between them affects performance and redistributes costs.
From the description of work -

> *This task will deliver technology to describe resources in the pool, making this information available to the interested stakeholders. It will determine the information model regarding resources, describing their salient features such as geo-location, topological location, congestion, cost etc., which may then be used for resource discovery and selection. It will develop or adopt a mechanism to disseminate dynamic resource information in a timely manner.*

2.2 Constructing the Information Model

An Information Model defines the concepts specific to a domain and defines the relationships between these concepts and is an extension of the principles behind Entity-Relationship models[10]. Once defined, an Information Model is then used for generating data models that are specific to particular parts of the system. A data model contains application specific implementation details that can be used by system designers for generating working applications that conform to the information model.
The Trilogy 2 Information Model has emerged organically from the work done throughout the course of the project. In the following section we use a simple example to show how the model has been derived, describe the model itself, give some details of how the model applies to the Trilogy 2 use cases and provide some background on alternative approaches.

2.3 Deriving a simple Information Model

To understand the concepts relevant to Trilogy 2 it was important to first gather the set of use-cases for the various systems being worked on in Trilogy 2. The use-cases have been captured in the initial Information Model working document D2.2 and have been subsequently refined in the use-cases work package in WP3. The use-cases have been generated by the individual Trilogy 2 partners who are working on systems related to resource liquidity. By covering a broad set of use-cases we intended to capture a broad domain with seemingly unrelated concepts to investigate how they may be related and brought together to enable the Liquid Net proposed by Trilogy 2.
We have performed an analysis of the use-cases described in the original Deliverable D2.2 document and captured the concepts embedded in thos use cases. These are described in a first pass that resembles the model produced by mOSAIC[33]. The DMTF Common Information Model or CIM[15] was seen as overly complex for the domains that are covered in Trilogy 2, but it is suggested that at a later stage the model could be made to conform with the CIM.
The following simple example highlights the methodology used to derive the Trilogy 2 Information Model. It is based on the Cloud Liquidity Use Case (see section 4.2 of D3.2). This Use Case is about migrating virtual machines (VMs) across the wide area network. As a minimum a virtual machine consists of some form of virtual storage disk (vDisk), a virtual network interface (vIF) and some state defining what underlying hardware is needed to run the VM (e.g. type of hypervisor, amount of virtualised resources, etc). In order to migrate the VM to a new location you need suitable hardware to be able to run this VM (same type of hypervisor, sufficient resources) and you need to be able to transfer a copy of the vDisk to the new location.
Looking at each of these elements in turn we extract the underlying concepts captured in **bold** text and in *italic* for the relationships:

- A **vDisk** has a given **size** (bytes) and is *stored* as a given number of **blocks** in one or more **stripes** on one or more **physical disks** *attached* to one or more **servers**. The **vDisk** will *contain* a **filesystem** which will *contain* the **VM OS** and **data**.

- The **VM** has **state** *associated* with it describing its **virtual resources (VIFs, vDisks, etc.)**, its current **power state**, its **VM Owner** and any **state** relating to **Security**.

- The **VM** *runs* on a **server** which has a **Processor** of a given **Architecture** and with a number of **cores**. The **server** has a given amount of **RAM** and *runs* a **Host OS** with a **Hypervisor**. It has one or more **NICs** *connected* to one or more **networks**. These **networks** may have multiple **VLANs** defined and the **VM** will be *associated* with various **network Addresses**.

- The **server** is *part* of a **cluster (cloud)** *associated* with a **Cloud Owner / Operator**. The **cluster** is within a **Datacentre** which is *run* by a **Datacentre Operator**. The **Data centre** will have a known **physical Location**.

- The **wide area network** is run by a number of **Network Operators**. It will use some form of **Routing** in order to ensure **data** reaches its **destination**. It may have a number of **Middleboxes** *interacting* with the **Data flows / subflows**. Individual **data flows** may be **Secured** using a **protocol** such as TLS or SSH and they may be *part* of a **multi-path flow**.

The above is far from complete but demonstrates how the process works. Alongside the list of concepts there are also a number of relationships that need to be defined. For instance for a VM these include Create, Power On, Power Off, Migrate and Backup. There are also various actions relating to monitoring resource usage, billing, security, AAA, etc.

2.4 The Trilogy 2 Information Model

An Information Model defines the concepts and abstract elements of a system as well as the relationships between those abstract concepts. Throughout the project we have divided resources into three broad categories using a similar approach to that adopted in the OpenStack architecture[1] as seen in Figure 2.1:

- Processing

- Storage

- Network

As we stated in the Trilogy 2 Architecture, "Transport, Processing and Storage are all Strongly Inter-Dependent". In other words while an action may be primarily related to one resource type in most cases it also involves either one or both of the others. In the Architecture document we gave the example of writing some data to storage. While this is clearly mainly a Storage related action it also requires Processing (Compute) and in many cases also leads to some information being transferred over the Network (in the broadest sense, most inter-process-communication on a modern multi-core server involves some form of networking). Clearly within these broad categories there are a large number of specific resources and concepts that need to be captured in any comprehensive Information Model. While this Information Model has been developed in the context of the specific Trilogy 2 Use Cases, we have aimed to make it generic enough to be applicable more widely.
The table below attempts to capture the full set of resources and concepts needed for the Information Model. A top level diagram that shows how these concepts are represented in the Trilogy 2 Information Model is shown in Figure 2.2. These distinct concepts combine together along with the concepts described at lower levels to capture the majority of the Trilogy 2 use-case concepts.

[1] https://www.openstack.org/software/

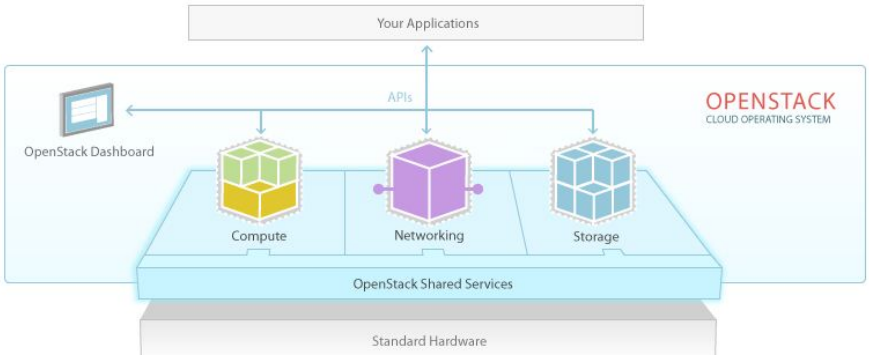


Figure 2.1: OpenStack early Architecture from 2011

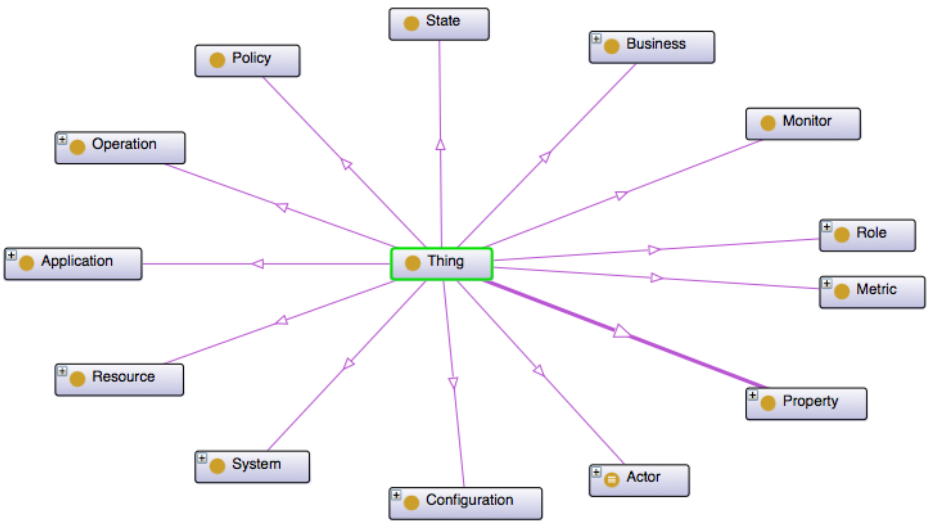


Figure 2.2: The Trilogy 2 Information Model top level concepts

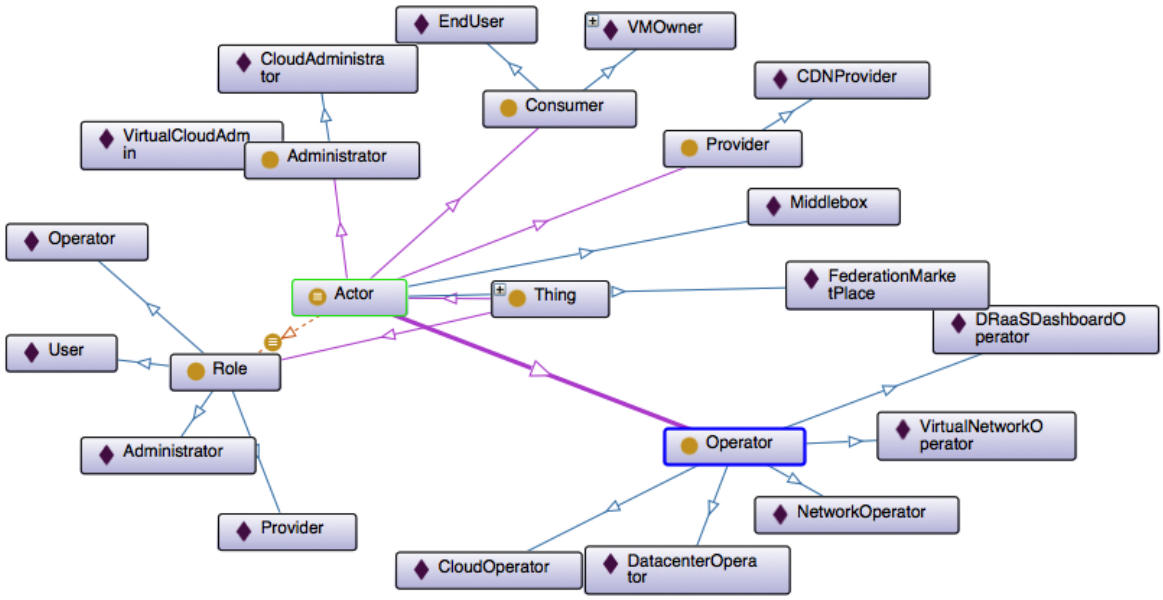


Figure 2.3: Actors as captured in the Trilogy 2 Information Model

2.4.1 Actors

As an example of how the Information Model was generated we capture the terms relevant to the Federation and Market actors in Table 2.2. This table captured high-level terms that could then be split into two conceptual elements; actors and roles. The set of Actors could then be categorised into a scheme that differentiates the type of individual from the type of role that they perform. A visual representation of these concepts as captured for the actors can be seen in the OWL model visualisation shown in Figure 2.3.

Table 2.2: Actors within the Information Model (**O**perator, **C**ustomer, **U**ser)

Role	**Type**			**Notes**
	O	**C**	**U**	
Cloud Operator	✓	✓	×	
Cloud Administrator	×	✓	×	May also be cloud operator
Virtual Cloud Admin	✓	✓	✓	Is both a user and customer of the cloud operator
Datacentre Operator	✓	✓	×	
VM Owner	×	✓	✓	Customer of Cloud/Virtual Cloud. User of service.
Federation Marketplace	✓	×	×	Acts on behalf of users and Cloud Operators.
DRaaS Dashboard	✓	×	×	Acts on behalf of users and Cloud Operators
Network Operator	✓	✓	✓	May be customer (VNF)
Virtual Network Operator	✓	✓	✓	
CDN Provider	✓	✓	×	Often also a Cloud Operator
Middlebox	✓	×	✓	May be enabled as VNF for ISP.
End User	✓	✓	×	

2.4.2 Operations

Some of the more complicated elements to represent and conceptualise in the Information Model were the various functions and operations that would be performed. There are some protocol specific functions that we denote as bottom-level members in the Information Model. These members are just particular instantiations and do not cover a concept themselves. Given the wide-spectrum of types of operations as can be seen in Table 2.3 it meant that some of the concepts pertaining to state also had to moved into a separate concept.

Table 2.3: Operations relating to the Information Model

Operation Type	Operations
Connection and Session Management	• Handshake • Congestion Control • Flow Control • Keepalive • Connection Teardown • Flow Handover / Handoff • HTTP Daemon • TCP Re-termination
Network Functions	• BGP • OSPF • VPN • Unicast • Anycast • Multicast
Monitoring and Management	• Congestion Exposure • Data volume • DPI and Flow Classification • VM Usage
Security	• Access Control • Authentication • Authorisation • Intrusion Detection • Encryption
Data Input / Output	• Write Data • Read Data • Copy Data • Cache Data
VM Lifecycle	• Create • Destroy • Migrate (both hot and cold) • Power Operations (On, Off, Reset) • Rate-limiting (of Network and I/O) • Modification of VM (change vCPU, RAM, Storage, vNICS)
Communication	• IPC

A visual representation of these concepts can be seen in Figure 2.4

2.4.3 Systems

- vCPE
- VM
- HV
- router
- Virtual routing forwarding instance (VRF)
- VR (VMs that implement routing protocols)
- Point of presence

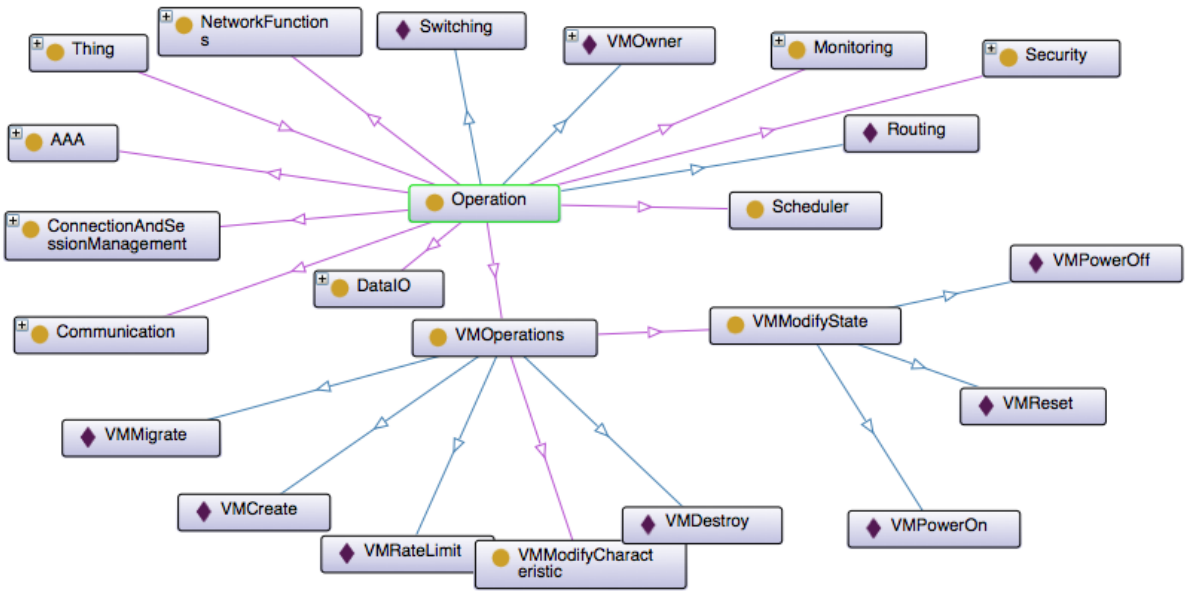


Figure 2.4: Operations as represented in the Trilogy 2 Information Model

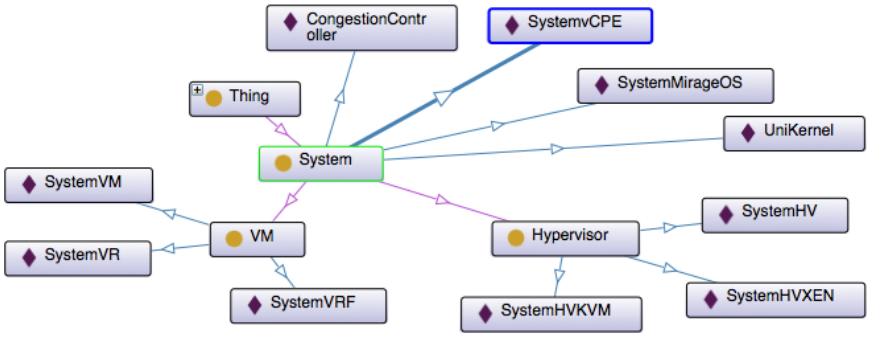


Figure 2.5: System representation in the Trilogy 2 Information Model

- Storage cache (minicache)
- MirageOS
- KVM / XEN
- Uni-kernel

A visual representation of these concepts can be seen in Figure 2.5

2.4.4 Software

- HTTPd daemon - Apache
- Monitoring system
- Scheduling system
- Database - MySQL
- File-system
- Files
- Web-browser
- VOIP clients

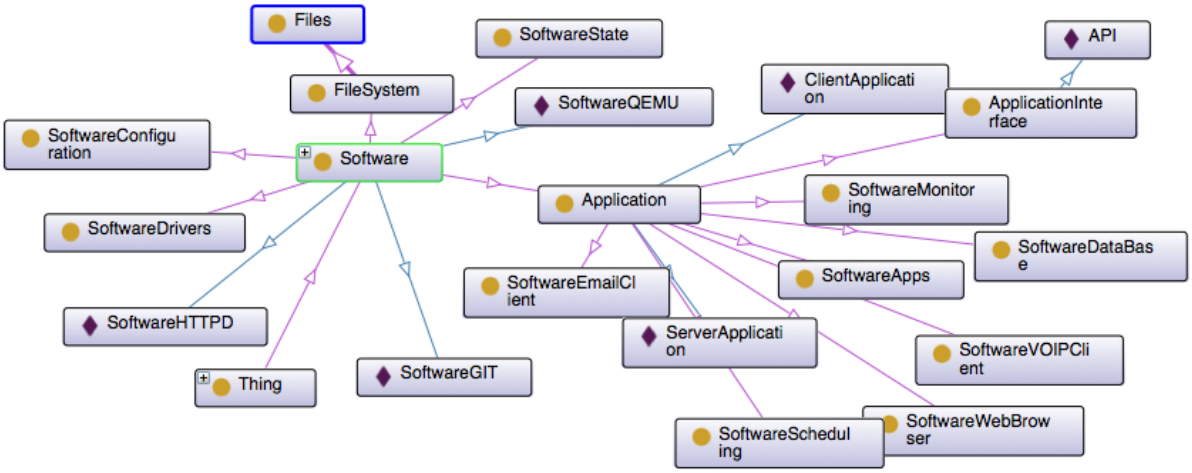


Figure 2.6: Software representation in Trilogy 2 Information Model

- Email clients
- Apps
- qemu
- drivers
- (git)

A visual representation of these concepts can be seen in Figure 2.6

2.4.5 Context

- Configuration
- State (VM On / Off, Connection up / down, etc)

2.4.6 Business

- Pay-as-you-grow
- Pay-per-use
- Pay for metrics that they want to improve
- Contract / licences
- Costs

  Electricity

  Power (Watt / SI J/s)

- Charges

  Price (USD / GBP / Euro)

2.5 Example of applications of the information model

The Information Model as described in previous section captures the concepts relevant to Trilogy 2. The Information Model can be used to generate Data Models that are applicable to specific applications that will conform with the Information Model. For instance the Information Model can lead to the generation of a Data Model such as the one used by the OnApp platform as seen in Figure 2.10. Data Models contain implementation specific details that are less generalised and more useful for system designers. Different data models can be derived from the same Information Model. For instance the high-level concepts captured in

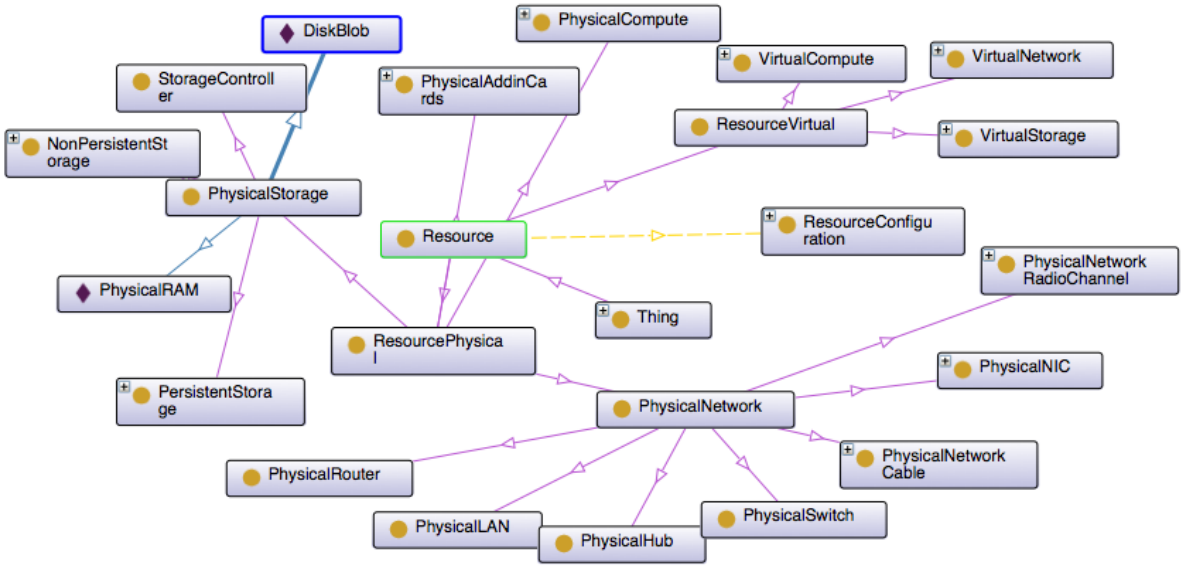


Figure 2.7: Resource representation in the Trilogy 2 Information Model

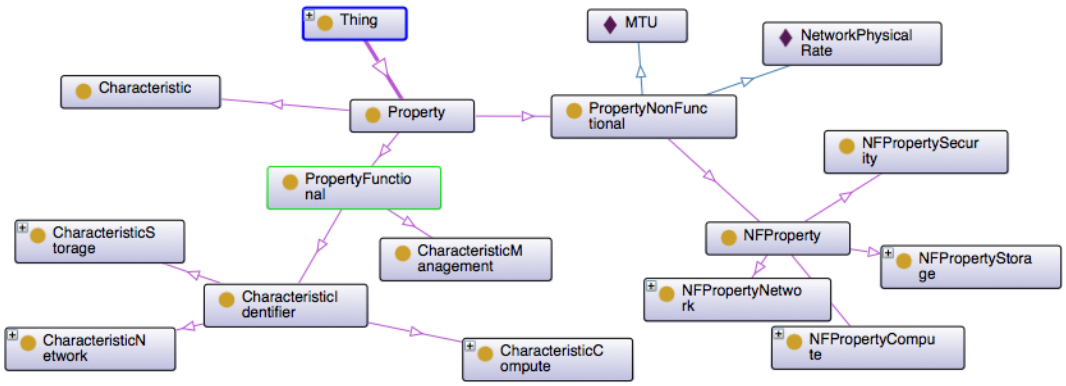


Figure 2.8: Properties representation in the Trilogy 2 Information Model

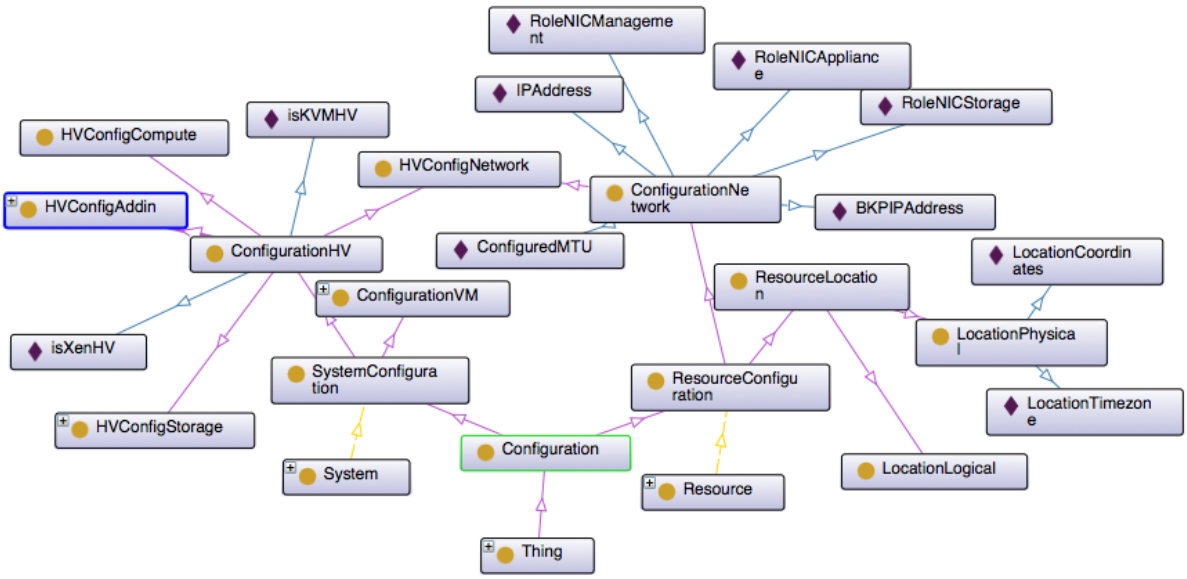


Figure 2.9: Configuration representation in the Trilogy 2 Information Model

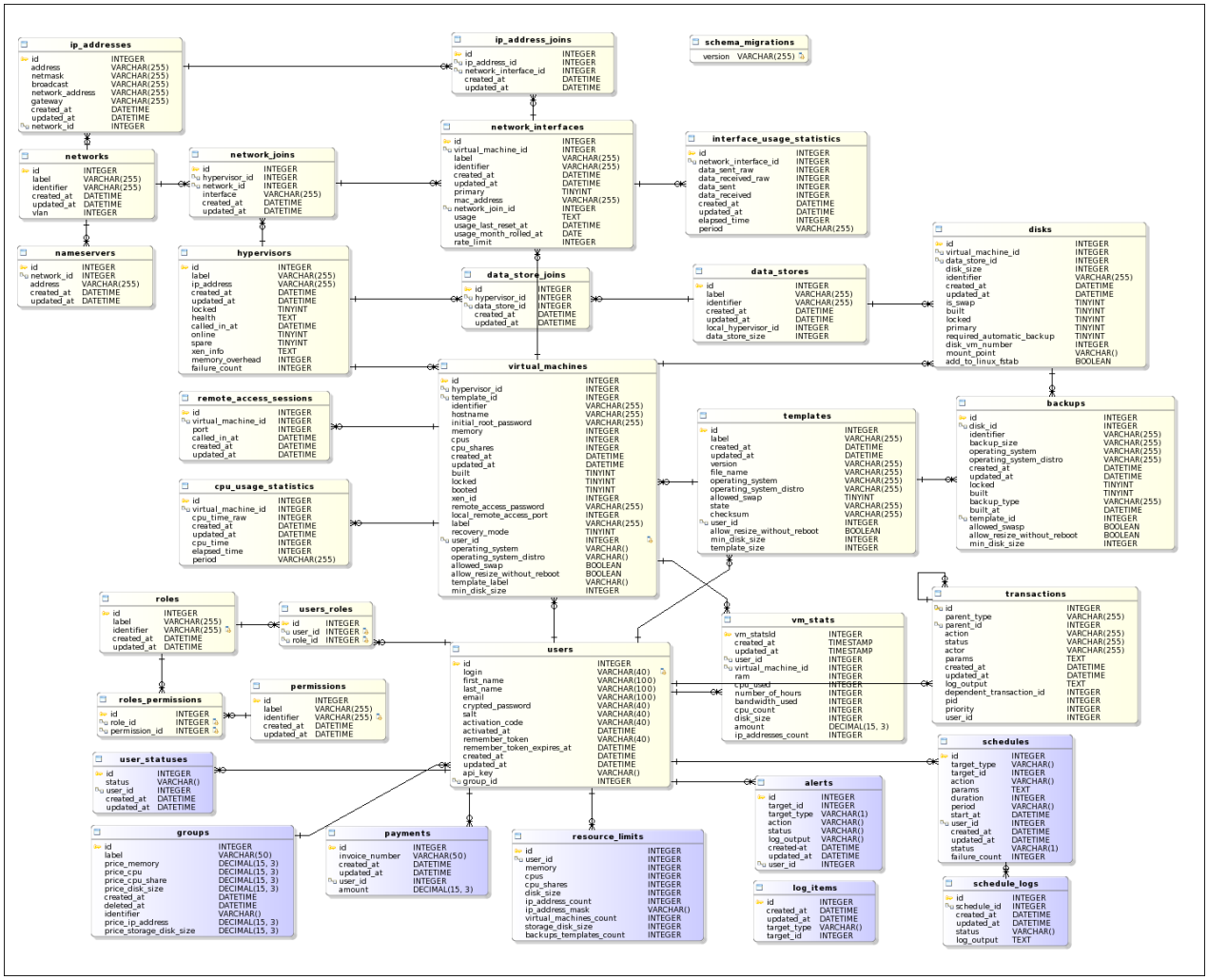


Figure 2.10: OnApp DB Schema

the Information Model could equally map to the OpenStack data model (see Section 2.5.4). Concepts that are not present but needed by the OpenStack data model they could be incorporated by relating them to existing concepts and adding missing concepts as necessary.
We now describe how the Trilogy 2 Information Model can be applied to three of the Trilogy 2 use-cases. The three use-cases used to demonstrate the usage of the Information Model are:

(i) Transparent compute migration

(ii) Disaster Recovery as a Service (DRaaS) that uses Storage and Network resource pools

(iii) Multi-WIFI to demonstrate the use of Network resource pooling

2.5.1 Transparent Migration Example

The Intel Virtual Machine Migration for Mobile (vM3) devices use-case as described in WP3 has many similar concepts to SWBRAS and VM Migration across cloud sites so will be used as a representative example for how the Trilogy 2 Information Model can be applied to these use-cases.
The following text summarises the use-case description:
The Android implementation is composed from two main parts: Migration Services(MS) and VMSlot (VMs), as seen in Figure 2.11. Migration services manage resources on the mobile device, and exposes capabilities to other nearby devices. Migration Services run on each devices that implements vM3. MS identifies other devices on the local network that are capable of receiving apps and communicates with them to access their resources without any need for continuous user intervention. The VMSlot is the app container solution for Android. The solution is based on QEMU and requires KVM to be enabled in the host OS. By using hardware

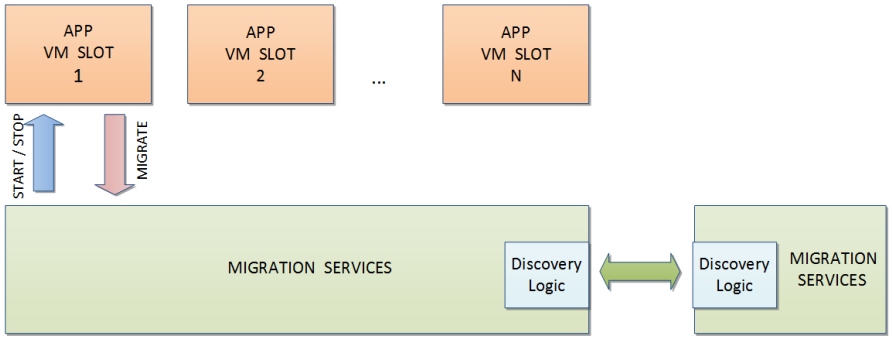


Figure 2.11: The Intel Virtual Machine Migration for Mobile devices architecture as captured from D3.2

virtualization technologies, the app container will run with near-native performance, so the user wont see the difference between an Android app and the vM3 solution. The guest OS will start a stripped-down version of the Chromium browser that will handle the streaming. The MS will start and manage VMs, based on each specific app requirements and available hardware. When the user starts an app, a small footprint OS is booted and initialized inside a container, and the web app is ini- tialized afterwards. The MS will manage the containers IP address, routing and other migration parameters. VMSlot also accounts for appication isolation, based on the fact that in Android an application cannot have more than one instance running at the same time. When the user wants to initiate migration to another device, a request is sent to the MS asking for the list of neighbouring peers. The user can then choose a peer or manually enter the IP address. The MS will negotiate the migration parameters (e.g. throughput, maximum accepted downtime, etc.) and will initiate the migration. During the migration, the applicatiob still runs on the first device. When the state is fully synchronized, the application shuts down on the first device and starts on the receiver, resulting in a user-perceived downtime of less than one second.
This can then be represented by Trilogy 2 Information Model concepts:
A **MobilePhone** *hasOperatingSystem* **AndroidOS**. The **AndroidOS** *contains* two **SoftwareServers**; an **Application** that *performsOperation* **OperationMigration** and empty **SystemVMs** (VM Slots). The Migration services **Application** *hasRole* **RoleManager** and *performs* operation *CommunicationBroadcast* to other **Systems** *inLocality* **PLocationLocal**.
To support this behaviour the **System** must *run* a **SystemHVKVM** (KVM Hypervisor) and **SoftwareQEMU**. At the time an **EndUser** *performs* **MigrationOperation** a **CommunicationMessagePointToPoint** *sends* to the Migration Service **Application** and a list of **PLocalityLocal** (local) **Systems** is displayed. Once a peer is selected the Migration Service **Application** *negotiates* the **NetworkSession** and *performs* **OperationMigration**. The **StateRunning Application** continues to run. The **SoftwareData** *replicates* to the **destination**. Once **StateSynchronised** the **State** is changed for the **Application** on the **Source System** and the **Application** *starts* on the **Destination System**. This results in a **MetricDowntime** as perceived by the **EndUser**.

2.5.2 DRaaS

The DRaaS use-case is captured in the 7 in Section 7.1. This use-case describes storage and network liquidity. To allow for DRaaS there must be three **Actors** in the system; **DRaaSOperator**, **Consumer** and **Supplier**. To ensure true resilience in the face of a disaster, the **Consumer** and **Supplier** of DRaaS should be in different locations, e.g. have *hasLocality* **PLocalityRemote**. The **Consumer** will *performRole* **RoleManage** a **Cloud** that *contains* **SystemVMs** on **SystemHVs**.
On selecting the DRaaS service the **Consumer** *hasContract* **BusinessSLA** with the **DRaaSOperator**. Similarly the **DRaaSOperator** *hasContract* **BusinessSLA** with the **Supplier**. The **HVZone** *enables* **OperationReplication** and a **NetworkLink** is set up between the two **Clouds**. The **SoftwareData** will *replicate* as **DataBlocks**. A **ShadowVM** *runs* on the **Supplier Cloud** until there is a failure event on the **Source**. At this point a **OperationFailover** can be performed. A **SystemVM** corresponding to the **Source SystemVM** *starts* on the **Destination**. The **ShadowVM state** is changed to **VMPowerOff** and the **VM** *runs* on the **Destination Cloud**. The **Pricing** while running on the **Destination Cloud** has greater **MetricPrice** than running the **ShadowVM**. Once the failure condition is resolved the reverse procedure can be performed with **OperationFailback** performed.

2.5.3 Multi-WIFI

Multi-WIFI is described in greater detail in D3.3 Section 2.2. It is summarised briefly here for convenience:
Traditional WiFi mobility techniques (as with all other L2 mobility mechanisms) are based on the concept of fast handover: when a mobile client exits the coverage area of one Access Point (AP), it should very quickly find another AP to connect to, and quickly associate to it. There is a great wealth of research into optimizing fast handover including scanning in advance, re-using IP addresses to avoid DHCP, synchronizing APs via a backplane protocol, even the using additional cards to reduce the association delay. This seems to be the wrong approach for many reasons including:

(i) To start the handover mechanism, a client has to lose connectivity to the AP, or break-before-make

(ii) There is no standard way to decide which of the many APs to associate with for best performance

(iii) Once a decision is made, there is no way to dynamically adjust to changes in signal strength or load

The solution proposed by MultiWifi is to associate to multiple access points using MPTCP and then to increase and decrease the flow as the different sub-flows status change. Experimental results as described in D3.3 are promising.
The use-case uses a mobile phone that has access to the Internet through WIFI available on multiple Access Point routers. The use-case assumes that the end-user is allowed to use any of the access points.
Describing this use-case through the Information Model:
An **EndUser** *hasRole* **RoleOwner MobilePhone** and connects to the Internet through multiple **AccessPoints** that offer **NetworkWifi**. The **AccessPoints** may have different owners, e.g. **Providers** *hasRole* **RoleOwner**. As the **MobilePhone Property Location** changes the **NetworkSpeed** offered by the different **AccessPoints** will differ due to differences in distances. **MPTCP** offers an aggregate of multiple **TCP NetworkFlows**. By establishing multiple **SubFlows** it is possible for the **Client System** to weight particular **Subflows** more highly and get closer to full **NetworkSpeed** supported by the **WIFI NIC**.

2.5.4 Applying the Information Model to Openstack

OpenStack[2] is one of the most popular open source software platforms for Cloud computing. It is primarily an Infrastructure-as-a-Service (IaaS) platform and is a non-commercial alternative to OnApp. It is used by many research communities and is often the de-facto platform used for Cloud management projects that need more control than is allowed by commercial providers such as Amazon AWS who manage the platform online. OnApp is closer to OpenStack as it provides the software for the management of Cloud resources under licence, whereas Amazon you can only use the web platform and APIs to manage the resources remotely provided.
The overall functionality of OpenStack is to create and manage Cloud resources and is captured (non-exhaustively) as a collection of services in the diagram shown in Figure 2.12 (source Wikipedia). As seen in that diagram apart from the high-level management features and services that are used to manage collections of resources and apply policies and strategies, the main resource management system is OpenStack Nova[3].
Nova maps well into the Information Model as the majority of concepts are the same between OnApp and OpenStack and the OnApp platform has been used as one of the primary constituents of the Trilogy 2 Information Model. Nova doesn't currently have an Information Model as far as the authors are aware. There is however an extensive data model that is provided in the form of APIs that are in active development. However, a conceptual architecture as generated for the Juno version of OpenStack captures how Nova interoperates with other components in OpenStack at an abstract level and could be mapped to an Information Model. This is captured in Figure 2.13.
Taking Figure 2.13 into consideration the following mapping can be made to the Trilogy 2 Information Model. OpenStack HEAT can be mapped into a **System** that has **Role**, **RoleOrchestrator**. Horizon is a **System** that has **Role Provider**. Horizon also *implements* **SoftwareUI**. Each of Cinder, Nova, Neutron and Glance are **System** that have unique **Role** for *provide* a unique **Resource** to a **SystemVM**. Nova additionally has **Role RoleAdmin** of the **SystemVM**. Swift has **Role RoleBackup**. Ceilometer has **Role RoleMonitor** that *perform* **MonitorActivity** for the other **System** and the **SystemVM**. Keystone independently has **Role RoleACL** that *provide* **Permission ACL** for the other **System**.

[2] http://www.openstack.org/
[3] https://wiki.openstack.org/wiki/Nova

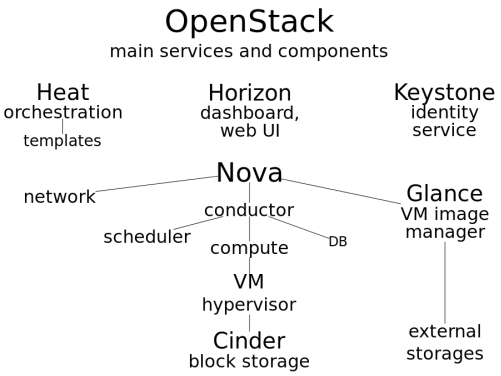


Figure 2.12: A tree diagram showing the principle OpenStack services provided

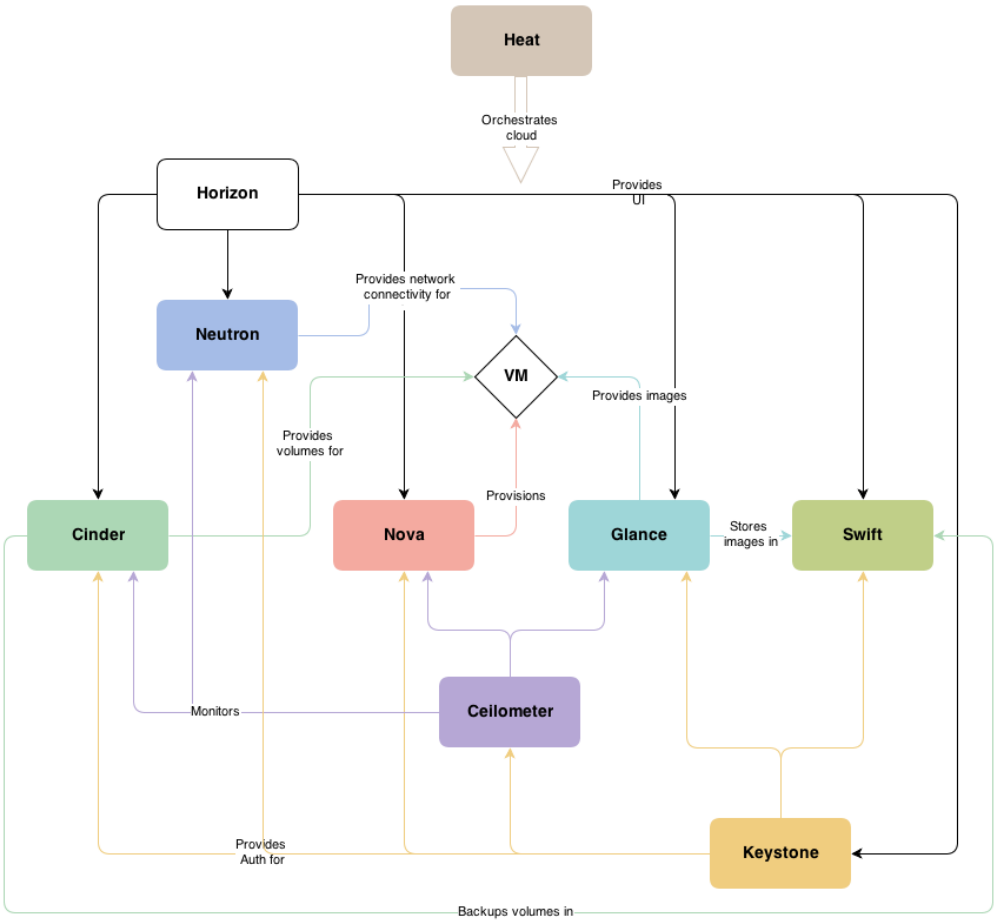


Figure 2.13: The OpenStack conceptual architecture as captured for the Juno version of OpenStack

It is relatively easy to map the concepts in the Information Model to OpenStack. CPUs (IM: **PhysicalCPU**) are commonly referred in Openstack as virtual cores (IM: **vCPU**) and their management is handled by the ***nova*** component. Compute load (IM: **MetricCompute**) is modeled as a set of metrics regarding the activity (**MetricActivity**) of a guest Virtual Machine (IM: **SystemVM**). To reliably collect data (IM: **MetricStats**) on the utilization of the physical and virtual resources comprising deployed guests, persist these data for subsequent retrieval (IM: **DataRead**) and analysis (IM: **OperationAnalysis**), and trigger actions when defined criteria are met, the following components are used:

- ***Ceilometer*** (data collection)
- ***Aodh*** (alarming service)
- ***Gnocchi*** (time-series database and resource indexing service)

Network layers (IM: **NFPNetworkLayer**) are based on ***Neutron***, a component which implements services and associated libraries (IM: **SoftwareLibrary**) to provide on-demand, scalable, and technology-agnostic network abstraction. In particular, Neutron gives cloud tenants an API to build rich networking topologies, and configure advanced network policies (IM: **Policy**) in the cloud. Examples include:

- create multi-tier web application topology

  Enables innovation plugins (open and closed source) that introduce advanced network capabilities

- use L2-in-L3 tunneling to avoid VLAN limits, provide end-to-end QoS guarantees, use monitoring protocols like NetFlow.

  Lets anyone build advanced network services (open and closed source) that plug into Openstack tenant networks.

- LB-aaS, VPN-aaS, firewall-aaS, IDS-aaS (not implemented), data-center-interconnect-aaS.

  provides an API Extensibility Framework, including extensions for “provider network”, which maps Neutron L2 networks to a specific VLAN in the physical data centre.

Security is handled by two crucial components: ***Keystone*** which provides authentication and authorization; ***Barbican*** which provides secure storage, provisioning and management of user (tenant) secrets such as passwords, encryption keys and X.509 Certificates. Data units are handled by ***Cinder*** (block storage) or ***Swift*** (object storage) when referring to files, blobs and any kind of block/object storage.
Based on the above, we can map operations as follows:

Guest control: create/destroy/migrate/power on/power off are all handled by Nova through a rich REST API. Nova in turn issues these operations to the underlying backend driver based on the hypervisor (Xen/KVM/Hyper-V etc.) or the bare-metal infrastructure.

Guest monitoring: Ceilometer gathers statistics about the status and usage of key resources of the system. it has a project granularity so accounting is only done at this level

Access control: for guests is handled by barbican for tenants (users) is handled by keystone

Connection and Session management: This is handled by the VM guest operating system.

Routing and switching : This is handled by neutron and its underlying components (or plugins). These include open vSwitch, various HW specific switch models, Linux Bridge, Modular Layer 2 (ml2), several implementations of commercial Network Virtualization Platforms and implementations of various OpenFlow Controllers.

The mapping to the Information Model would therefore be as follows; A **CloudAdministrator**, a particular **Administrator** can *modifyState* of **SystemVMs**. They can **VMPowerOn**, **VMPowerOff** a **SystemVM**. They can also *perform* **OperationMigrate** or *modifyState* to **PowerOn** or **PowerOff** via a **SoftwareAPI** that in turn uses **SoftwareDriver**. The **SoftwareDriver** *depends* on the type of **SystemHV**, whether it be

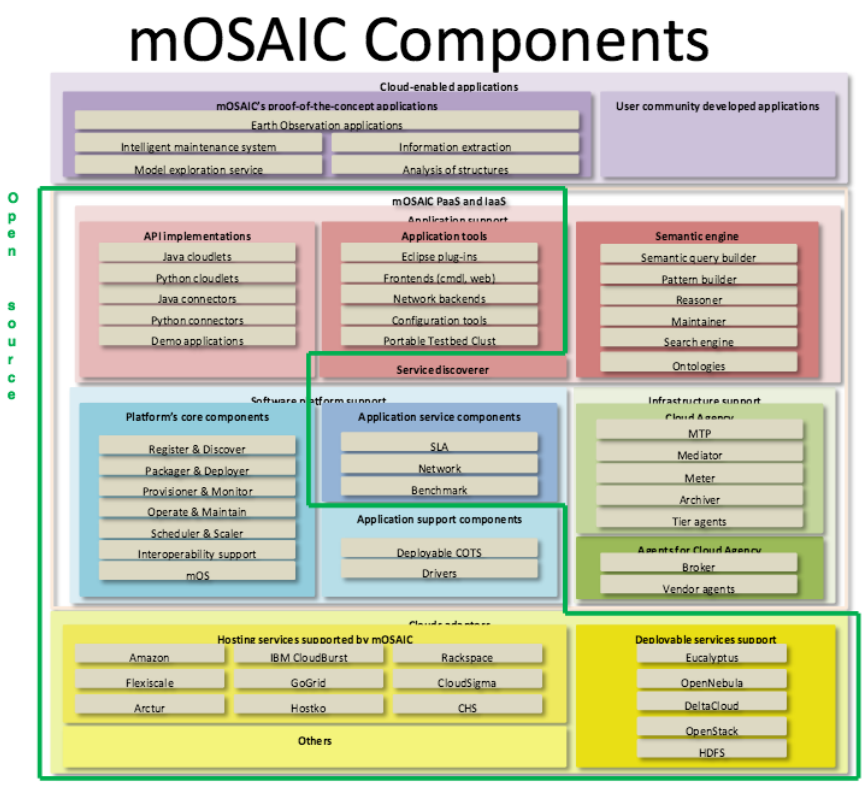


Figure 2.14: As captured in a concertation meeting, the mOSAIC ontology is not open source

SystemHVKVM, **SystemHVXEN** or just a **SystemServer**. **Operation Monitoring** of the **SystemHV** and **SystemVM** is performed by a **MonitorActivity**. **CloudAdministrator** 'Barbican' and 'Keystone' **perform Role Administrator** for **Actor** and **Role** via **Permission ACL**. The **Administrator** 'neutron' *manage* **Operations OperationRouting** and **OperationSwitching**. The **OperationRouting** and **OperationRouting** is **performed** on a **PhysicalSystem** or **LogicalSystem** that *perform* **Role RoleRouter** or **RoleSwitch**.

2.6 Alternatives to the Trilogy 2 Information Model

There are a large number of alternative information models and ontologies that are potentially relevant to the the work done in the Trilogy 2 Project.

One of the most useful alternatives is the mOSAIC ontology[33]. Unfortunately this is not open source (as is suggested in the diagram captured in Figure 2.14[4]). However the resource descriptors and high level description are captured in D1.2 of the mOSAIC deliverables, retrieved from InterCloudTestbed Project website[5].

Although the complete mOSAIC ontology is not publicly available, the designers methodology and the top-level concepts are captured in their public deliverable (see Figure 2.15). The presentation from the concertation meeting also suggests that most cloud brokerage and IaaS type platforms focus primarily on the Component, Protocol and Layer concepts discussed in mOSAIC, whereas NIST primarily focuses on Deployment_Model, CloudSystemVisibility, Essential_Characteristic and Service_Models.

The mOSAIC review cited the Unified Cloud Interface[1] as one of the earliest semantic Cloud ontologies. They also suggested that CCIF were working on an ontology that was not available in 2011. Checking the Google Groups page suggests that activity in this forum closed in 2012[6].

A literature review published in 2012[4] indicated that the most often cited work for Cloud ontologies was a work by Youseff et al. produced in 2008[55].

For higher level management systems, solutions have been proposed for inter-cloud management such as

[4] http://www.cloudwatchhub.eu/sites/default/files/MOSaiC_diMartino_Concertation-Meeting_12-13Mar2014.pdf

[5] http://www.intercloudtestbed.org/uploads/2/1/3/9/21396364/p2302-13-0012-00-drft-beniamino-di-m pdf

[6] https://groups.google.com/forum/#!forum/cloudforum checked at the time of writing (December 2015)

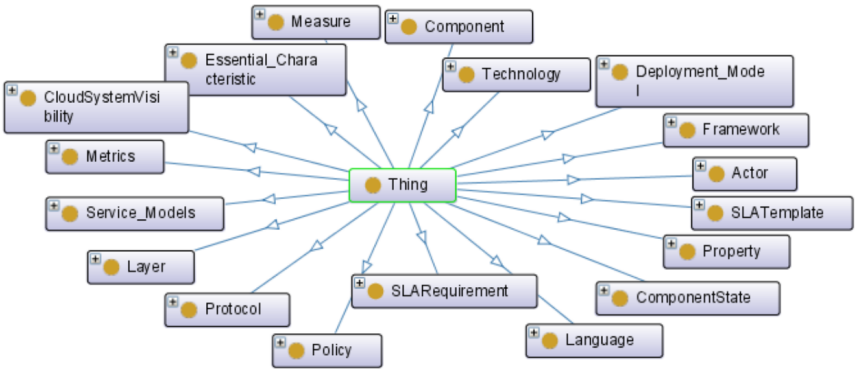


Figure 2.15: A screenshot from mOSAIC D1.2 document showing the top level concepts defined in mOSAIC ontology

FI-WARE's[7] Inter-Cloud Bridge system[46] and the InterCloud as a Service[47] (ICaaS).
SeaClouds[8] describes resources from a slightly different perspective where it looks at distributing multi-component services across distributed cloud resources.
The Oracle cloud resource model API [8] created in 2010. In particular the reference in Section 9 to the Cloud model is particularly relevant. (Now located in the guides as - [9])
The Common Information Models (CIM) is a standardised set of information model that are part of the schema generated by DMTF[15][10] In particular - CIM_System, CIM_Physical, CIM_Network, CIM_Device, CIM_Core, CIM_Policy The CIM provided by the DMTF is 'complex' and is 'more comprehensive' than the SPLUNK CIM[11] that is used by many enterprises to utilise machine data for the purposes of operational intelligence.
Yangui S.[54] identifies many of the same systems for provisioning on the cloud including;

- PaaSage
- mOSAIC
- Cloud4SOA
- Contrail
- Cloud-TM
- 4CaaSt
- RESERVOIR

System	SOA support	Portability	Standardised Model	Standardised API
PaaSage	Yes	No	No	No
mOSAIC	No	Yes	Yes	Yes
Cloud4SOA	Yes/No	Yes	Yes	Yes
Contrail	No	No	Yes	Yes
Cloud-TM	No	No	Yes	Yes
4CaaSt	No	No	Yes	Yes
RESERVOIR	No	No	Yes	Yes

[7] https://www.fiware.org
[8] http://www.oracle.com/technetwork/topics/cloud/oracle-cloud-resource-model-api-154279.pdf
[9] https://docs.oracle.com/html/E28814_01/resource.htm
[10] http://www.dmtf.org/standards/cim/cim_schema_v2441
[11] http://docs.splunk.com/Documentation/CIM/latest/User/Overview

For the Cloud resources description he identifies the following applicable descriptors/frameworks;

System	SOA support	Portability	Standardised Model	Standardised API
PaaSage	Yes	No	No	No
mOSAIC	No	Yes	Yes	Yes
Cloud4SOA	Yes/No	Yes	Yes	Yes
Contrail	No	No	Yes	Yes
Cloud-TM	No	No	Yes	Yes
4CaaSt	No	No	Yes	Yes
RESERVOIR	No	No	Yes	Yes
-	-	-	-	-
TOSCA	Yes	No	Yes	Yes
CAMP	Yes	No	Yes	Yes
OCCI	No	Yes	Yes	Yes

2.7 Conclusion

The Information Model is available from the website http://trilogy2.eu/wp-content/uploads/trilogy2-base.owl. It has only been used in the scope of Trilogy 2 specific domain concepts and as such should not be seen as a reference model without prior investigation and tailoring to the application specific needs.

The Trilogy 2 Information Model is based on a broad set of use-cases that investigate different aspects of resource Liquidity. The interests of the Trilogy 2 partners are diverse and investigate Storage, Compute and Network resource sharing from different approaches. The partners also take into account the interaction of resources at different levels of the network, looking into how operators and end-users can benefit from Liquidity. From these use-cases, the key concepts have been extracted and modelled using an iterative approach. Starting from a small subset of use-cases and building up by adding more concepts required that the model be re-designed to incorporate these concepts. The use-cases provide a non-exhaustive set of requirements. The motivation behind the model is not to re-invent the wheel in re-defining concepts that have already been well defined by different knowledge engineering communities but rather to work on how to link the various knowledge-bases together in a useful manner. Data models can be generated from an Information Model. We have highlighted a generalised Cloud Data Model through the OnApp Data Model. This could be mapped to other specific implementations such as OpenStack and we have described how some of these concepts may be mapped. To demonstrate the applicability of the Information Model to a set of use-cases we have taken three representative use-cases that cover different areas of the Trilogy 2 domain and shown how the use-cases can be described in terms of the Information Model.

Table 2.1: Properties in the Information Model

Property	Sub-properties	Metric	T2 Resource
CPU	• Arch (AMD/ ARM/ x86) • ISA (x86_64/ ARMv8b) • VT-x	• Floating point speed (GFlops) • Frequency (Hz) • Number of cores • IPC / IPS (instruction per clock / second)	Compute
Compute Load	• Activity (of a VM) • Requests • Efficiency • Throughput • Virtual CPU	• Requests per second • Overcommit ratio • bits per second	Compute
Location	• Coordinates (WGS84) • Timezone (UTC / SI) • IP address (IETF) • MAC Address (IEEE) • VLAN (IEEE)		Compute, Network
Network Layers	• Physical • Virtual Physical • Network • Transport • Application	• Protocol choice (MPTCP, ECMP, PVTCP)	Network
Virtualisation	• CPU Virtualisation • Storage virtualisation • Network virtualisation	• Hypervisor (Xen, KVM, HyperV, etc) • VLAN	Compute, Network, Storage
Security	• SMTCP • MPTLS • SSH • TLS • API Key • Data Encryption • HTTPCrypt		Compute, Network, Storage
Data unit	• Packet • Flow • Subflow • Stream • Stripe • Blob • File • Template	• Bits • Bytes • 5-tuple • latency • throughput	Network, Storage

3 User Control

In this section we present the tools for controlling liquidity developed during this third and last year of the Trilogy 2 project. We have pursued three main lines of work during this last year: end-to-end encryption, incentives for creating liquidity by the end user and controlling MPTCP traffic.

End-to-end encryption was presented in the Trilogy 2 architecture (deliverable D2.3) as one of the key mechanisms to allow the end users to control the extent in which the liquidity tools deployed by the ISPs (in particular NFV) can affect the user's traffic. As presented in D2.3, an operator deploying NFV is able to pool all its resources (in particular processing resources) to dynamically create network functions that affect the end user's traffic. While some of these network functions can be perceived as beneficial from the user's perspective, others functions may be not be so (an example of these not so beneficial functions include traffic shaping for specific applications using Deep Packet Inspection (DPI) techniques). We argue that end to end encryption is a mechanism to allow the end user to somehow control how the deployed network functions will affect its traffic. In the particular example presented before, by encrypting the traffic, the user decides what traffic is exposed to the ISP and therefore inspected by the DPI engine. In deliverable D2.4 we presented two protocols for providing end to end encryption with MPTCP. In this section we first present a feasibility study of the deployment of such solutions in the real Internet. We see that while it is feasible to deploy such solutions with a low number of errors in general, it is not the case for web traffic carried in port 80. In order to deal with that specific limitation and to overcome other issues with current TLS based solutions for securing web traffic, we then present HTTPCrypt, a novel solution for opportunistic encryption of web traffic, aligned with the work carried out on TCPINC, described in Deliverable D2.3.

Incentives are another important tool to provide control of liquidity. If the incentives are properly set, users will generally take the incentivised course and overall the system will be manageable and behave as expected. In this section we next present Kadupul, a mechanism to create the proper incentives for end user to create network liquidity. We argue that especially in the fringes of the network, there are large pools of resources that are underutilized because lack on incentives from the users controlling these network pools to offer them to other users. In particular, given current pervasive wireless technologies, there is a large number of direct paths between users using wifi/bluetooth or other wireless technologies. These network pools are unexploited because users have no incentives to make them available to other. Kadupul is a tool that uses Bitcoin based micropayments to create the incentives to users to offer their network resources to others to forward traffic.

Finally, we present the MPTCP subflow controller. MPTCP is one of the key liquidity tools of the Trilogy 2 project. A key aspect for the creating of networking liquidity with MPTCP is the creating of subflows through different paths. The subflow controller described in this section allows the user to control the creation of liquidity by determining which subflows to create. Different strategies are presented as well as associated use cases for each of them.

3.1 MPTCP security feasibility measurement

3.1.1 Deployability analysis of the proposed MPTCP security extensions

In deliverable 2.3 we presented pervasive encryption as a fundamental mechanisms to empower endpoints and allowing them to regain control of their communications. We proposed two alternative designs to secure MPTCP, both of which rely on encrypting the data exchanged. In this section, we investigate the deployability of such solutions in the real Internet.

Modern networks often rely on dedicated hardware components generically dubbed *middleboxes* to perform advanced processing functions like, for example, enhancing application performance, traffic shaping, optimizing the usage of IPv4 address space or security. One major criticism of middleboxes is that they might filter traffic that does not conform to expected behaviors, thus ossifying the Internet and rendering it as a hostile environment for innovation [23].

This does not mean that it is impossible to deploy new protocols, but that in order to ensure success it is imperative to first understand the interaction of the proposed solutions with the middleboxes active along the path.

In this section, we aim to understand the feasibility of the deployment of secure flavors of MPTCP by using a novel methodology for performing large scale Internet measurements, using a crowdsourcing solution approach. The emerging sea of crowdsourcing (such as the Amazon Mechanical Turk, Microworkers and

others) can provide an accessible alternative to perform large scale Internet measurements. By expanding the traditional crowdsourcing focus from the human element to use a diverse and numerous group of end-user devices as measurement vantage points [16] we can leverage on crowdsourcing platforms to run Internet wide measurements.
In order to assess the reaction of deployed middleboxes to secure version of MPTCP, we measure the success rate when we try to establish TLS connections encryption in traditionally unsecured ports. In particular, we attempt to initiate TLS connections in 68 different ports that normally do not use any form of encryption and analyze the success of the connection. This is a first necessary step towards a full comprehension of the behavior of middleboxes relative to secure MPTCP deployment.

3.1.2 Methodology

In this section, we describe the measurements methodology we employ to assess the potential success of deploying secure version of MPTCP in the Internet using crowdsourcing. We try to establish TLS connections from a large number of vantage points (from now on, *measurement agents (MAs)*) to a large number of ports, which traditionally do not use TLS in a target server (from now on, *measurement server (MS)*), using crowdsourcing based measurements. In order to mimic the behavior of an encryption protocol in the Internet, we try to establish TLS connections from a large number of vantage points (from now on, *measurement agents*) to a large number of ports in a target server (from now on, *measurement server*) which we control. Crowdsourcing platforms connect *employers* and *workers* from around the world. The employer is the one who creates the task (or the *"micro-job"*) for workers and specifies the parameters of its campaign, e.g., the size of the set of users performing the task or their geographical location at country level.
We argue that this approach can become an important tool for evaluating innovation solutions, primarily due to the large number of accessible and diverse measurement vantage points. Additionally, we can benefit from the freedom of deploying our own custom-designed measurement tests.
We recruit users through the Microworkers crowdsourcing platform to complete measurements on the feasibility of pervasive encryption in the current Internet ecosystem. To capture how effective would secure MPTCP actually be if deployed in today's Internet, we collect and analyze the results from more than 2,000 MAs that try to establish TLS connections in a large number of ports which normally do not use TLS.
The MAs attempt to establish both HTTP and TLS connections to 68 different ports, namely 10 well-known ports, 56 registered ports and 2 ephemeral ports. We use the success rate of the HTTP connections as the benchmark against which we compare the number of TLS successful connections. We establish then the success rate of the TLS connection by contrasting the result against the status of an unencrypted HTTP connection established in the same port. We also store and analyze in detail the server side packet exchanges.
The procedure is as follows. First, we start by asking the user to connect using a HTTP connection in port 80 to a webpage we provide. Meanwhile, in the background, HTTP and HTTPS connections are performed from the measurement devices to our servers in all the other 67 ports. In this case, data about the performance are collected in the MS.
Second, the webpage we provide contains a short form asking for additional input about the type of Internet access they are using. Finally, on the server side, we also collect and store metadata on each of the MAs that connect to our servers, such as the IP address, the user agent type, the language, and any other information included in the HTTP header.

3.1.3 Results

In the campaigns for fixed lines, we recruit 1,165 *workers* from 53 different countries. The MAs are hosted in 286 ASes overall.
For the mobile case study, we recruit 956 *workers*, from 45 different countries and 183 *ASes*.
Considering that each MA performs 68 connections to our MS, we build a complex dataset for a total of 114,228 connections [1].
Table 3.1 refers to the results obtained from aggregated results, for both fixed line and mobile network (label *Aggregated*), the results from users that use a fixed line and from users connected to a cellular network (labels *Fixed*, *Mobile*). We also compare the rate of successful TLS connections for users we detect using a proxy and for users that do not (labels *Proxy*, *Non-proxy*) in mobile network scenario. To better understand how proxies or other middleboxes behavior impacts the performance of the TLS protocol in unconventional ports,

[1] The data set is freely available on *http://it.uc3m.es/amandala/dataset.php*

Table 3.1: Results from aggregated results, fixed line and mobile network

Analysis	Port 80 (%)	Average other ports (%)
Aggregated	16,5	5,8
Fixed	9	6,95
Mobile	25	4,54
Proxy	70	4,23
Non-proxy	10	6,8

Table 3.2: Packets analysis, percentage of received SYN

Analysis	Fixed Line		Mobile	
	SYN(%)	NO SYN(%)	SYN(%)	NO SYN(%)
All	96,8	3,2	36	64
Port 80	88,3	11,7	27,7	72,3
Proxy			22,2	77,8
Non-proxy			12,7	87,3
Proxy (80)			9,6	90,4
Non-proxy (80)			36,4	63,6

we focus on the packet analysis, splitting the analysis for fixed line, mobile and for users that use or not a proxy.

Table 3.2 refers to the percentage of SYN we receive when users try to establish a TLS connection to our MS from fixed line use case and from mobile network, considering all port and particularizing the analysis for port 80 (labels *All*, *Port 80*). Moreover, in the case of mobile network we particularize the analysis for proxy/non-proxy case (labels *Proxy*, *Non-proxy*, *Proxy (80)*, *Non-proxy (80)*).

We observe that in the case of proxies, 90% of the SYN packets are missing. While this may seem non causal at first (as the SYN packet is forwarded before the middle box actually knows whether this is a regular HTTP connection or a TLS connection), proxies usually wait until they receive the GET from the client to establish the connection to the server in order to apply their policies. This explains why in the case of TLS, we miss a high number of SYN packets.

We also try to understand if the filtering of TLS is consistent across the different ports for a given MA. In other words, if the TLS connection fails in a given port, how likely is that it will fail in other ports. In order to quantify this, we estimate the conditional probability of failure in a given port X given that the TLS connection in port 80 has failed. We choose the port 80 as it is in general a port with high failure rate. We estimate the aforementioned conditional probability for the case of fixed line and for the case of mobile network without proxies. We can see that the estimated conditional probability is around 90% in both cases (slightly higher in the fixed line case), implying that when the TLS connection fails in port 80, it is very likely that it will fail in the other ports.

3.1.4 Discussion

We find that in average the failure rate of TLS over different ports is near the 6%. We also find that in the case of mobile networks where proxies are used, the failure rate can be as high as 70%. We conclude that it is probably feasible to roll out TLS protection for most ports except for port 80, assuming a low failure rate (6%).

We believe that our results can serve as a lower bound for the failure rate for using protocols other than expected in different ports.

3.2 HTTPCrypt - Low Latency Opportunistic Encryption

We introduce HTTPCrypt, a scheme of opportunistic encryption for the plain HTTP protocol designed to build web services compatible with the existing clients, servers and proxies ecosystem. HTTPCrypt uses state-of-art cryptographic primitives to provide both high performance throughput and low latency connection establishment, as well as complete interoperability with any middlebox that supports plain HTTP traffic. Unlike other opportunistic encryption schemes, HTTPCrypt defines procedures to establish name-based authenticity of a server, even in the case of networks with restrictive access policies. We evaluated the practicality of deploying HTTPCrypt by integrating it into a popular HTTP server stack and evaluating it against alternative opportunistic encryption schemes.

Transport Layer Security (TLS) has long been the standard choice for HTTP data encryption (to prevent passive snooping) and the validation of peer identities (to prevent active attacks). Clients connections are secured by first establishing a TLS connection to the endpoint, and then issuing standard HTTP requests

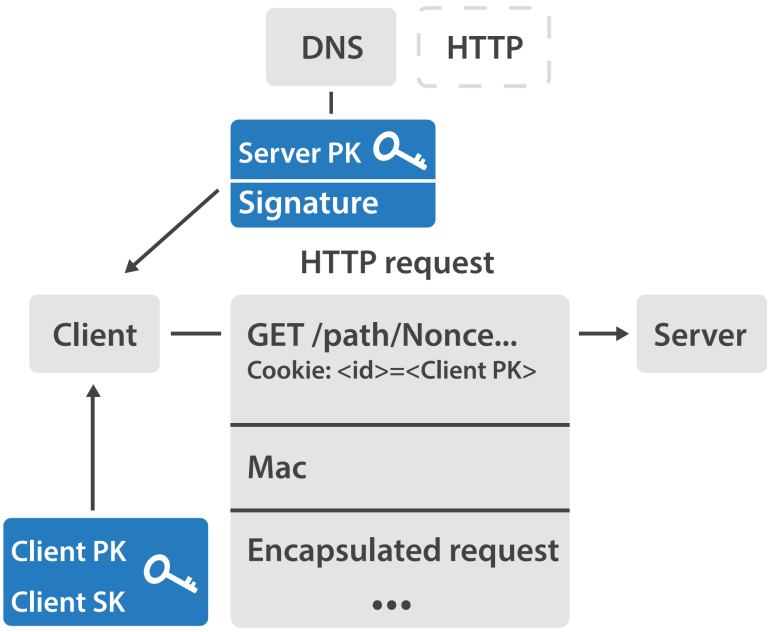


Figure 3.1: The overall structure of an HTTPCrypt connection. The server public key can be retrieved out-of-band via DNS, or directly over HTTP

across TLS tunnel. This separation has significantly increased the latency of encrypted web browsing due to the multiple network round-trips required, and motivated the proposal of TLS protocol extensions to optimise the process.
The introduction of the HTTP/2.0 standardisation process, the first major HTTP protocol revision since 1999, has brought up the possibility of supporting opportunistic encryption for all web services, including those connections whose identities cannot be verified due to server misconfiguration, self-signed certificates, or expired signatures. All such proposals have involved upgrading the connection to TLS, thus maintaining the dependency to the complex suite of libraries that implement the full TLS protocol.
We introduce HTTPCrypt: an alternative application level HTTP extension that enables HTTP payload encryption with the following desirable properties:

- Transparency for existing HTTP caches and proxies, meaning that it is expensive to distinguish HTTPCrypt requests from plain HTTP requests;
- No significant latency increase of opportunistically encrypted connections vs plain HTTP;
- Removes the dependency on TLS and replaces it with cryptography suitable for embedded devices;
- Optional end-to-end identity checking;
- Low performance overhead for server software, including support for using the same efficient kernel system calls to transfer large files.
- Easy migration to the encrypted connections for the existing software clude by evaluating our implementation of HTTPCrypt vs a popular application stack.

3.2.1 Design Goals

HTTPCrypt is designed to work with the HTTP protocol and its many quirks, and avoids forcing a dependency on TLS as the sole method of opportunistic encryption. We will next explain our approach to HTTP compatibility, the impact on performance and latency, and finally how our approach is easy to embed.

HTTP Compatibility

The HTTP protocol has explicitly supported proxying since its inception, and the influence of middleboxes can no longer be ignored when updating protocols. Middleboxes have hampered the adoption of many new Internet protocols; from full transport stacks such as SCTP or CurveCP, to extensions to existing protocols such as TCPCrypt or MPTCP. It is also common to encounter monitored gateways that provide Internet access purely through HTTP proxies that actively intercept or block HTTPS traffic.

The logical choice to avoid the influence of middleboxes while adding opportunistic encryption is to make HTTP requests that are difficult to distinguish from ordinary requests. In particular, the Cookie HTTP header is well suited to carrying cryptographic data, as it usually contains an encrypted payload that is indistinguishable from random data. Request URL is another suitable place for putting cryptographic data by the same reason as cookies.

Another main difference with the use of opportunistic encryption vs a fully established secure transport is that peer identities need not be fully verified. HTTPCrypt supports establishing the remote peer's identity via side channels such as the local DNS service, and we explain later why this is a reasonable approach in the modern Internet. The basic scheme of an HTTPCrypt request is depicted in Figure 3.1. A client obtains the server public key out-of-band, and makes a normal HTTP request with the session key contained in the cookie. The encrypted contents then follow containing nonce and authentication tag.

In plain HTTP protocol, there is no payload within GET requests. However, for HTTPCrypt any request requires encrypted content. The first way to solve this incompatibility is to use POST requests for all HTTP messages. Another option is to encode the complete encrypted HTTP request within the request URL. In this case, the limit of HTTP request that could be encoded is about 2K due to URL length restriction (that is 2083 bytes). Since encrypted payload contains a nonce, all request URLs will be unique preventing thus caching on proxies. HTTP pragmas could also reduce chances to be cached by middleboxes for HTTPCrypt request but this does not matter for the protocol itself merely helping to reduce unnecessary caching by intermediate proxies. HTTPCrypt is compatible with all the major HTTP transfer encodings, such as chunked encoding, and can maintain keep-alive connections just as normal HTTP does. Moreover, HTTPCrypt natively supports name based virtual hosts without the need for protocol extensions such as the TLS Server Name Indication.

Unlike TLS, where there is a ciphersuites agreement phase, we claim that it can be skipped for the vast majority of REST based HTTP services: in case of ciphersuite migration, it is possible to specify a new scheme explicitly, for example by creating a new set of access URL's or even by creating a dedicated host for the new ciphers. Moreover, the current encryption scheme used by a specific server could be explicitly stated in the DNS.

3.2.2 Latency and performance

HTTPCrypt requests follow the HTTP model of starting without any preliminary handshake phases. The client is responsible for obtaining the server public key, and the only extra information that needs to be passed to a client connection is the client's own public key. The client can therefore encrypt an HTTP request immediately using its own private key and the server's public key, and assume that the server can decrypt the request using its own private key and the client's public key. One drawback of this approach is the vulnerability to replay attacks, since a server cannot send its own random data before the initial client request. We discuss later how developers can mitigate this problem at the application level to prevent replaying the whole session. The initial request can always still be replayed, but this is not a security flaw if mitigated at the application level or if used to access read-only data.

HTTPCrypt is designed to establish HTTP sessions by skipping intermediate phases of key exchange or capability agreement for a connection. The disadvantage of this approach is that it reduces the flexibility of the connections, but it is also simpler and prevents downgrade attacks, such as one recent TLS vulnerability.

The performance and latency benefits from the above simplifications are significant, and make this is a viable approach for widespread implementation of opportunistic encryption. The data passed over HTTPCrypt is encrypted in-place, with a small authentication tag for a data chunk placed prior to each the encrypted payload. This allows using of multi-buffer kernel system calls such as writev to avoid data copying and improve performance. It also permits implementations to transfer arbitrary sized chunks of payload to optimize the connection utilization, and not be limited to a window of the maximum intermediate buffer size. It is very common to encounter many short requests in HTTP, particularly when dealing with proxies that do not fully support HTTP/1.1 keep-alive. Opportunistic encryption schemes that require TLS handshakes are costly both

in terms of connection setup latency and the request rate. In contrast, HTTPCrypt skips the full handshake-term connections, thus supporting the asynchronous requests to advertising networks or loggins counters that are commonplace in modern websites.

3.2.3 Integration with the existing code

The TLS protocol stack is very complex, and even embedded implementations contain tens of thousands of lines of code. One reason for the code bloat is that all TLS implementations have to implement multiple cipher suites, key exchange and signing schemes for the purposes of backwards compatibility. Rather than propagate this complexity into opportunistic encryption (which we want deployed as widely as possible), HTTPCrypt specialises its encryption to be based on the NaCL cryptobox primitives. Cryptobox can be implemented using both a high-performance profile or as a small library within about a thousand lines of portable code. The design of HTTPCrypt proposes the minimal changes to the existing applications: just use any suitable cryptobox library (choosing either performance or code size), get any HTTP parser library and encrypt the payload using cryptobox construction. Unlike TLS, HTTPCrypt does not interfere with the IO processing logic nor require some intermediate buffering.

3.2.4 Protocol Description

We now describe describe the architecture of HTTPCrypt in detail, via the handshake procedure, request structure, cryptographic primitives and session resumption.

Handshake Procedure

The client initially obtains and checks the ephemeral public key of a server. Since we are dealing with opportunistic encryption, it is not necessary to protect this phase against active attacks. Methods of retrieving the public key include:

- Obtain an ephemeral public key from the corresponding DNS record, and check the authenticity of this reply using DNSSEC or DNSCurve.
- Obtain a DNS record with the current ephemeral public key as a SPKI or x.509 certificate using, for example EdDSA signature scheme to fit the whole certificate in a DNS record (which is typically allowed to be not larger than 512 bytes for many constrained networks).
- Perform a plain-text HTTP OPTIONS request to the target HTTP server and obtain the current certificate.

The Domain Name System is the preferred way to obtain the ephemeral public keys since DNS is used to resolve names and to verify domains ownership. If DNSSEC or DNSCurve are enabled for name resolution, it is also possible to validate the published key via DNS without PKI system. On the other hand, DNSCurve breaks intermediate DNS caching and DNSSEC can significantly increase requests latencies for clients as each name must be validated using multiple requests for each name component when using DNSSEC. DNS caching could reduce the negative effect of validation but it is still required to ask a caching server for each name component for a client. Moreover, DNSSEC and DNSSCurve can be filtered in restrictive networks, where plain DNS requests are the last resort available for the clients. Therefore, HTTPCrypt enabled services should publish the full PKI certificate in the DNS record to be compatible with all clients. Using of the modern signature schemes, such as EdDSA, allows storing a full x509 certificate within 512 bytes that is typical limit for a DNS record.
Storing ephemeral keys in the DNS defines their expiration time as equal to the DNS TTL value. HTTPCrypt defines these methods to store ephemeral keys in DNS:

- DANE TLSA records that contain the full ephemeral key signed by a trusted key within x.509 or SPKI certificate;
- TXT records can carry the keys and signatures encoded with Base64 encoding;
- AAAA or A resource records can also encode keys and signatures (described below)
- AAAA record tunnelling

Not all types of DNS records are allowed by some network policies; for instance, a common practice is to filter all unknown DNS resource records. TLSA records have been proposed relatively recently, and so are treated as unknown by many DNS recursive forwarders. TXT queries are forbidden in some networks since such records are frequently used to construct IP-over-DNS tunnels. This restriction also denies DNSCurve requests that use the TXT compatibility mode for their encrypted payloads.
In such a constrained network, HTTPCrypt can use IPv6 AAAA records to encode public key material. We store the relative order of the key material record inside the first 16 bits of the address and use the remaining 120 bits to encapsulate the payload in network-endian order. The first 12 bits are used to define IPv6 addresses. The order field is required to reconstruct the payload when a DNS server performs round-robin rotation of resource records. With 4 bits of order available, it is possible to encode up to 16 addresses with 112 bits of useful payload. Therefore, to encode a 256-bit key and 512-bit signature, we need 8 IPv6 addresses: three for the public key and five for the signature. An additional address can be used to publish when this signature has been issued and the validity interval.

3.2.5 Deriving a session key

After obtaining the server's public key, a client generates (or reuses) its own key pair and derives the shared secret using the curve25519 scalar multiplication procedure on the server's public key and its local private key. An additional round of pseudo-random permutation then produces the final session key. The resulting session key is subsequently used to encrypt and authenticate the HTTP payload. The authentication tag only covers the encrypted payload, since middleboxes can modify HTTP headers or the structure of a plain HTTP request. In contrast, the HTTP payload is not altered by proxies.
After receiving the HTTPCrypt request from a client, a server derives the shared secret using its local secret key and the client's public key from the Cookie HTTP header. The cookie also contains the ID of the server's public key. All further requests down the TCP connection (including HTTP Keep-Alive sessions) are encrypted using the established shared key. The keep-alive timeout must thus be less than the ephemeral key lifetime to guarantee forward secrecy for all long term connections.

3.2.6 Request structure

HTTPCrypt requests are designed to be difficult to distinguish from plain HTTP requests. Therefore, HTTPCrypt uses the HTTP Cookie header to provide the public key of a client to the server. The structure of this header is: Cookie: IDsk||pad=Kclient ||pad
All the values are encoded with base64 that is commonly used in HTTP cookies. The padding prevents middleboxes from detecting the specific length of the public key and ID (if desired). Paddings should have unpredictable length and content for that purposes. The server key ID field applies a SHA3 cryptographic hash function to the server's public key and takes the first 10 bytes of its output: ID = take(10, PRF(pubkey)) The server uses this ID when rotating ephemeral keys to select the correct key for a particular connection. Two ID collision probability is 1/240 which is suitable for the purpose of public keys distinguishing on a server's side in conjunction with Host header.
In HTTPCrypt, the encrypted payload encapsulates the real HTTP request, but some HTTP headers are used from the unencrypted part, for instance the:

- Host is used to distinguish name-based virtual hosts and is required if those hosts have different public keys. Alternatively, the HTTP server may distinguish hosts by the IDsk value and ignore this header.
- User-Agent header may be checked by middleboxes.
- Content-Length, Content-Range headers are used by the HTTP session as-is.
- Pragma, Expires and Cache-Control prevents proxies from caching encrypted requests.

While most of the HTTP headers could be sent unencrypted to save computational resources, some headers such as Cookie or Referer must remain encrypted (the former because its use is overloaded by the HTTPCrypt request structure). We do not define the exact list of headers that must be encrypted as this depends on the applications' architecture. If the payload is placed within HTTP body, HTTPCrypt uses the unencrypted URL to store the initial nonce for the request. We propose to use the first path component after the HTTP path as nonce whilst the initial path could be used by a web server to find HTTPCrypt protected resource.

In the case when request is completely encoded within URL, the encrypted payload is placed (base64 encoded) into the query parameters component of URL.

3.2.7 Chunked encoding

In HTTP standard, chunked encoding must be supported by any client, therefore any proxy can split the original request into chunks. Hence, HTTPCrypt provides support to transfer chunked HTTP messages securely by means of the internal encrypted chunks. To distinguish between chunked and plain encoding, HTTPCrypt uses the special query ?chunked=1 to tell a client that the request is split to a set of encrypted chunks. The traditional Content-Transfer-Encoding header could not be used since it can be altered by middleboxes (for example, when a proxy performs gzip encoding). However, encrypted chunks support requires more complicated procedure to decode, so it might be avoided when implementing simple services. In this case, a client always reconstruct a server's reply as a single chunk; even if the plain request contains chunks they are merged into a single buffer for decryption and verification.
When HTTP chunked encoding is used, each chunk has its own nonce, MAC tag and additional length of the encrypted content. The second length specifies the size of data that is authenticated by this MAC with the specific nonce. It is needed when a middlebox or a proxy performs chunks resegmentation. In this case, a client will still be able to restore the original chunks sent by a server, decrypt and authenticate them properly. To avoid reordering of chunks, the subsequent nonces after the first one should be incremented as counter. Additionally, each authentication tag should be calculated using the previous authentication tag in addition to the current encrypted chunk content as following: M(i) = Mn(Mn1 (i1) ||P) where P is the current encrypted payload and n is the current nonce value. In this scheme, an attacker cannot remove any chunk of data from a message: if he or she removes the first chunk, then MAC won't be valid for the subsequent chunk as the original MAC is calculated using the first chunk MAC. The same logic is valid for all subsequent chunks. Even in the case if chunks have the same content, the fact that they are encrypted using different nonces and fact that the MAC tag is calculated over the encrypted payload (encrypt-the-nmac method) provide negligible chances that MAC tags for these equal chunks will be the same. Nonetheless, the last chunk is special. In plain HTTP, the last chunk has always zero length. However, re-using of this plain scheme opens a security breach: an attacker can simply remove some chunks from the end of a message adding unencrypted zero-length chunk. Therefore, HTTPCrypt defines a special encrypted zero-chunk which contains a next nonce, zero encrypted length and authentication tag calculated as following: M = Mn+1(Mn(i1) ||0)
After this special tag, all following payload could be used for compatibility with HTTP (namely, the last zero chunk) and should be ignored by HTTPCrypt protocol.
To reconstruct the final HTTP message HTTPCrypt defines the following procedure:

(i) Parse the original request and append all unencrypted headers except for the Cookie header.

(ii) For each encrypted chunk or the whole message increment the previous nonce, read the authentication tag and verify the encrypted content using the previous MAC tag as additional data for authentication (for the second and subsequent chunks). If the verification succeeds, decrypt the contents and parse the encapsulated request.

The request URI in the encapsulated request replaces the original URI and all encrypted headers replace the corresponding unencrypted headers. If MAC verification fails for any chunk, the peer sends an HTTP 500 error code inside the encrypted payload to prevent connection termination by an adversary who can inject arbitrary HTTP messages to the peers.

3.2.8 Cryptographic primitives

HTTPCrypt encryption is based on the Cryptobox construction introduced by Daniel Bernstein in the NaCL cryptographic library. This construction defines both secrecy and integrity for the messages based on public key cryptography, and is a conjunction of (i) public key exchange procedure; (ii) a stream cipher; and a (iii) one-time authentication algorithm.
The wire format of Cryptobox specifies a cryptographic nonce, an authentication tag (MAC) and the encrypted payload. Nonces for the first message in an HTTP session is generated randomly by both a client and a server. Nonces length is chosen to be long enough to make the probability of repetition negligible. Bernstein defines a nonce length of 24 bytes which is 2192 of possible nonces values and the probability of nonces collision as low as at least 1/2128. For the subsequent chunks or messages within the same keepalive

HTTP session, client and server should monotonically increase their initial nonces in counter like matter. Switching to counters allows to skip nonces in all but the first chunks.

Algorithms selection

The original NaCL implementation suggests the following components to be used in the cryptobox:

- curve25519 and hsalsa20 as the public key exchange operation
- xsalsa20 for symmetric encryption
- poly1305 for message authentication

At present, the salsa cipher family is superseded by chacha ciphers that have the same internal architecture but are optimised for the modern hardware and vectorized operations. We later evaluate the chacha cipher vs other state-of-art ciphers such as AES. The most significant advantage of chacha is good performance on the commodity hardware, including embedded systems based on ARM or MIPS processors as well as x86. Hence, our final selection of symmetric encryption algorithm is the chacha20 stream cipher. Moreover, the hsalsa and xsalsa ciphers are replaced with the corresponding chacha variants.

Session Resumption

Since the generation of public key shared secrets is an expensive procedure, HTTPCrypt defines several approaches to skip it for already establishes sessions. HTTPCrypt uses the same common principles that are defined in TLS by means of a session cache and ticket mechanism.

Session Cache

The session cache stores some of the client's state on the server side to avoid recalculating it. The state in HTTPCrypt includes the hashes of the client's and server's ephemeral public keys, and the resulting shared secret. When re-establishing a session, a server finds the cached state based on the public keys fingerprints and reuses the shared secret instead of regenerating it using public key cryptography. Session expiration is based on the server's ephemeral public key lifetime, and a least-recent-use expiration strategy if there are more clients than session cache space available. The server must destroy sessions associated with an expired ephemeral key in order to ensure that forward secrecy is preserved.

Sessions tickets

Session tickets are a mechanism for an HTTPCrypt server to support session resumption without having to store per-client state. This method implies that a client supports and enables tickets when resuming an HTTPCrypt connection. When using session tickets, the shared state is encrypted by a symmetric secret key known only by the server and subsequently passed to the client. The client can then reconnect by including a previously obtained session ticket in the initial handshake. The server decrypts and verifies this ticket and restores the session without an additional key exchange procedure being required. Session tickets in HTTPCrypt have the same structure and definition as in TLS with the only difference that an HTTP header is used to store and pass a session ticket.

3.2.9 Security Analysis

We now elaborate on the HTTPCrypt protocol's security properties, support for forward secrecy and denial of service resistance. We define the threat model for HTTPCrypt as follows:

- HTTPCrypt should be resistant to passive, active and denial-of-service attacks;
- HTTPCrypt should provide both secrecy and integrity for transmitted payload;
- There should be no easy way to distinguish HTTPCrypt requests from plain HTTP traffic.

HTTPCrypt Security Model

To defend against active attacks such as Man-in-the-Middle, HTTPCrypt uses the traditional model of peer validation using public key signatures and trusted 3rd party authorities. When using DNS-based signatures the authorities are defined as trusted DNS anchors, while in the case of a certificate chain HTTPCrypt uses the traditional PKI model where a peer's key can be signed by any trusted authority.
HTTPCrypt recommends the use of DNS chains of trust granted by means of DNSSEC or DNSCurve anchors. Nevertheless, for embedded appliances or difficult-to-change infrastructure the cost of a complete DNSSEC

validation might be too expensive. Furthermore, the cryptographic algorithms and standards recommended by DNSSEC (e.g. 1024-bit RSA), are more expensive than state-of-art cryptography.
In contrast, DNSCurve provides secure and efficient cryptographic primitives but is not widely deployed in the Internet for various reasons both historical and more practical since DNSCurve does not interoperate with intermediate DNS caching. Therefore, the traditional PKI model based on EdDSA signatures is a reasonable fall-back choice for HTTPCrypt validation.

Replay Protection

Protecting against replay attacks is more complicated than in HTTPS, since HTTPCrypt does not require a server handshake with a random cookie provided by the server. In HTTPCrypt, the first request can always be replayed by an adversary for the duration of the server's ephemeral key it is purely client initiated. It is possible (and recommended) to implement replay protection at the application level, for example by providing a unique authentication token from server to a client before granting access to the restricted area. However, a session before the first server reply is not protected against replay attacks.
To protect the subsequent session, the server places the random cookie in the encrypted and authenticated reply, for example inside a predetermined HTTP header. If the first request sent over HTTPCrypt is limited to an idempotent GET method, an adversary can capture and replay the first request, but will not be able to gain any advantage since the server's reply then includes a random element. Therefore, an attacker cannot replay any subsequent messages within a session, in particular side-effecting operations such as HTTP POST or DELETE methods. Here is the practical example of replay protection applied to an HTTPCrypt session. Initially, a client sends a GET request that contains a client's random cookie within the header inside the encrypted payload to provide protection from replayed server's messages: Client-Random: <24-bytes-of-random-data >
A server, in turn, generates the full authentication token (that should be long enough to make repetition probability negligible) by appending its own random cookie and pushes it inside the encrypted header: Random: <client-random><server-random>
This model provides effective replay attacks protection. For example, if an adversary can repeat the server's replies, then a client will not be able to match its own random part. Similarly, a server will fail to verify its own cookie and will drop the replayed requests if the client is replayed. Placing random cookies inside the authenticated and encrypted payload prevents an adversary from both observing or modifying the tokens.

Forward Secrecy

HTTPCrypt uses slowly rotating ephemeral public keys for HTTP servers to provide forward secrecy. Clients generate a new keypair for each unique HTTPCrypt session. If DNS is used to store and validate public keys, the rotation of ephemeral servers keys is implicitly defined by the DNS time-to-live (TTL) property used as the lifetime value for the ephemeral server's key. To avoid time synchronisation issues, servers should generate new ephemeral keys at each period of time equal to the DNS resource record's TTL value, and publish keys material to the DNS server.
HTTPCrypt does not mandate an exact procedure for updating the DNS, since any of the standard methods used all serve; e.g. AXFR, an LDAP directory or by executing scripts via SSH. Servers just need to be able to store ephemeral keys for the time equal to two DNS TTL values to be able to interact with the clients that have previous keys cached in some DNS cache Hence, the real ephemeral key lifetime is two DNS TTL periods. The servers should destroy the keys from persistent storage once they have expired.

Denial-of-Service Protection

Availability is an important property of HTTPCrypt if it is to achieve wide deployment. Unlike TLS, the HTTPCrypt server computes the shared key one the first stage of a connection. This operation is expensive in terms of CPU resources, whereas in TLS all computationally complex procedures are performed at the later stages of the handshake (starting from the second message received from a client).
At first glance, this is a disadvantage of the HTTPCrypt design. However, we observe that the TCP three-way handshake protects a HTTPCrypt server from promoting spoofed requests, to the established state. On the other hand, if an adversary is able to establish a valid TCP connection (for instance, via a distributed botnet) then there are no obstacles to continue to the additional stages of a TLS negotiation and force the server to execute CPU expensive computations. Therefore, HTTPCrypt is no more vulnerable to denial-of-service attacks than HTTP+TLS. Moreover, since the random response cookies are signed by the server in TLS, it

requires more resources to perform signing and shared secret generation than the HTTPCrypt mechanism of merely generating a shared secret and encrypting the cookie. HTTPCrypt could also upgrade its scope to include cryptographic puzzles as part of its handshake, for example as defined in the MinimalT protocol. The concrete definition and evaluation of crypto puzzles are beyond the scope of this work.

Discussion

We now describe the implementation peculiarities used by our HTTPCrypt prototype, beginning with low-level cryptographic optimisations, operating system acceleration, and integration with existing application stacks.

Cryptobox Optimizations

As a default ciphers suite HTTPCrypt proposes chacha20 as stream cipher and pseudo-random function, poly1305 as one time authenticator, and curve25519 as key exchange function. However, these primitives could be easily switched to another ones (for example, further we demonstrate openssl cryptography used by HTTPCrypt). In our experiments, we use optimized versions of chacha20-poly1305 and optimized version of curve25519 ECDH implementation derived from Sandy2x. Both bulk encryption and key exchange algorithms benefit from AVX instructions set implemented in Intel processors.

We have also extended the original NaCL cryptobox primitive to allow vectorization of the encryption, for example, to encrypt headers and body in the same call to the API. This change still allows to encrypt and authenticate data in-place and avoid intermediate buffering.

3.2.10 Operating System Optimizations

Contemporary operating systems provide various high performance systems calls to optimise the I/O handling for serving HTTP requests with a high throughput. For example, an HTTP server running on Linux or FreeBSD can utilise the sendfile system call to transfer a file to a socket directly via the kernel without requiring intermediate copying through user-space buffers. Some variants of this system call (e.g. the FreeBSD sendfile) also accept arbitrary prefixes as arguments. Unlike TLS that requires intermediate buffering, HTTPCrypt can use the semantic of sendfile call to send files encrypted without requiring copying through userspace. To support HTTPCrypt, the sendfile interface needs to be extended to accept a session key and a generated nonce. Bulk data can be encrypted directly in the kernel. In contrast, in TLS other protocol components, such as TLS Alerts or other extensions all require complex processing that is not easy (or wise) to put into the kernel.

To migrate to HTTPCrypt from plaintext HTTP, applications must also use a cryptographic quality random number generator to generate nonces and key pairs. This is particularly important where there is not much entropy available, for example in embedded devices or virtual machines. However, this requirement is held for TLS as well.

3.2.11 Embedded usage

HTTPCrypt is particularly well suited to embedded devices where including the full TLS suite is too large or too slow to use. The CPUs used in embedded appliances are often not able to drive encrypted connections at a reasonable rate as they have neither hardware cryptographic acceleration nor optimised instructions cores. HTTPCrypt with the static key model is a better choice to protect communications on such embedded devices. Despite the fact that this scheme does not guarantee forward secrecy, it is still better than plaintext HTTP connections by providing stronger confidentiality and authentication properties. There are several optimised embedded implementations of the Cryptobox construction elements used in HTTPCrypt; for example, ARM NEON specific optimisations to speed up ChaCha20-Poly1305. There is also a generic implementation of Cryptobox optimized for code size and memory consumption called TweetNaCl. This library supports the digital signatures created by the Ed25519 algorithm, which can be used to check the identity of ephemeral keys via the PKI chain-of-trust model. We have evaluated the performance of generic version of our HTTPCrypt prototype on Cortex A20 ARM board. And even with the generic unoptimized C versions of all cryptographic primitives it has shown highes requests per second rate then TLS stack (50 complete encrypted requests per second against 30 requests per second for TLS). However, the detailed evaluation on embedded platforms is beyond the scope of this work.

3.2.12 Evaluation

We have built the prototype of HTTPCrypt built on top of the http-parser library that is in turn based on the popular Nginx HTTP server code. The goal of our tests was to compare HTTPCrypt with the standard web workloads using TLS. We have compared our implementation against Nginx 1.9.5 built with OpenSSL 1.0.2d using TLS v1.2 protocol with nistp256 curves for both ECDSA and ECDHE.

Integration with Existing Software

HTTPCrypt is designed to be integrated with existing application easily. While it is relatively straightforward to migrate the already written plaintext services to TLS by means of a proxy such as stud1, it is more difficult to integrate TLS stack directly into an application that is not specifically designed for to support TLS for the following reasons:

- TLS alerts and handshakes change the connection processing logic significantly, especially for asynchronous or non-blocking applications;
- TLS uses intermediate buffering for all data transfers that leads to additional latency and performance penalties.

In contrast, in HTTPCrypt, there are no protocol alerts or additional handshake stages to complicate integration. Applications thus can send or receive data without any intermediate steps, leaving the event processing logic in the client and server unchanged. In TLS, read operation might require writing and vice-versa: write operation might request reading. Moreover, an application can create messages in HTTPCrypt without copying data to an intermediate buffer since all data is encrypted and authenticated in-place. Reading and processing of HTTPCrypt requests is implemented by extracting the authentication tag from the encrypted payload, and parsing the following encapsulated HTTP request as defined earlier.
For HTTPCrypt testing, we wrote our HTTP server and benchmarking tool based on the same principles as wrk (non-blocking IO) and the same HTTP parser. We ran a sequence of experiments using both Nginx and the HTTPCrypt prototype. The HTTP client is the wrk HTTP benchmarking utility with a single testing thread and 50 parallel connections in the test runs. We first ran the servers in plain HTTP mode with no encryption enabled at all (Unencrypted). When then disabled SSL sessions cache/tickets in order to evaluate the performance of complete TLS handshakes (Encrypted, uncached). In the last experiment, we turned on the SSL session cache to evaluate the performance of session resumption (Encrypted, tickets) The selection of cipher suites and the ECDHE curve was based on the assumption that on the tested CPU with hardware AES support (via AES-NI instructions) and vectorised operations (AVX instructions), the speed of these particular primitives was optimal. We used the openssl speed command to ensure that the performance of specific algorithms was optimal on the tested hardware:
256 bit ecdh (nistp256)
256 bit ecdsa(nistp256)
aes-256-gcm
8582.6 op/s
15000.0 signs/s
731623.42 kB/s
In all cases, we evaluated serving static files using a single process on the client and one on the server to estimate latency and the number of requests per second that were processed. The client and server were connected over the 10G network interface, and all requests were successfully processed. We have also evaluated HTTPCrypt proxying using both forward proxies, such as squid or tinyproxy, and reverse proxies, such as nginx or lighttpd. We have not found any compatibility issues when traversing HTTPCrypt requests over these proxies.

Performance Evaluation

Figure 3.2 shows the workload patterns obtained from the HTTPCrypt test suite. The important results are the tests with encryption enabled, and in this case HTTPCrypt demonstrates significantly superior requests per second than Nginx/TLS for the transfer of small files. For larger request the throughput benefits of HTTPCrypt are not so clear, since the handshake cost is negligible comparing to the cost of bulk encryption and network transfer.

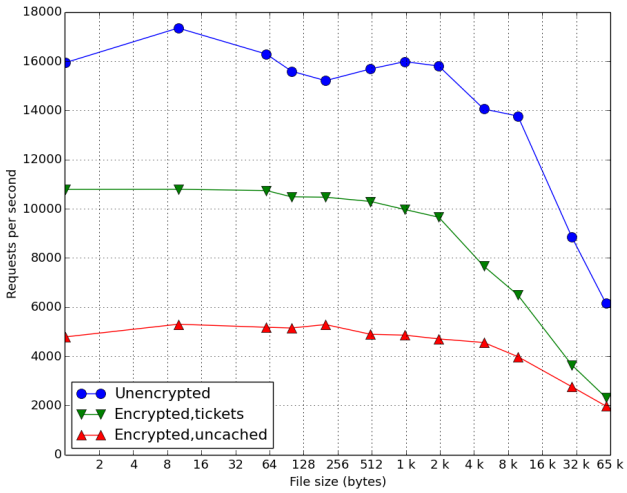


Figure 3.2: The performance of HTTPCrypt prototype with ChaCha20-Curve25519 crypto while serving files using 1 process on Intel Xeon E5 2.4 GHz

The better performance of HTTPCrypt is achieved by use of the faster ECDH crypto primitives, the elimination of the handshake stages, and skipping encryption of the unnecessary HTTP headers in favour of the payload. Furthermore in the Figure 3.3, we demonstrated the difference between OpenSSL and HTTPCrypt comparing them with nginx+TLS as the baseline. OpenSSL mode used the following set of crypto primitives: NIST p256 curve for ECDH, AES-256-GCM for authenticated encryption and hchacha20 as PRF (PRF selection does not influence the overall performance since its execution time is negligible comparing to ECDH procedure). This test shows that even with the equal cryptography model HTTPCrypt is significantly faster than TLS for small requests that are common for the web RPC services.
In the Figure 3.4, we demonstrated scaling of HTTPCrypt prototype from the number of independent processes running on the same machine comparing to TLS. For both nginx/TLS and HTTPCrypt we used the equal number of processes for both client and server. This experiment illustrates the capabilities of HTTPCrypt to scale with the CPU cores amount increasing.

Overhead evaluation

The traffic overhead is another important property of an encryption protocol. Therefore, we have compared the extra data required for HTTPCrypt and TLS connections depending on the payload size. We have chosen the plain HTTP request of the same size as the baseline and estimated both incoming and outcoming traffic on the client side. The results shows that HTTPCrypt introduces less overhead than TLS, especially for small requests, that could reduce the overall traffic for the encrypted connections comparing to the TLS case.

Latency evaluation

Request latency is an important property for opportunistic encryption, since we want this to be deployed widely and with no user-visible impact. We compared the request latency for different encryption methods using the unencrypted latency as the baseline.
The evaluation results are shown in Figure 3.5. We measured the latency between connecting to the HTTP server, sending a request and receiving the reply that concludes the complete HTTP session (keep-alive was disabled). Connection latency is included as well since the socket connection time introduces a small and constant delay. The latency tests are consistent with the earlier throughput evaluation: HTTPCrypt provides lower delay than TLS, but latency degrades when serving large files due to the current lack of IO optimisations in our prototype. However, once again the important result is that for encrypted connections, the benefit from HTTPCrypt compared to HTTPS is very significant, especially for the common case of small HTTP response sizes.
The most expensive computational operations for TLS are generating a shared secret and signing the random cookie. In contrast, HTTPCrypt does not require the server to do any signing, since the random cookie is cheaply placed within the encrypted and authenticated payload. The work in the first stage is significant

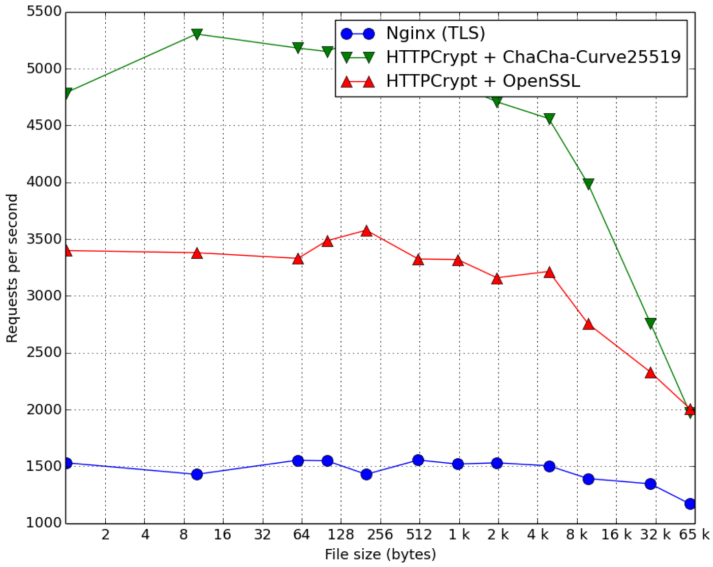


Figure 3.3: The performance of HTTPCrypt prototype with OpenSSL crypto while serving files using 1 process on Intel Xeon E5 2.4 GHz

comparing to the first stage of TLS; however this difference does not make HTTPCrypt more vulnerable to denial-of-service attacks than TLS.

3.3 Kadupul - incentive based enforcement liquid control

Devices connected to the Internet today have a wide range of local communication channels available, such as wireless Wifi, Bluetooth or NFC, as well as wired backhaul. In densely populated areas it is possible to create heterogeneous, multihop communication paths using a combination of these technologies, and often transmit data with lower latency than via a wired Internet connection. However, the potential for sharing meshed wireless radios in this way has never been realised, due to the lack of economic incentives to do so on the part of individual nodes. In this work, we explored how virtual currencies might be used to provide an end-to-end incentive scheme to convince forwarding nodes that it is profitable to send messages on via the lowest latency mechanism available. Clients inject a small amount of money to transmit a message, and forwarding engines compete to solve a time-locked puzzle that can be claimed by the node that delivers the result in the lowest latency. Our approach naturally extends congestion control techniques to a surge pricing model when available bandwidth is low and does not require latency measurements.

Devices connected to the Internet today have a wide range of local communication channels available. For example, most new wifi-routers and access points have two or more radios (one for 2.4 GHz and one for 5GHz communication). Connected to each access point there are clients with several radio technologies available, such as Bluetooth and NFC. Other physical communication channels also exist, for example LEDs, cameras and microphones depending on available hardware.

In urban areas it is possible to create heterogeneous, multihop communication paths using these technologies. As radio waves propagate at the speed of light, these paths offer lower-latency communication. However, there are few economic incentives for edge nodes to act as low-latency data forwarders, and the disincentive of wasting their batteries on other nodes' traffic. As an example of how wireless edge nodes can be used for faster forwarding, consider a user A who wants to send messages to user B a few kilometers away, as illustrated in Figure 3.6.

Using a traditional forwarding path the messages may first have to be delivered through the core network of A's ISP (or mobile operator), then be forwarded to the core network of B's ISP, before finally being delivered to user B. The forwarding latency in this example depends on the number of hops and distance to travel via the core networks of the ISPs, not the geographical distance between the nodes. As a result, nodes A and B may experience the same latency whether the geographical distance between them is one or tens of kilometers.

An alternative forwarding path could be established as a wireless path through intermediate radio devices between A and B, such as node C in Figure 3.6. This forwarding path could potentially achieve significantly

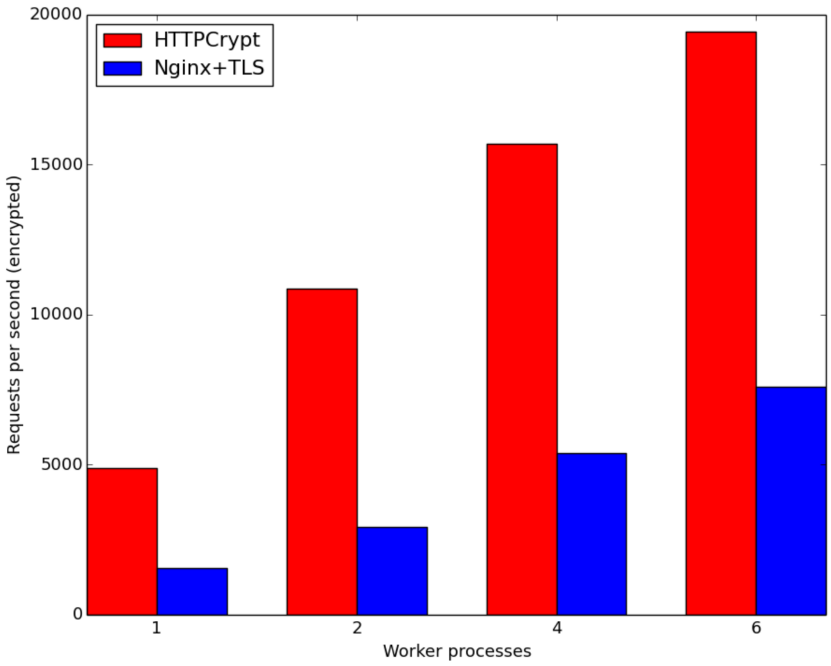


Figure 3.4: HTTPCurve and Nginx+TLS scaling from the number of worker processes

lower latency than traditional methods, as well as being resilient to wide-area networking failures since it only depends on the local communications network. There are however few devices today that are willing or able to participate in the network as low-latency edge forwarders. We argue that the primary reason for this is not technical, but caused by lack of incentives compared to the increased workload and need for investment in the edge nodes. For example, it is not uncommon for edge nodes to participate as forwarders in city-wide mesh networks due to bad or expensive Internet connectivity, or to act as forwarders to improve communication during a political crisis. Examples of the latter are Guifi in Spain, the Occupy Wall Street movement mesh network and the Open Mesh Project in Egypt. In these cases the incentives caused by external factors outweigh the forwarding cost.

The perceived cost of forwarding a message depends on the workload imposed on the edge node. Depending on the technology, this workload may be low (forward on regular Wifi), but one could imagine higher workloads - such as for a mobile phone that has to turn on Bluetooth discovery for long time periods. The owner of the node may also have to install custom software or invest in upgraded hardware to forward messages faster. Other factors, such as power consumption may also play an important role. A forwarding incentive can be created by simply offering payment to the forwarders, for example by using a decentralized virtual currency like Bitcoin. A useful feature of many virtual currencies is that micro-payments can be issued with minimal transaction costs and without relying on a centralized authority. It is more difficult to create a payment system with incentives for minimizing latency, since it is difficult to measure latency objectively in a way that can be accepted by all nodes. For example, the latency observed by the sender, a forwarder and the final recipient may not be the same, and all parties have economic incentives to under or overreport the latency.

Kadupul is a system that incentivises low-latency forwarding between edge nodes without relying on latency measurements. This is accomplished by creating a reward system based on time-locked puzzles. Time-locked puzzles can be used to hide information until the puzzle is solved or the solution is provided by the creator or a third-party. Recently, a time-locked puzzle based on Bitcoin was proposed and implemented, which allows Bitcoin rewards to be locked for a given time period and be collected by the first node that solves the puzzle (or is told the solution). We build on Todd's time-lock puzzle mechanism to propose a forwarding model for rewarding forwarders by giving them an advantage in solving a puzzle. A forwarder can collect a reward if it provides the correct solution to a puzzle protecting it. Each forwarder is provided with a solution to a reward after it has forwarded a message. The catch is that each puzzle is public and solvable by anyone after a known amount of time. This creates a race to forward the message before the puzzle has been solved by other nodes. Our main contribution is a new forwarding model that creates incentives for low-latency forwarding without

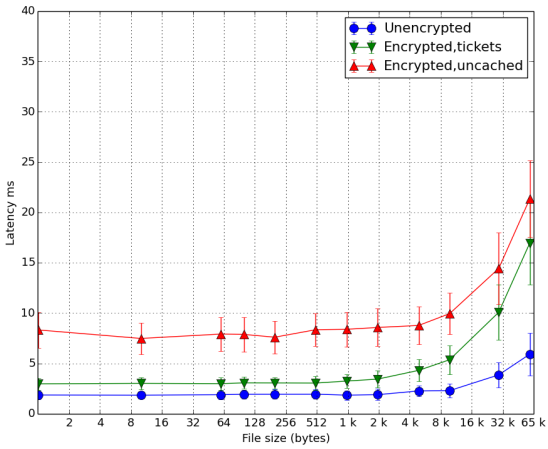


Figure 3.5: HTTPCrypt Latency

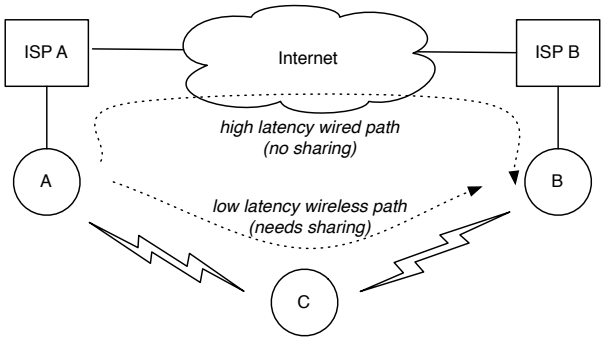


Figure 3.6: ISP and edge forwarding paths between nodes A and B

having to measure the latency. Economic incentives for optimizing other parameters in ad-hoc networks, such as bandwidth or power, have been proposed in earlier works, primarily Bit coin as a reward system, the ideas described here can be used with other virtual currencies as well, as long as they provide a similar underlying P2P protocol. We will now describe the basic mechanisms in Kadupul and propose several forwarding models based on time locked puzzles as incentives.

3.3.1 Kadupul Design

We describe the core functionality of Kadupul by first discussing how the forwarding paths are established and negotiated. We then describe in detail how time locked puzzles can be used as a low-latency forwarding incentive and propose several forwarding models based on the mechanism.

We assume that most of the nodes that participate as forwarders in Kadupul rarely change their geographical location (although they can). This means that although nodes may be anonymous as users of the virtual currency, they will over time be able to gather information about other radio nodes in their region. Kadupul nodes may use this to estimate risk by observing the behaviour and reliability of nodes they cooperate with over time.

A message forwarded by Kadupul can be of any size. For example, a high quality video can be transferred as a single message.

Establishing forwarding paths

Some coordination must be performed in advance to form heterogeneous multihop communication paths. The forwarders along the path may be able to discover each other directly using radio or other techniques, but if a wide range of technologies are being used over large geographical areas this may not always be possible. In the following we provide an example of how a forwarding path may be set up with the help of a decentralized P2P network on the Internet. We assume that in most cases the forwarding path is established by the sender. This requires an initial discovery step where the sender finds potential forwarders that together can forward

information from the sender to the receiver. Potential forwarders can for example be discovered based on their location and local communication range, using a decentralized, Internet-based P2P network.
When a potential forwarder is found, the sender contacts the node directly over the Internet and requests a forwarding quote. If the price and capabilities of the forwarder is acceptable, the sender attempts to find other nodes that are able to receive the message when it is forwarded. The process is repeated with these nodes. If no recipients are found (or the prices are unacceptable), the sender must find a different forwarder. Note that a forwarder may have different radio technologies available, be able to transfer on different frequencies and at different power levels. Each configuration may have a different expected forwarding latency and price.

Time-locked forwarding rewards

After establishing the forwarding path, the sender must publish a set of rewards. We now describe the mechanism and protocol used to publish, as well as collect rewards for the forwarding nodes in more detail. Todd proposes an interesting scheme for implementing time-locked puzzles with Bitcoin rewards. A chain of rewards can be hidden in puzzles and published. Each puzzle has a value in Bitcoins that can be collected by any node that knows the solution to the puzzle. The solution can either be provided or can be calculated after a known amount of time. The puzzle is constructed in such a way that it can be created in parallel, but only solved in serial. More specifically, the scheme uses multiple rounds of a SHA256 hash to calculate a Bitcoin secret key from a randomly chosen initialization vector for each block of the reward puzzle chain. If the reward chain has for example 10 blocks, 10 initialization vectors are chosen and SHA256 is executed iteratively on each vector. Based on the result, a Bitcoin secret key is generated, which in turn is used to generate a public key and a Bitcoin address. The number of iterations determines how long it will take to recover the key pair if only the initialization vector is known. When the key pair is found, the public key must be revealed to collect the reward.
When the reward chain is made public, the initialization vector in each block (except the first) is obfuscated by XORing it with the SHA256 of the public key of the previous block. Thus, to decode a reward key pair iteratively you would first need the public key used to collect the reward from the previous block. This prevents the puzzles from being solved in parallel, while forcing reward collectors to reveal the missing piece of information required to collect the next reward. To estimate the time it takes to unlock the reward we must assume a lower bound for how long it takes to perform the hash operation. As SHA256 is used to mine Bitcoins there have for several years been strong economic incentives to create faster hardware and software solutions to calculate it. A list of known Bitcoin hardware miners showing speeds in million double SHA256 hashes per second (Mhash/s) and power consumption (MHash/J) is maintained online. For example, the fastest ASIC-based device currently available can calculate 5.500.000 Mhash/s at 1506 Mhash/J. However, these devices rely on parallel operations and can not be used to crack a time-locked puzzle that must be solved in serial. A more realistic lower bound is thus similar to a very fast single core CPU or FPGA. The lower bound can be adjusted over time if it is observed in the block chain that a puzzle is broken faster than intended. This mechanism can be adapted to create forwarding incentives in several ways, and we discuss four such forwarding models next. Note that control traffic (but not the actual data) is transferred over a higher latency control plane, which for example could be the Internet.

Double incentive forwarding

The objective in this model is to create a mechanism that makes the forwarders lose their reward unless they forward the message intact to the next hop as soon as possible, but also to create an incentive for assisting other forwarders. The full process is shown in Figure 3.7.
Figure3.7a gives an overview of the initial tasks that must be performed before forwarding can begin. First, the sender must find the forwarders and negotiate the forwarding fees. The sender then generates and makes public a chain of rewards using time-locked encryption. For simplicity, we assume that one reward block is generated for each forwarder. The reward attached to each block may differ in value depending on the terms that were negotiated with the respective forwarder. As shown in Figure 3.7b, the sender proceeds to send two values to each forwarder. The first value is a secret that enables the previous hop on the forwarding path to retrieve its reward. The second value is a nonce that when combined with a secret from the next hop on the forwarding path, as well as the hash of the message, results in the key required to unlock one of the rewards. To avoid having to store the full message in each node, the nodes may use a rolling hash function to hash the message. The nodes then only need to maintain a buffer that contains a window with enough information to create the hash and to act as a message queue for forwarding.

Now the forwarding itself can begin, as shown in Figure3.7c. The message is forwarded along the path and acknowledged by the next hop by sending the secret back to the forwarder. The acknowledgement address in the control plane is also sent with the message - for example an IP address and port number.
Figure 3.7d illustrates the reward collection process. After a node has forwarded the full message and received the required secret from the next node, it can reconstruct the puzzle solution and collect a reward. A forwarder will only be able to claim a reward when:

(i) The previous node in the routing path was able to claim their reward and thus revealed the public key of the previous block

(ii) It has received the full message successfully so that it can generate a hash

(iii) The next node in the path has revealed the necessary secret generated by the sender, thus acknowledging that the full message was forwarded.

If the receiver is untrusted, the sender may encrypt the message with a key derived from the public key of the final reward to make the message unreadable until all forwarders have been rewarded. This mechanism ensures that all nodes have incentives both to forward the message to the next hop (to obtain the missing secret to unlock the reward) and to supply the previous hop with its secret (otherwise they are unable to decode their own reward). As the reward has a time-lock, the nodes are also given an incentive to perform the forwarding as fast as they can, or the encryption may be broken by other nodes in the Bitcoin network trying to claim the reward by brute forcing the keys. The node that can most effectively balance high-throughput and low-enough latency forwarding stands to profit most from solving the majority of puzzles it sees and claiming the rewards.

All or nothing

Note that in the forwarding model discussed above, the identity of every forwarder must be revealed to the next forwarder to allow the message to be acknowledged with the correct secret. In a broadcast network this may not be necessary. For example, a forwarder may be instructed to listen to a specific Wifi channel in promiscuous mode to receive the message and then just forward the message as it was received over another broadcast link. It may not be possible for the forwarders to reply with an acknowledgment in the same way if it has a weaker radio than the previous hop. It is possible to use an alternative acknowledgement mechanism to avoid revealing the identity of the forwarders. This forwarding scheme is shown in Figure 3 and assumes a forwarding mechanism that can hide the Internet address of the forwarder, for example by using anonymous broadcast. Instead of distributing secrets in advance, the final receiver acknowledges the receipt of the message to the sender and the sender then unlocks the puzzles for all the forwarders. This scheme requires more coordination between sender and receiver, but makes it difficult for the forwarders to collude. It also increases the forwarding risk as none of the nodes will receive their reward if the message is lost or delayed along the way, which may affect the forwarding price.
This forwarding model can be useful in ”off-the-grid” mesh networks, where one of the goals is to avoid eavesdropping of traffic by global passive sniffing of conventional networks.

Contract forwarding

Kadupul forwarding may also be used without establishing the forwarding path in advance. In this case the sender negotiates a forwarding contract with another node to bring the message to the recipient. It is then up to the node that accepted the forwarding contract to deliver the message as fast as possible. This node may use any number of subcontractors for the message to reach its final destination. This forwarding model is potentially simpler to implement and use for the sender, but the sender is no longer in control of the path the message travels. It may also increase the forwarding price as more work is left to the forwarders.
This forwarding model can especially be useful in combination with the other forwarding models. For example, a pull based delivery system can be constructed by letting the recipient negotiate a contract with the sender. When the contract has been accepted, the sender uses another forwarding model to deliver the content to the recipient.

Competing forwarders

The ”all or nothing” model above can be extended to create a competition between forwarders along multiple paths. This can be accomplished by using Random Linear Network Coding (RLNC) or fountain codes to

encode partial messages and then forward- ing the messages along multiple paths at the same time. RLNC and fountain codes are useful because the original message can be reconstructed when enough coded packets have been received. Fountain codes are end-to-end, but RLNC also allows recoding at intermediate nodes. When the recipient has received enough partial messages to reconstruct the original message, rewards are distributed to the forwarders depending on how much innovative information they forwarded.
This forwarding model can be especially useful for transferring messages that should be distributed to multiple edge nodes in the same area, such as for multicasting video streams.

Edge node caching

Kadupul naturally creates incentives for edge nodes to cache content. If an edge node is able to store content that is delivered frequently, it can volunteer to deliver the full content during the negotiation process and collect the full reward for the delivery. It would also be able to deliver the content in much shorter time than if it would have to be forwarded again, allowing it to provide a better offer than its competitors. Since each node must pay for its own cost of storing the cached data in the hope of future requests, they also have an incentive to develop efficient prediction and cache eviction algorithms.

Discussion and Conclusions

We have proposed a mechanism for creating an economic incentive for low-latency data forwarding and described how the mechanism can be useful for establishing low-latency forwarding paths between edge nodes on the Internet. The mechanism is decentralised and does not require latency measurements. We have mainly focussed on edge networks in this work, but the mecha- nism can also be used in other types of networks where low latency should be rewarded.
An advantage of Kadupul in edge networks is that forwarding nodes have incentives to avoid congestion and long processing delays. If an area is congested, nodes that are able to forward with low latency would be able to demand a higher price. This increased price encourages central nodes to invest in special hardware to reduce the congestion. Two nodes may for example cooperate and set up long-range, point-to-point links that increases hop length and reduces interference. Similarly, nodes that are near a congested area, may set up point-to-point links that forward data across the congested area to a node on the other side. Commercial operators may also take advantage of the system, for example by selling low latency access using a privately owned frequency range.
Although all of the pieces needed to implement this type of forwarding exists today, some technical challenges remain before it can be fully realised.
For instance, it may take a long time to set up the initial forwarding path because several potentially slow tasks must be performed before forwarding can begin. For example, the reward puzzles can be generated in parallel, but may still be time consuming to produce if the puzzle is to be protected over longer time periods. In addition, if Bitcoin is used, transactions are relatively slow and it takes a while to publish rewards. Similarly, it may take some time for the forwarders to collect their rewards after receiving the solution to the puzzle. These delays require that additional time must be added to the puzzles to make sure that rewards do not expire until the forwarders have been able to collect them. Today, this makes the method mostly suitable for applications that will use the path for longer time periods, such as for large, planned data transfers. However, we are confident that these delays will become much smaller in the near future. As virtual currencies are becoming more popular, their protocols and software implementations are constantly being optimized to reduce delays. Furthermore, the number of reward puzzles that can be generated in parallel can dramatically increase if they are calculated using GPUs or FPGAs. Multiple rewards can be given to each hop while still being required to be solved in serial, and so the number of rewards created in parallel can exceed the total number of hops. Kadupul helps to balance the economic needs of service providers and users to deliver a viable model for deploying reliable multihop edge networks, as well as enhancing the resilience of the global network by only using local links when possible.

3.4 Managing Multipath TCP sufblows

3.4.1 Introduction

An important part of a Multipath TCP implementation is the strategy that it uses to create subflows. The Linux kernel implementation [38] includes modules that implement these strategies. As of this writing, the Linux implementation contains two strategies that are called path managers for historical reasons : `full-mesh`

and ndiffports. In both cases, only the client creates the subflows. The server never creates subflows because the client is often behind a NAT or firewall that blocks connection attempts [19]. The full-mesh path manager listens to events from the underlying network interfaces and creates one subflow towards the server over each active interface. These subflows are created immediately after the creation of the connection or when an interface becomes active. It enables smartphones to react to losses of connectivity [37]. The ndiffports path manager creates n subflows over the same interface as the initial one immediately after the establishment of the connection. This path manager was designed for datacenters [42] where it enables the utilisation of paths that are load-balanced with Equal Cost Multipath (ECMP).
A few researchers have explored how Multipath TCP should manage the available subflows and interfaces. RFC6897 proposes some extensions to the basic socket API to enable applications to add/remove addresses to a Multipath TCP connection [44]. However, none of the existing Multipath TCP implementations implement this proposed extension [19]. Paasch et al. [37] evaluate how wireless devices can adapt to losses of connectivity. This paper proposes three modes of operation for Multipath TCP on smartphones: single-path, backup and full-mptcp. Bocassi et al. [7] propose the Binder path manager that leverages loose source routing and Multipath TCP to aggregate different paths in wireless mesh networks. Lim et al. [49] propose an extension to Multipath TCP that enables to adapt the utilisation of the subflows based on information extracted from the MAC layer. This extension is evaluated experimentally, but there are no details on how it has been implemented. Several researchers have evaluated the energy impact of using Multipath TCP on smartphones [40, 30]. Peng et al. propose models that demonstrate the benefits of managing the subflows to reduce energy consumption but do not propose any implementation [40]. Lim et al. propose an energy-aware Multipath TCP (eMPTCP) [30]. eMPTCP delays the establishment of subflows on smartphones over the LTE interface. However, when the smartphone switches from LTE to WiFi, they propose to reset the round-trip-time estimation of the LTE subflow to zero msec to force the utilisation of this subflow [30]. This speeds up the utilisation of the LTE subflow, but is not an architecturally clean solution to the subflow management problem.
Schmidt et al. proposed the utilisation of *socket intents* [45] to allow applications to inform the stack of what they know about the future communication pattern. These *intents* include information such as the type of transfer (query, bulk, stream) or the information about the flow (number of bytes, duration, ...). We also use this kind of information in our design and *socket intents* could be a way to exchange it with the subflow controller.

3.4.2 The subflow controller

The Linux implementation of Multipath TCP [38] resides entirely in the kernel. Most of the kernel code is devoted to the transmission and reception of data, but the management of the subflows is also performed in the kernel by the full-mesh and ndiffports path managers. This design choice was motivated by performance reasons. An unfortunate consequence of this choice is that if an application wants to control the utilisation of the subflows, it must include a new kernel module. This is not a good solution and only three path managers have been implemented in the kernel in several years.
We reconsider this design choice by clearly separating the Multipath TCP *data* and *control* planes. The data plane includes all functions that deal with the transmission of data. It remains in the kernel for performance reasons. The control plane includes all the functions that manage the subflows that compose a Multipath TCP connection. From a performance viewpoint, there is no reason to place these functions in the kernel. Furthermore, some applications might want to implement complex policies to manage their subflows. This kind of code does not really fit inside a kernel. To enable the applications to interact with the Multipath TCP kernel code, we define a new Netlink family. Netlink [43] is an interprocess communication mechanism supported by the Linux kernel that allows applications to interact with the kernel through messages. This is similar to the approach proposed earlier by M. Coudron [11].
However, writing code to send and receive Netlink events can be complex for application developpers. To ease the development of subflow controllers, we abstract all the complexity of handling Netlink in a library [14] that is linked with the subflow controller running entirely into userspace. This library (Figure 3.9) interacts with the Netlink path manager that is part of the kernel. This path manager uses the existing in-kernel path manager interface (shown in red in Figure 3.9) and exposes this interface through Netlink. The path manager is implemented in 1100 lines of C code while the library contains 1900 lines of code.
The Netlink path manager provides a flexible API that exposes events and state information from the kernel [14]. Callback functions provided by the subflow controller are triggered when a specific event happens

in the Multipath TCP kernel or based on other inputs. The subflow controller receives only notifications for events it registered to.

The `Netlink` path manager sends and receives messages that contain information about the connection, the subflow(s), the type of event, etc. It supports many more events and commands than M. Coudron's earlier prototype [11]. The `created` event is triggered when a Multipath TCP connection is established. It contains the four-tuple, the id of the initial subflow and other information required to identify the connection. The `estab` event indicates the success of the three-way handshake and the `closed` event marks the termination of the Multipath TCP connection. These events enable a path manager to manage the connections established by several applications.

The `add_addr` and `rem_addr` events provide the IP addresses announced and removed by the remote host with Multipath TCP options [21]. Thanks to these events, the subflow controller can store the addresses of the remote host and establish new subflows only when and if needed. This is more flexible than the existing in-kernel path managers that immediately create subflows. This also reduces the state maintained for each Multipath TCP connection in the kernel.

The `sub_estab` and `sub_closed` events enable an application to control the utilisation of the subflows. The `sub_estab` event is triggered once a new subflow has been established. A server could use this event to limit the number of subflows that it currently accepts (e.g., only accept subflows originating from different addresses to prevent ressource abuse with parallel subflows). The `sub_closed` event is triggered when a `RST` segment is received over one subflow or when a subflow is terminated due to excessive retransmissions. This event is also associated with an error code (based on standard `errno`) that indicates the reason for the removal (e.g., excessive expirations of the rto, destination unreachable, etc.).

The last event is the `timeout` event. On a TCP connection, the expiration of the retransmission timer is usually an indication of severe losses. With regular TCP, there is nothing that the application could do if the retransmission timer expires too often. With Multipath TCP, the situation is different since a second path could have very different loss characteristics than the current one. This event reports the current value of the retransmission timer and can be used as a trigger to create an additional subflow.

In addition, the `Netlink` path manager also gathers the events that are triggered by the interfaces when a local IP address is enabled (`new_local_addr`) or disabled (`del_local_addr`).

In addition to subscribing to some of these events, the library enables the subflow controller to modify the state of Multipath TCP connections through commands. Our initial implementation supports several types of commands. First, it is possible to request the creation of a subflow. A controller can create a subflow based on an arbitrary 4-tuple (IP addresses and ports). A similar command allows to remove any established subflow (either created through the controller or established by the peer). This enables the subflow controller to easily adjust the utilisation of the subflows. The controller can also retrieve information from the control block of the Multipath TCP connection or one of the subflows. In practice, this is equivalent to the utilisation of the `TCP_INFO` socket option on Linux.

3.4.3 Sample use cases

In this section, we illustrate the benefits of the userspace subflow controller with different use cases that demonstrate how a smart application can intelligently interact with Multipath TCP.

3.4.3.1 Smarter long-lived connections

Some applications such as `ssh`, various chat applications, or notification applications on smartphones use long-lived connections that can last hours or days. These connections pose operational problems in networks that contain middleboxes like firewalls or NAT that maintain state for each established connection. The typical example is a connection that has been established but did not recently transmit data. Many NATs or firewalls will drop the state for this connection after some time. Although the IETF recommends a timeout of not less than two hours and four minutes, many deployed NATs and firewalls are more aggressive and remove unused state after a few hundreds of seconds [24]. Furthermore, many networks include cascades of such middleboxes [34]. Some applications react by sending data on a regular basis over each established connection. As an example, RFC3948 [26] recommends to send keepalive packets every 20 seconds. An unfortunate consequence of this battle between applications and middleboxes is that mobile devices need to consume a lot of energy simply to preserve the state for the established TCP connections in the middleboxes. Given the importance of energy consumption on such devices [40, 30], this is not a good approach.

Our first subflow controller is a reimplementation of the `fullmesh` path manager that is present in the Multipath TCP Linux kernel. This controller is implemented in about 800 lines of user space C code. It implements a listener for all the events described in Section 3.4.2. It listens to the `new_local_addr` and `del_local_addr` events to react to the activation and deactivation of local addresses like the in-kernel path manager. In addition, it also listens to the `sub_closed` event to react to the failure of any subflow. When such an event occurs, the subflow controller analyses the error condition (excessive timeout, `RST`, reception of an `ICMP` message, etc.) and reacts accordingly. It tries to reestablish the failed subflow and sets different timeouts based on the error condition (e.g. a short timeout if a `RST` was received and a longer timeout upon reception of an ICMP network unreachable message). Experiments with this controller show that it correctly maintains the subflows established over the different paths even under difficult network conditions.

3.4.3.2 Smarter backup

Multipath TCP [21] supports backup subflows. The backup status associated to a subflow is a binary flag that is exchanged in the `SYN` segment at subflow establishment time. It can also be changed dynamically with the `MP_PRIO` option [21]. According to RFC6824 [21], a backup subflow is a subflow that should only be used to transmit data once all other (non-backup) subflows have failed. This is the classical definition of a backup interface that works well on hosts. When an interface fails on such hosts, Multipath TCP immediately detects the failure and moves the traffic to the backup interface [37].

On mobile devices such as smartphones, the availability of one interface cannot be represented as a binary variable. When a smartphone moves around an access point or a cellular tower, there are regions where the wireless network does not work at all, regions where it works perfectly and regions where an IP address is assigned to the smartphone, but the radio conditions are so bad that most of the packets are lost. We use a Mininet [22] emulation to illustrate the situation experienced on smartphones. A connection starts over one interface and the second is set as a backup interface. After 1 second, the packet loss ratio over the primary path increases to 30%. Multipath TCP tries to retransmit the data over this interface and applies the exponential backoff to its retransmission time until it reaches the maximum value (15 doublings on Linux). At this point (after 12 minutes in our experiment with the default Linux configuration), TCP eventually terminates the subflow. This triggers Multipath TCP to use the backup subflow since it is the only available one.

The userspace subflow controller enables a different model for backup subflows that improves user experience. Since Multipath TCP supports break-before-make [21], our controller does not immediately establish the backup subflow. On a smartphone where the cellular interface would likely be used as a backup, this reduces both energy and radio resource consumption. The controller simply listens to the `timeout` event. When a retransmission timer expires, it checks the current value of the timer. If the timer becomes larger than a configured threshold, the subflow is considered to be underperforming. The controller then closes the underperforming subflow and creates a subflow over the backup interface to continue the transfer. This is illustrated in Figure 3.10 which shows the evolution of the data sequence numbers (the color indicates the subflow used to send the data). The transmission starts over the primary subflow (in green in Figure 3.10). When the retransmission timer reaches one second, this subflow is terminated and a new subflow is created over the backup path (in red in Figure 3.10).

3.4.3.3 Smarter streaming

We consider a simple streaming application that sends one 64 KBytes block every second. It expects that each block of data will be delivered within one second. We use this application over an emulated network with two 5 Mbps links between the client and the server. Each link has a 10 msec delay. The link bandwidth is almost an order of magnitude larger than the application goodput (520 Kbps).

For this Mininet experiment, we first use the default `full-mesh` path manager. When there is no loss, each block of 64 KBytes is delivered within 100 msec. However, when there are losses over the initial subflow the block delay quickly increases as shown by the CDF in Figure 3.11.

A closer look at the packet traces reveals the reasons for the low performance achieved with the default `full-mesh` path manager. When a retransmission timer expires on the initial subflow, the corresponding data can be reinjected on the other subflow. However, the data is still retransmitted on the initial subflow. If the retransmission is lost, the retransmission timer is doubled. This can happen several times and most of the data is sent on the second subflow. If at this point the scheduler decides to send some data on the underperforming subflow, this data is protected by an already very long RTO. If the data is lost, Multipath TCP waits a long time to retransmit it, which explains the long tail of the CDF in Figure 3.11.

We prototype a subflow controller that expects the blocks of data to be delivered within 1 second. 500 msec after each start of block, it measures the progress of the data transfer by extracting the `snd_una` state variable from the kernel. If fewer than 32 KBytes have been sent, it considers the subflow to be underperforming and opens another subflow on the other interface. This controller also monitors the evolution of the RTO. If the RTO of a subflow becomes larger than 1 second, it is immediately closed. With this controller, if the initial subflow is fast enough to support the stream no additional subflow is established. If the initial subflow does not have enough bandwidth, a second subflow is established. The controller can also close the initial subflow if its performance is too bad. Our experiments with different packet loss ratio (not shown in the figure) for graphical reasons show that our controller provides almost the same CDF of the block delays for packet loss ratios in the 10-40% range.

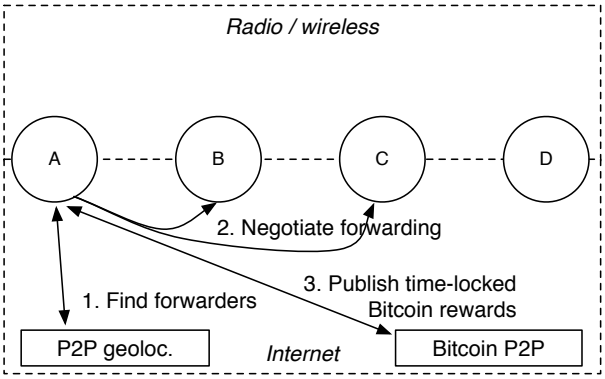
3.4.3.4 Smarter exploitation of flow-based LB

In many networks, there are multiple paths between a pair of single-homed hosts given the widespread usage of Equal Cost Multipath (ECMP), link bonding and other techniques that perform flow-level load-balancing. Typically, load-balancing routers compute a hash over the four-tuple to select the path for each flow. This implies that hosts cannot easily predict which path will be used for a particular flow. The `ndiffports` kernel path manager was designed with this use case in mind [42]. If many paths are available, the n subflows that it creates are likely to use different paths. However, if the number of available paths is close to n, several subflows might use the same path. Measurements and simulations with this use case have always used a fixed number of subflows [42].

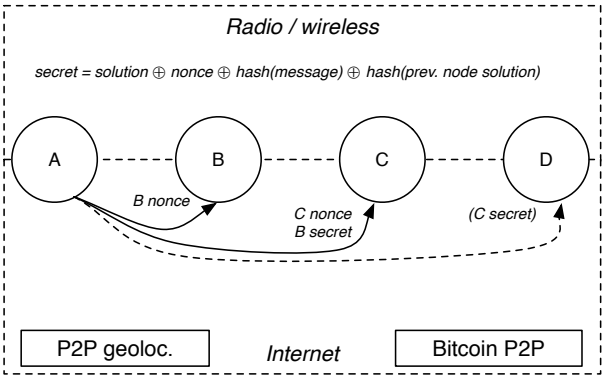
The flexibility of our subflow controller enables a different approach to the management of the subflows in such a scenario. When the connection starts, our controller creates n subflows. These subflows use random source ports and are load-balanced in the network. Regularly (every 2.5 seconds in our current implementation), the controller queries the Multipath TCP stack to retrieve the `pacing_rate` of each subflow. This `pacing_rate` is a state variable that measures the current rate of a given TCP connection. It is included in recent versions of the Linux TCP stack [18]. Our controller compares the `pacing_rate` of the different subflows, removes the one with the lowest rate and immediately creates a new subflow. This is a very simple heuristic that will need to be updated based on experience in real networks. This controller is implemented in only 230 lines of C code. We evaluate it in a simple Mininet environment. The client and the server are attached to different routers. The two routers load-balance the flows over four available paths that have a capacity of 8 Mbps and delays of respectively 10 msec, 20 msec, 30 msec and 40 msec. The client sends a 100 MBytes file and opens 5 subflows. We expect that by opening 5 subflows over 4 paths, our controller will end up continually using all four paths, while the in-kernel `ndiffports` path manager will likely have at least some of its 5 subflows using the same path. This is illustrated in Figure 3.12 which shows the CDF of the file transfer times. With the in-kernel `ndiffports` path manager, we can identify 3 clusters around 28 s, 37 s and 55 s, corresponding to the subflows using 4, 3 and 2 paths respectively. Even with a very simple implementation, our subflow controller tends to use the 4 available paths, outperforming the in-kernel `ndiffports` path manager significantly. For reference, the shortest time using the four paths is 27.8 s, and the worst time using only one path is 111.7 s.

3.4.3.5 User space path manager performances

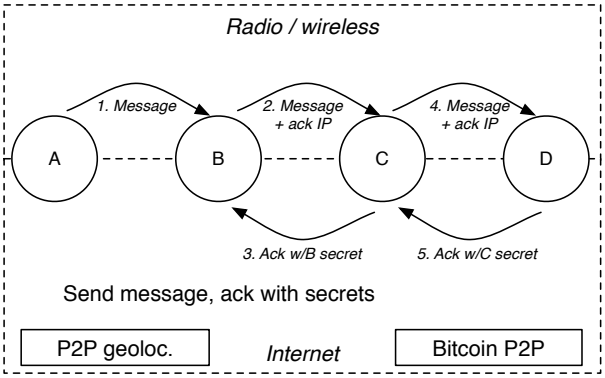
As a first evaluation of the CPU cost of the user space path manager, we perform a simple experiment in a lab between two hosts connected with a direct 1 Gbps Ethernet link. The server has an Intel(R) Xeon(R) CPU X3440 @ 2.53GHz and 8GB of memory. The client has an Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz and 4GB of memory. The server runs the `lighttpd` HTTP web server and the default in-kernel path manager. The client performs one thousand consecutive HTTP/1.0 GET queries for a 512 KB file. This experiment is performed with two variants of the `ndiffports` path manager : user space and in-kernel. These two path managers create a second subflow as soon as the initial subflow has been established. We measure the delay between the `SYN` of the initial subflow (i.e., containing the `MP_CAPABLE` option) and the `SYN` of the second subflow (i.e., containing the `MP_JOIN` option. Figure 3.13 provides the CDF of the delays measured with the two different path managers. It shows that the in-kernel path manager is slightly faster than the user space one. On average, the user space path manager increases the delay by 23 us. This additional delay is small and remains acceptable for a client. We also performed experiments during which we stressed the CPU on the client by running additional processes. In this case, both the in-kernel and the user space path managers are affected. The delay increase due to the user space path managers remains smaller than 37 us.



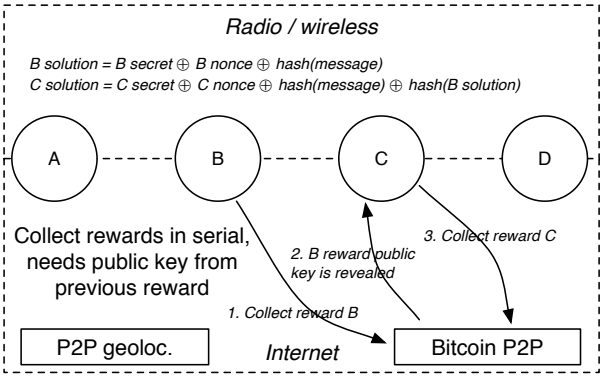
(a) Sender prepares forwarding path



(b) Sender distributes secrets and nonces



(c) Sender sends message, nodes acknowledge with secrets



(d) Forwarders reconstruct solutions and collect rewards

Figure 3.7: Kadupul Forwarders

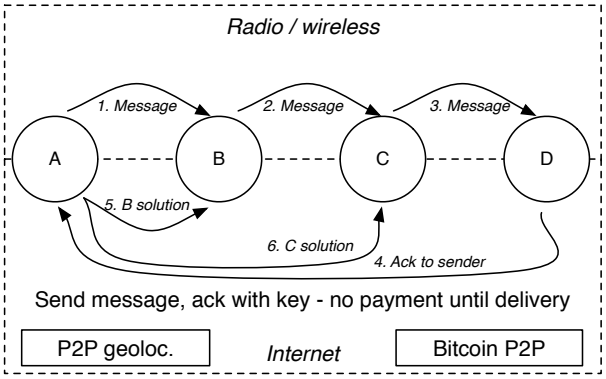


Figure 3.8: Deferred payment

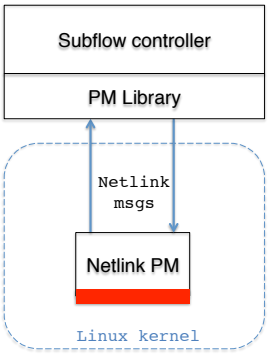


Figure 3.9: The subflow controller and the `Netlink` path manager

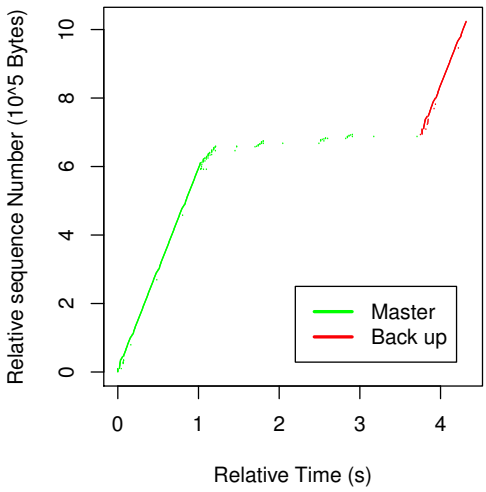


Figure 3.10: The subflow controller detects when the retransmission timer becomes too long and creates the backup subflow at this time.

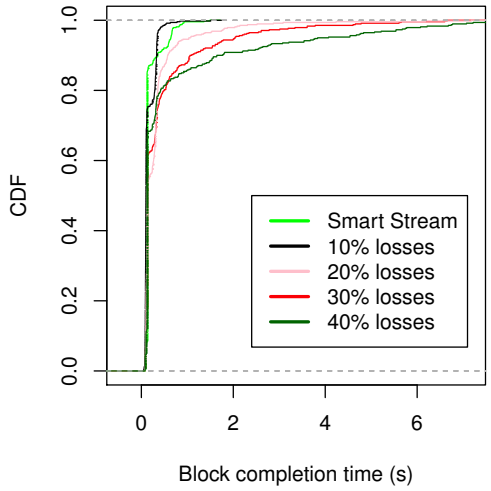


Figure 3.11: CDF of the delay required to deliver a 64 KBytes to the client under different packet loss conditions.

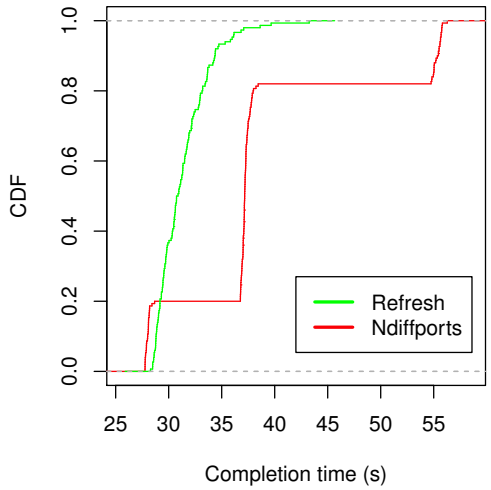


Figure 3.12: By regularly restablishing low-performing subflows, our subflow controller improves network utilisation

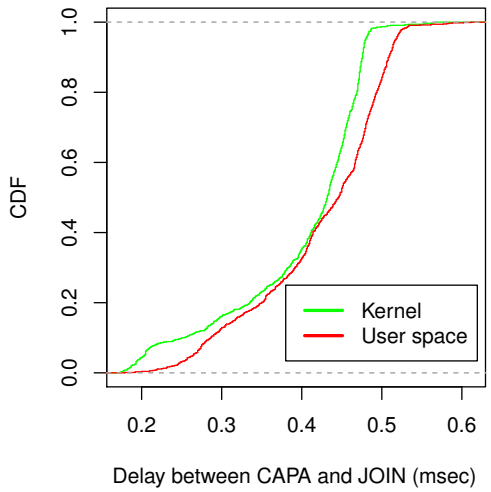


Figure 3.13: Kernel path manager is slightly faster than user space path manger to open a second subflow

4 Operator Control

In this section we present the tools developed in the third and last year of the Trilogy 2 project for the control of liquidity from the operator perspective.
We first present the control tool available in the Federated Market and Cloud.net. As described in deliverables D1.3 and D2.4, the Federated Market is a platform developed in the Trilogy 2 project to allow cloud operators to offer their compute/storage/network resources to clients. Through the platform, the providers and clients can trade cloud resources. However, in order for this platform to be successful, control of the traded resources is paramount. In this section, we describe both the incentive and the enforcement mechanisms built in the platform to provide the required guarantees to all the involved parties.
The other control tool for operators we cover in this section is related to MPTCP. The Trilogy 2 project has done extensive work on MPTCP as an end-user liquidity tool. Indeed, MPTCP is an end to end protocol that allows endpoints to pool all the paths available and distribute the traffic load among them. In section 3 we described different tools to allow the end users to control the liquidity created by MPTCP. However, the use of MPTCP also affects the ISPs offering connectivity to the end user hence, it is natural to ask to what extent the ISP can control the liquidity created by MPTCP. In this section, we try to answer that question by modeling the use of MPTCP by the end user and the reaction by the ISP as a game using game theory. We observe that the ISP can user packet drops as a mean to influence MPTCP behaviour and affect the traffic distribution among paths. We investigate in which scenarios the ISP can benefit from doing so.

4.1 Federated Market and Cloud.net

4.1.1 Federation

The OnApp Federation has been described in detail in Work Package 1 (WP1). In this Deliverable we describe some of the incentive and enforcement mechanisms that have been developed in the course of the third year of Trilogy 2.

4.1.1.1 Enabling Private Federation Via Tokens

A number of customers have requested the ability to enable private resource sharing from the Federation. This has led to the development of a token based system that allows buyers to acquire resources from ‘hidden’ sellers. Resources can only be traded in this ‘private’ market after a token has been successfully entered in the UI interface and authenticated by the market. The seller is responsible for generating the tokens and can then provide these tokens to selected buyers. This allows sellers to control who sees their resources, which are normally visible to everyone who takes part in the Federation.
When sellers add resource zones to the Federation they decide whether they want to make those resources publicly accessible or require a token, as seen in Figure 4.1.

4.1.1.2 Users and roles in the Federation

The main control tool used to control the Federation and Cloud.net is access control lists (ACL). Users of the platform have to be authenticated and are then authorised with certain roles in the platform. The Federated

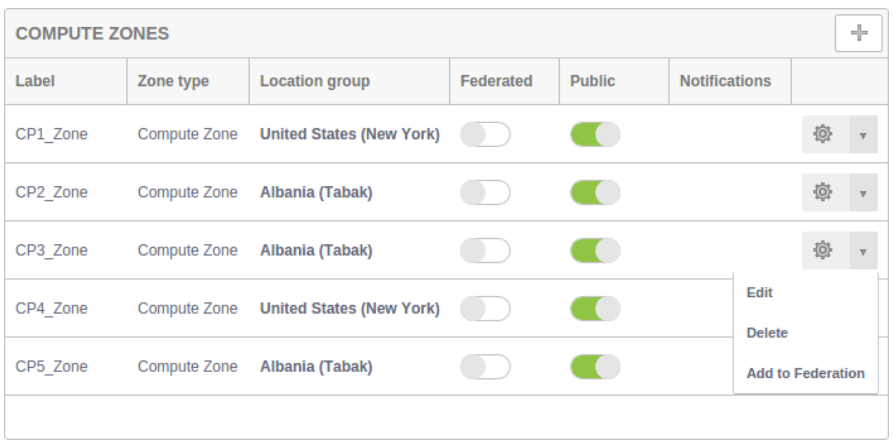


Figure 4.1: Accessibility of cloud resource zones can be controlled through the UI

market maintains a list of Cloud sites and the users who are authorised to perform certain roles in those sites. If an authenticated user makes a valid API request to an authorised cloud then the market will send the relevant commands to that Cloud site. The set of operations that each of the roles can perform is increased to match the new functionality allowed in the Federation.
Recently different types of administrator roles have been added who can manage different parts of the Federation. Support accounts for instance offer one level of administrative functionality that allow support team members to log into the administrative part of the Federation and view and change permissions on a case by case basis. There is also an additional observer type role where an administrator can capture the statistics and then use them for generating reports that show the usage of the Federation.

4.1.1.3 Managing the network for Federated VMs

One of the difficulties with enabling Liquidity for resources is the lack of control of certain types of resources on a remote site. In particular network resources are more complicated as the underlying network topology will not be the same and is often not shared between the buyer and seller. Although not complete, certain network properties can be requested from a remote site and if supported (and allowed) will become part of the remote configuration. On the Federation, new features added include:

- The ability to set up firewall rules for remote VMs
- Managing remote IP addresses from a set of presented IPs
- Adding certain IP addresses to the whitelist for the Federated account

4.1.1.4 Ensuring that users have credit on the platform

As discussed in D1.4 there is a billing system integrated into the Federation that reports the usage activity to a third party billing services provider, Zuora. To ensure that only users with sufficient credit can perform operations the notion of thresholds has been added. The three key thresholds are Positive, Low and Suspended. Buyers and Traders can only create VM resources if their credit balance is Positive. If the credit balance is Low then certain alert messages are sent to the buyers warning that payment is needed. At the point that the credit drops below the Suspended level the VMs belonging to that user are stopped. After a certain time (nominally 30 days, but depends on the type of account) those VMs will be deleted through a clean-up process that runs on the Market.

4.1.2 Tools to control and manage resources in Cloud.net

Cloud.net as described in WP1 is a centralised management platform that runs on the Federation and enables the buying, trading and selling of VM resources, CDN bandwidth and edge access points (PoPs).

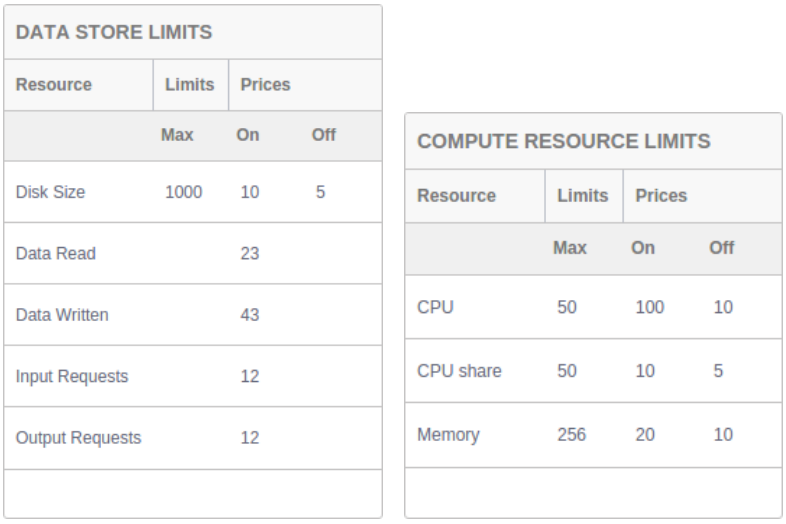
4.1.2.1 Payment in Cloud.net

As a centralised management platform, Cloud.net utilises a different payment system to the Federation. Two different payment models are supported under one converged tracking system known as the wallet. The Pay-as-you-go model allows for VM resources to be paid for from existing credits in the wallet account. To ensure that users don't abuse the platform, a VM creation is only allowed if the customer has over 30 days worth of funds for all existing VMs as well as the new VM. The second billing model is through the credit system. This works as an up-front payment model where 30 days of VM time are billed for at the time of VM creation.
For true liquidity of resources the billing systems have to be tightly controlled such that the seller of resources is assured that they will be paid for the resources used. Currently the 30 day model for billing works from an administrative level but it does not allow for the quick creation and tear-down of resources that will be needed for future uses of Cloud.net.

4.1.2.2 Enforcement activities

As more users have started to use Cloud.net there have been more enforcement policies required to ensure an acceptable level of service for the majority of the customers on the platform.

- Only two IP addresses are allowed on VMs that are not billed to prevent the creation of bespoke networking systems
- VMs that have not been paid for are automatically stopped after three days of warnings



DATA STORE LIMITS

Resource	Limits	Prices	
	Max	On	Off
Disk Size	1000	10	5
Data Read		23	
Data Written		43	
Input Requests		12	
Output Requests		12	

(a) DataStore limits

COMPUTE RESOURCE LIMITS

Resource	Limits	Prices	
	Max	On	Off
CPU	50	100	10
CPU share	50	10	5
Memory	256	20	10

(b) Compute resource limits

Figure 4.2: The OnApp UI shows both the pricing and hard limits on resources

- VMs that have not been paid for are automatically scheduled for deletion after twenty days of warnings

VMs that are stopped or scheduled to be deleted will trigger an admin alert that is then sent to OnApp. Deletion must be performed by an Administrator. This is to give a chance for the sales team to contact the VM owner and see if there was a reason for non-payment. A VM that has not been paid for will use CPU, storage, RAM and network resources until it is stopped. At the interval between 3 days and 20 days of warnings where a VM is stopped but not deleted it will not use CPU, RAM and network resources but it will continue to use up disk space.
Some enforcement actions are controlled through the resource pricing system. Resources can have an active and inactive price as well as a hard constraint. The pricing is managed from both the Buyers and Sellers through 'billing plans'. The wholesale billing plan from a Seller is exposed to potential Buyers whom can then set a billing price for their customers, the end-users. To demonstrate how this appears in the UI an example Data Store billing plan is shown in Figure 4.2a and an example Compute Resource plan is shown in 4.2b.

4.1.2.3 Incentive activities

The main incentive used by Cloud providers to attract customers on Cloud.net, up until recently has been offering comparable services at a cheaper rate than competitors. On Cloud.net the pricing model is currently static with a VM costing the same regardless of whether the VM is idle or running at full capacity and using all the resources available to it. The pricing model also doesn't take into account the load of the servers and the time of day that it is running. Although not currently implemented, dynamic and on-demand pricing models have been considered that take into account the price of electricity during the day and the higher demands put on servers by certain types of workloads. These offerings will initially be made for Software-as-a-Service (SaaS) type services.
Amazon currently does this with spot price instances in a separate model to the rest of AWS. Multiple bidders state the price they are willing to pay for a certain period of compute time and then the highest bidder at any given time will get access to those resources. This allows elastic workloads that can take place at any time to be run on an opportunistic basis. This is mainly suited for large-scale scientific workloads that can be run in isolation and without a requirement on when the operation takes place. However it's believe that most Cloud.net users require high-availability as they generally offer interactive and web-services that need to be accessible at all times.

4.1.3 Tools to control and manage resources in DRaaS

Disaster Recovery as a Service (DRaaS) is one of the main Trilogy 2 use-cases. It has been developed during the course of the project and makes use of the principles of Liquidity to provide a valuable service to customers.

4.1.3.1 Incentives and Enforcement

DRaaS works by copying data in real time from the original VM to a 'shadow' VM on the DR provider's cloud. During normal operation this means that the main resource being used is storage (with a small amount of RAM and CPU used to calculate block hashes, etc.). However once a VM is failed over it then consumes the same amount of RAM, CPU and network as the original VM did.

To stop users from continuing to use the fail over cloud permanently there is as a combination of SLA agreements and punitive billing measures. The contractual agreement in the form of an SLA will normally govern how long a fail over site will continue to provide access to failed over VMs. After this time has elapsed the DR provider can then decide what course of action to take, normally one of: letting the VMs remain in an active mode but with significantly greater charges, stopping them or possibly destroying them. Typically the SLA will allow for a certain period of fail over before the enforcement actions are applied. This incentivises the original cloud owner to try and resolve the issues as soon as possible.

4.2 Operator games in the age of MPTCP

The MPTCP congestion control algorithm attempts to achieve (i) fairness w.r.t. to other (MP)TCP flows utilising the same link and (ii) an efficient use of network resources. To this end, MPTCP will favour the links with the lowest loss rate, pulling traffic away from lossy links as long as the goal to do "no worse than TCP on any path" is satisfied. At the same time, MPTCP will keep some traffic on other links in other to probe for speed changes.

While this approach has a lot of known benefits [52], we observe that it can also be exploited by ISPs (Internet Service Providers): by dropping a fraction of client traffic, an ISP can make MPTCP connections move traffic to some of its competitors. Using its knowledge of the subflow RTT and its client's subscription, the provider can compute a fraction of packets it can drop such that: a) TCP clients' throughput (or MPTCP clients with single uplinks) will not be affected, except for reduced buffering in the network (i.e. reduced bufferbloat, which is a plus), and b) MPTCP clients that have a choice in paths will move their traffic away, thus reducing the operators' upstream bandwidth costs.

In this work, we seek to understand what this policy dropping does to the client-provider ecosystem. Will (m)any providers use policy drop? Will policy drop change the types of subcriptions offered to clients or not? Finally, which subscriptions will Multipath TCP clients choose?

In order to understand the interaction between (MP)TCP clients and their ISPs, we employ a game-theoretic approach. We model this interaction as a two-stage game. In the first stage, providers decide on a unique fixed subscription size, and whether or not they will police traffic (the throughput-altering technique which will be explained in more detail, below). In the second stage, the client choses a subset of providers whose offers maximise his/hers utility.

We are interested in answering the following questions: (i) is traffic policing always in a providers' best interest? (ii) can policing be beneficial for clients? (iii) how does policing affect subscription sizes and provider selection in a free market?

To this end, we study a refinement of the Nash Equilibrium for our game, namely *subgame perfect equilibrium* [36]. This concept determines: (i) the best provider strategy, given that a client will play a "*best response*" — a strategy which maximises utility and (ii) the best client strategy (i.e. the best response to the providers' strategy).

More formally, suppose $\boldsymbol{x} = (x_1, \ldots, x_n)$ corresponds to a subscription size x_i put forward by each provider $i \in \{1, \ldots, n\}$ and $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_n)$ corresponds to a traffic policing decision $\delta_i \in \{0, 1\}$, for each provider. The response of m clients, consists in a $n \times m$-dimensional matrix Q, where the element $q_{ij} \in \{0, 1\}$ represents client j's option for provider i. The matrix $Q(\boldsymbol{x}, \boldsymbol{\delta})$ is a *best client response* iff:

$$\forall j \forall \boldsymbol{q} \in \{0,1\}^n : U_j(\boldsymbol{x}, \boldsymbol{\delta}, Q(\boldsymbol{x}, \boldsymbol{\delta})) \geq U_j(\boldsymbol{x}, \boldsymbol{\delta}, Q(\boldsymbol{x}, \boldsymbol{\delta}) \mid_{j \leftarrow \boldsymbol{q}})$$

where $U_j(\boldsymbol{x}, \boldsymbol{\delta}, Q)$ is client's j utility if $\boldsymbol{x}, \boldsymbol{\delta}$ and Q are played, and $Q \mid_{j \leftarrow \boldsymbol{q}}$ is the matrix Q where column j has been replaced by $\boldsymbol{q}$. Thus, a best response for all clients, is a matrix where no client j can switch to a provider selection $\boldsymbol{q}$ which ensures strictly higher utility.

A pair $\boldsymbol{x}, \boldsymbol{\delta}$ is a *best provider move* iff:

$$\forall i \forall x_i \forall \delta_i : \mathcal{U}_i(\boldsymbol{x}, \boldsymbol{\delta}, Q(\boldsymbol{x}, \boldsymbol{\delta})) \geq \mathcal{U}_i(\boldsymbol{x} \mid_{i \leftarrow x_i}, \boldsymbol{\delta} \mid_{i \leftarrow \delta_i}, Q(\boldsymbol{x}, \boldsymbol{\delta}))$$

where $\mathcal{U}_i(\boldsymbol{x}, \boldsymbol{\delta}, Q)$ is provider's i utility if $\boldsymbol{x}, \boldsymbol{\delta}$ and Q are played, and $\boldsymbol{x}\ |_{i \leftarrow x_i}$ (resp. $\boldsymbol{\delta}\ |_{i \leftarrow \delta_i}$) is $\boldsymbol{x}$ (resp. $\boldsymbol{\delta}$) where component i is replaced by x_i (resp. δ_i). Thus, a best provider move is one where no provider i can select a different subscription size x_i or policing decision δ_i such that his/hers utility is maximised *given that clients will play their best response*.
A triple $\boldsymbol{x}, \boldsymbol{\delta}, Q$ is a subgame perfect equilibrium iff (i) $\boldsymbol{x}, \boldsymbol{\delta}$ are best provider moves and (ii) Q is a best client response to $\boldsymbol{x}.\boldsymbol{\delta}$.
In this section, we build a general (cost-independent) utility model for our game. In the next section, we instantiate our model starting from observations regarding prices in the Romanian ISP market, and review our results.
Our utility model relies on the following main ingredients: *prices*, *link availability*, and total throughput when *traffic policing is performed*.

4.2.1 Prices

Suppose a provider secures a x-Mbps uplink which must be split among potential clients. We denote by $\mathbf{c}(x)$ the cost per Mbps of such an uplink, and by $\mathbf{tr}(x)$ — the *total traffic cost* of carrying x Mbps (at cost $\mathbf{c}(x)$). We argue that any modelling choice of $\mathbf{c}$ must make:

$$\mathbf{tr}(x + y) \leq \mathbf{tr}(x) + \mathbf{tr}(y) \quad \forall x, y \geq 0 \tag{4.1}$$

i.e. the transit cost of a bigger $x + y$ Mbps subscription is at least as expensive as that of two smaller x and y subscriptions.

4.2.2 Availability

Availability is the ratio between the uptime and the up- and down- times of the Internet link. When a client owns n subscriptions each offering the same availability α , the availability of the combined links is $1 - (1 - \alpha)^n$. We argue that a client will be more sensitive to *the uptime probability given that one provider link fails*, rather than availability, as defined above. We call the former *perceived availability* and define it as: $av_\alpha : \mathbb{N} \to [0, 1]$, which maps a number n of providers to a value:

$$av_\alpha(n) = 10 \cdot (1 - \alpha)(1 - (1 - \alpha)^{n-1}) + 0.5$$

Note that $(1 - \alpha)(1 - (1 - \alpha)^{n-1})$ is the probability of one provider failing $(1 - \alpha)$ given that all others do not fail $(1 - (1 - \alpha)^{n-1})$.

4.2.3 Throughput and traffic policing

If a client uses MPTCP while connected to two providers offering x resp. y Mbps subscriptions, it is possible for the providers to artificially decrease maximum throughput from $x + y$ to $\max\{x, y\}$. This is achieved by fixing a dropping rate δ, when traffic exceeds a certain threshold. The effects and traffic distribution are illustrated in the following table (where $x \geq y$).

Provider 1	*Provider 2*	*Max. throughput*	P_1 *carried traffic*	P_2 *carried traffic*
does not police	*does not police*	$x + y$	x	y
polices	*does not police*	x	$x - y$	y
does not police	*polices*	x	x	0
polices	*polices*	x	$x\frac{\delta_y}{\delta_x + \delta_y}$	$x\frac{\delta_x}{\delta_x + \delta_y}$

We defer a discussion on δ selection, for the following section. We also note that the throughput of a TCP client will not be affected by traffic policing: in this case, the perceived throughput will always be equal to the subscription size.

4.2.4 Client & provider utility

Suppose providers have made the move $\boldsymbol{x}, \boldsymbol{\delta}$ in the first stage of the game and that client j has selected a set $\mathcal{S} \subseteq \{1, \ldots, n\}$ of providers. Let t_{ij} be the bandwidth which is perceived by client j from provider i. We note that t_{ij} may not be equal to x_j for $j \in \mathcal{S}$, for MPTCP clients: providers may use *traffic policing*. The utility of client j is:

$$U_j = \begin{cases} av(|\mathcal{S}|) \cdot \beta \cdot \log(1 + \max_{i \in \mathcal{S}} t_{ij}) - \sum_{i \in \mathcal{S}} price(t_{ij}) & type(j) = TCP \\ av(|\mathcal{S}|) \cdot \beta \cdot \log(1 + \sum_{i \in \mathcal{S}} t_{ij}) - \sum_{i \in \mathcal{S}} price(t_{ij}) & type(j) = MPTCP \end{cases}$$

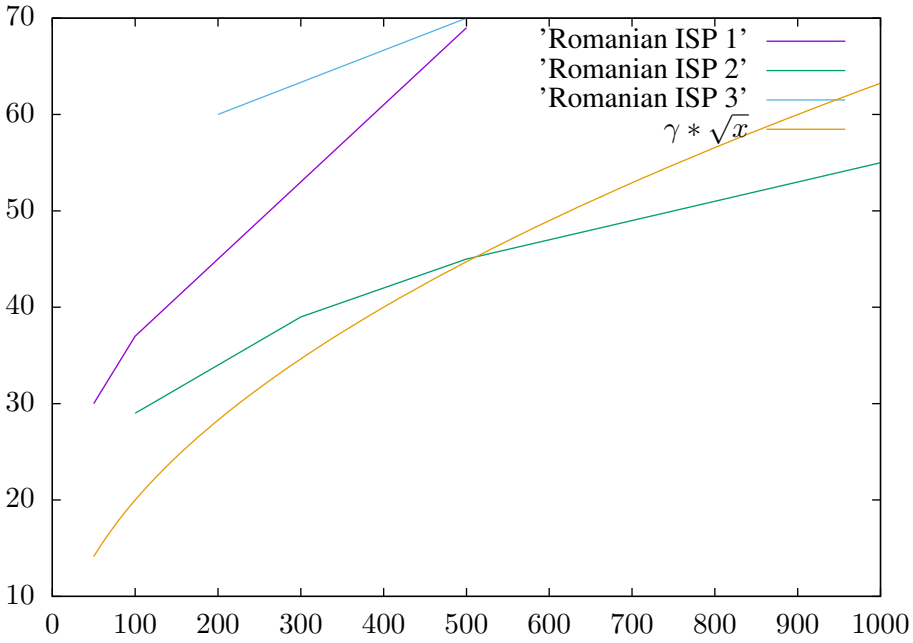


Figure 4.3: Our deduced cost function compared to Internet subscription prices in Romania (on the X axis: the subscription expressed in Mbps; on the Y axis: the price expressed in RON)

The positive component of the utility captures: (i) client satisfaction from the obtained throughput (the logarithmic expression), (ii) perceived availability, (iii) a weight β which models the clients sensitivity to all the above. The negative component is the sum of all subscription prices. We shall assume that *prices are fixed at marginal cost*, i.e. each provider will charge for throughput t exactly his cost for carrying t (and not the official subscription x). Note that this cost depends on the number of subscribers as well as the subscription size. The provider utility $\mathcal{U}_i$ simply represents the number of clients which selected provider i.
We use the following functions for estimating **c** and **tr**:

$$\mathbf{c}(x) = \gamma \cdot \frac{1}{\sqrt{x}} \quad \mathbf{tr}(x) = \gamma \cdot \sqrt{x}$$

with $\gamma = 23.05$ (fitted using our dataset). The plot in Fig. 4.3 compares the square root function with subscription costs from Romanian ISPs:

Proposition 4.1. *Any type of client will prefer buying two subscriptions instead of more.*

We skip the formal definition and give a simple intuition for it: client satisfaction increases logarithmically with respect to the number of providers, while prices grow linearly with respect to the same number. Also, the increase in perceived availability is major when going from one provider to two, and small — when going from two providers to more. Thus, only for a "two-provider" selection, the client satisfaction increase dominates the increase in price.

4.2.5 Throughput and shaping

In the previous section, we noted that perceived throughput may be different from the actual subscription size, if providers deploy traffic policing techniques. In [12], the policing upper limit δ_x, defined w.r.t. to the RTT and subscription size x, as follows:

$$\delta_x = \frac{1}{2}\left(\frac{50x^2}{RTT(RTT - 50x)}\right)^2$$

was used to successfully deter MPTCP from using a lower-quality link. We have observed via simulation that the same upper limit δ_x applies in our case.
We have implemented this game and performed an initial analysis of the results. These are presented next.

4.2.6 Initial results

In this section we present an initial exploration of the game between operators and clients in the age of Multipath TCP. Our aim is to understand what effects Multipath TCP and policy dropping will have on the Internet ecosystem.

We have implemented a prototype solver for our game model, which exhaustively searches for subgame perfect equilibria. We have limited subscription sizes to a few discrete alternatives (e.g. 3, 5, and 50 Mbps). The implementation makes an additional refinement of our theoretical model: it considers that providers have *imperfect information* over client types. This is modelled, for each provider i, by a function p_i which assigns to each client j the probability $p_i(j)$ that he employs MPTCP.

Thus, when evaluating their best move, each provider will compute the *expected client utility*, according to his information over types:

$$EU_j = p_i(j) * U_j^{type(j)=MPTCP} + (1 - p_i(j)) * U_j^{type(j)=TCP}$$

Our initial results show that:

- clients will always choose a high-subscription (HS) provider and a low-subscription (LS) provider, due to optimal cost distribution. We can view the low-subscription price as a "*price for availability*".
- if a HS and LS provider both shape, the effects on traffic (re)distribution are small.
- a LS-provider may use policing to attract clients: by reducing traffic, he/she is giving a cheaper subscription (i.e. a cheaper *price for availability*).
- HS providers can form an oligopoly and make a joint decision to police traffic — they will still be selected by clients.

We still have ongoing work regarding how imperfect information may alter the providers' best move. We are also considering more elaborate cost models (i.e. by also incorporating a connectivity charge with the bandwidth cost).

5 Tools for Understanding Liquidity

Understanding some characteristics of the resources to be pooled is fundamental to properly control them. In the previous sections we have presented different tools to control liquidity. In this section, we present tools for understand some key aspects of liquidity.
We start by describing the Web dependency graph analyser. This tool analyses the content of modern web pages, parsing all its components. The purpose of the tool is to understand whether and how modern web traffic can benefit from the different liquidity tools. It tries to provide some insight on whether web traffic would benefit from multipath transport such as MPTCP and if so, what is the correspondence between web page components and MPTCP subflows. Other liquidity tools, such as compute resource pooling or storage pooling can also benefits from the perspective provided by the graph analyser when trying to distribute compute or content across multiple servers.
The second tool for understanding liquidity we cover in this section is Symnet, a framework for symbolic execution of network functions. When widely deployed, liquidity tools such as NFV, make the network infrastructure a much more dynamic environment. Indeed, through NFV an operator can deploy and compose different network functions in very short timescales. While this provides a lot of flexibility, it also makes the network much harder to understand, predict and reason about. Symnet provides a symbolic execution framework that allows to explore the possible (execution) paths that different types of packets can go through a network, providing valuable insight about the behaviour of the network.

5.1 Web dependency graph analyser

In their early inception, web pages were simple, text based entities. They have long since evolved into their modern day form: complex, dynamic and rich in images and media. Today most web pages are still downloaded using HTTP/1.1 [20], a protocol which has changed little since the late nineties, and was not designed to deal with the dozens of inter-dependent objects these pages request. Inherently a pull based protocol operating over TCP, HTTP/1.1 can only download one object at a time over any TCP connection. Browsers attempt to speed up a page's overall completion time by opening a larger number of persistent TCP connections (6 in modern browsers) and using pipelining to download objects. Nevertheless, HTTP/1.1 still suffers from Head of Line (HOL) blocking, where downloading one object on a slow connection will stall all subsequent objects until it completes.

Recent years have seen the emergence of next generation web protocols that aim to tackle HTTP/1.1's shortcomings, with particular focus on minimising page load times. Of these, HTTP/2[6] (largely based on Google's SPDY[5, 48]) has recently been standardised. It provides a multiplexing layer over TCP, where each HTTP request/response pair forms an independent stream, allowing the browser to download multiple objects simultaneously over one TCP connection. Google has also experimented with a UDP-based transport protocol called QUIC[41] which moves the multiplexing mechanics of SPDY into the transport layer in order to prevent HOL at the transport level.

HTTP/1.1 and HTTP/2.0 were both designed to run over TCP. However, even over single-path TCP, there is not a good understanding of how the dependencies within a modern web page interact with the transport protocol. How might such protocols interact with Multipath TCP, when multiple paths are available with differing bandwidths and latencies? Does it always make sense for MPTCP to use more than one path to gain additional bandwidth, even if the latencies differ significantly, or are today's web pages so latency-bounded that using MPTCP might *increase* page load times? We have anecdotal evidence that shows this can indeed be the case. What if the transport protocol (MPTCP) could be aware of the web objects it is transporting - or conversely, if the web server were MPTCP aware, and could request some parts of the data are more latency-sensitive than others? Such a server should be able to achieve a much more effective trade-off between latency and throughput by careful placing of traffic onto subflows, minimizing page-load time. Taking this a step further, a multipath version of QUIC could perform such optimization on a per-packet basis - how much of a win might this be compared to TCP or MPTCP?
It is extremely hard to answer these sorts of questions. Ideally we'd just implement our designs and test on real web pages. Unfortunately, performing this evaluation using physical devices is infeasible. Not only

would we need to modify the client, but we must modify all content servers to understand the protocol, a feat that cannot be realised unless we are the content providers ourselves. Instead, we have built a simulation framework that will allow us to quickly prototype and evaluate application/transport protocol optimization using models of the most popular web pages.

It is completely inaccurate to model web pages as a series of consecutive resource requests. The page load process is much more elaborate: browsers mix the fetching of web resources with parsing and evaluating JavaScript and CSS. These activities are interrelated and create inherent dependencies amongst themselves. On a similar note, an accurate web dependency representation is necessary to communicate useful cross-layer information to the protocol. If the protocol wants to understand which web resources are delay-sensitive, it must understand the relationships between the various page load activities. To allow us to perform this analysis of transport protocol dependencies we have implemented a Web Dependency Graph Analyser, which allows us to correctly model web pages and better understand possible areas of latency improvement using new protocols that exploit this tradeoff between bandwidth and latency liquidity.

5.1.1 Web Page Load Process

Our goal in building the Web Dependency Graph analyser is to accurately model the inherent dependencies within web pages. We must therefore understand the underlying activities browsers perform to download and render these pages. In essence, a web page is a collection of resources, like images, text and media, whose structure and appearance are concisely specified by the page's main HTML file. The browser downloads the resources, and transforms the HTML description into the document object model, or DOM tree. Web page developers can use CSS and JavaScript to manipulate the DOM dynamically. With the constructed DOM as an intermediate representation, the browser renders the page for the client to see.
Suppose a client navigates to the website described by the HTML code in Figure 5.1. After the browser sends the initial request and downloads the main HTML file, its following components work together to display the web site:

- **Parser**: reads and tokenises the HTML tags in the main page as well as any iframes enclosed within the page. The Parser does not wait for the entire page to download, rather it begins operation when the first chunk of the web page arrives over the network. As it runs through the HTML tags, the parser builds the DOM tree. If the parser encounters an HTML tag that references an embedded resource, such as the `<link>` tag on line *23* in this example, it triggers its download via the Resource Loader.

- **Resource Loader**: is responsible for requesting and downloading the various resources referenced in the web page if they are not found in the cache. The resources can have different MIME types, like images, JavaScript or CSS files, iframes and other media. The Resource Loader requests each resource using an HTTP variant as the underlying protocol, although new Chrome browsers sometimes use QUIC.

- **Evaluation Core**: composed of the JavaScript engine, and CSS evaluator, executes any inline or external scripts. After the parser encounters the `<script>` tag in our example, and requests the referenced *.js* file using the Resource Loader, it is the Evaluation Core which executes the script after its download completes. In addition to evaluating inline or external JavaScript code, the core also evaluates JavaScript events and timer code.

- **Renderer**: uses the DOM to display the web page progressively in the browser. Since the rendering process is largely implementation dependent, and does not greatly impact a page's load time, the rest of the description will focus on the above three processes.

The activities performed by the Parser, Evaluation Core and Renderer are computation processes, whereas the Resource Loader performs the only network process. In theory, a browser can possibly execute the download, parsing and computation processes in parallel, reducing a page's load time to that of its slowest activity. Reality, on the other hand, necessitates the execution of some activities before others, therefore adding dependencies between them. Some dependencies are flow based: the parser, for example, cannot process an HTML tag until the chunk containing this tag arrives over the network. Others are serialisation


```
<html>
  <head>
    <script>
      function downloadImage()
      {
        //Create a new Image
        var anImage = new Image();
        anImage.setAttribute("src", "rainbow.jpg");

        //Append it to the div tag
        var element = document.getElementById("firstDiv");
        element.appendChild(anImage);
      }
      function fireTimer(){
        setTimeout(downloadImage, 10);
      }

    function loadFrame() {
      fireTimer();
    }
    </script>
  </head>
  <body onload=loadFrame()>
    <link rel="stylesheet" type="text/css" href="theme.css"></link>
    <div id="firstDiv">
      <p2> Some Paragraph </p2>
    </div>
    <script src="image_create_sea_append.js"></script>
    <div id="secondDiv">
     <p> Some more text </p>
    </div>
  </body>
</html>
```

Figure 5.1: An example HTML file

artefacts, for example the Parser and Evaluation Core cannot run at the same time, because they both modify a shared data structure, the DOM tree. To model web page loads correctly, we quantify these dependencies, and generate a faithful dependency graph between the various browser activities.

5.1.2 Dependency Graph Activities

The following is a formal definition of the web page activities, and how they relate to each other within the dependency graph. We say $a \rightarrow b$ when a is the parent of b in the dependency graph, or more concretely b cannot execute unless all of the activities represented by a have completed. Table 5.1 summarises the type of activities and their dependency relationships ($a \rightarrow b$ indicates that a is a parent of b. The expression $a \vee b$ indicates that events of type a or b are a possible parent. The expression $a \wedge b$ indicates that a and b must both complete before their child can execute).

- **Resource** represents a network download activity, where the Resource Loader sends a request for a particular MIME object to a remote server, and waits for the complete response. A resource download depends on parsing the specific HTML tag referencing it. Conversely, it may depend on executing a computation (evaluation, timer or event) which programatically triggers the download. In our example, the JavaScript in line *7* creates an image element, which in turn causes the Resource download. Evaluating CSS scripts may trigger font or image downloads as well.

- **Parse Chunk** models a unit of computation with duration equal to the parse time until the next resource download or inline script declaration. The parser does not usually consume an HTML frame in one continuous pass, but rather pauses to evaluate inline scripts, or yields to allow other scripts to execute. Therefore, we can split the parsing of a contiguous HTML file into several logical chunks, whose

boundaries occur at HTML tags that trigger downloads, or inline script evaluations. In our example, Figure 5.1 shows the parse chunks marked with the arrows on the left hand side.
A Parse Chunk's dependencies are flow based. In order to execute a Parse Chunk, the browser must download the corresponding HTML file chunk first, and execute the previous Parse Chunk. Any synchronous JavaScript evaluated as the result of the prior Parse Chunk will also block parsing, and therefore is a parent.

- **Download Chunk** represents the download of an HTML file chunk corresponding to a particular Parse Chunk. A Download Chunk depends on the completion of the pervious chunk.

- **Computation** is an activity executed by the browser's Evaluation Core with the ability to modify the DOM elements dynamically. Like Parse Chunks, we will subdivide computations along the Resource download boundaries to allow for easier simulation. Four different types of computation activities exist:

  (i) Script Eval: a JavaScript evaluation activity. JavaScript can either be inline, or require the download of an external *.js* file. Inline JavaScript executes synchronously, blocking the next Parse Chunk. External JavaScript, on the other hand, cannot execute until its Resource completes downloading. Developers may set one of three attributes to specify when an external script should execute: *sync, async* and *defer*. The default *sync* attribute indicates that the JavaScript file should complete downloading and executing before the parser can continue on to the next tag. JavaScript marked with the *async* attribute, allows the resource to load without blocking parsing. Once the JavaScript's resource download completes, the browser can evaluate the script asynchronously when it sees fit. Finally, a developer may choose to use the *defer* attribute to delay the evaluation of the JavaScript until the main frame has finished parsing. Because both the CSS and JavaScript evaluations modify the DOM, most browsers will prevent any future JavaScript from executing until previous CSS files have completed downloading and have been executed. Script elements may be created by other JavaScript functions, therefore Timers, Event and other Script Evals can be parents of the current Script Eval.

  (ii) CSS Eval: represents the evaluation of a *.css* file to generate the style rules applied to the DOM. Evaluating a *.css* file depends on downloading it first.

  (iii) Timers: are declared within JavaScript functions and schedule a function for execution at a specified interval in the future. When a timer fires, its evaluated function may trigger further activities, like Resource downloads. A Timer activity depends on evaluating the JavaScript that registered it, i.e. either a Script Eval or Event.

  (iv) Events represent the evaluation of a JavaScript event handler executed when various events fire during the web page load process. Typical events include the DOMContentLoaded event, which fires after parsing of a frame completes. The Load Event fires when a frame has completed parsing, and has downloaded all of its resources and iframes. Due to the numerous event types exposed by JavaScript, we will refrain from listing all of the possible JavaScript event types and which parents they depend on in this document.

5.1.3 Implementation

In the previous sections, we've defined the activities performed when a web page loads, and the Web Dependency Graph linking these activities together. This section will describe how we record these dependencies for real web pages and replay them in our simulator. To record real web page information, we have extended WProf developed by Wang et. al [51], a profiler for WebKit used by the Chrome browser. WProf inserts hooks into the open-source WebKit code, and uses them to log resource requests, parsed HTML tokens, and executed computations. A log entry indicates an activity's immediate parent, and some timing information, to allow for the calculation the computation and parse activities's duration.

We have augmented WProf to give us a more complete picture of the page load process. The following is a list of the most important changes we've incorporated into the profiler:

Parent Dependency	Activity
Parse Chunk $\vee$ Computation	$\rightarrow$ **Resource**
Parse Chunk $_{n-1} \wedge$ Download Chunk $_n$ Script Eval $\wedge$ Parse Chunk $_{n-1} \wedge$ Download Chunk $_n$	$\rightarrow$ **Parse Chunk** $_n$
Resource	$\rightarrow$ **Download Chunk** $_0$
Download Chunk $_{n-1}$	$\rightarrow$ **Download Chunk** $_n$
Resource $\vee$ Parse Chunk $\vee$ Timer $\vee$ Event $\vee$ Script Eval CSS Eval $\wedge$ (Resource $\vee$ Parse Chunk $\vee$ Timer $\vee$ Event $\vee$ Script Eval)	$\rightarrow$ **Script Eval**
Resource	$\rightarrow$ **CSS Eval**
Script Eval $\vee$ Event	$\rightarrow$ **Timer**
Event specific	$\rightarrow$ **Event**

Table 5.1: Summary of dependency graph activities and their possible parents

- Logging inline JavaScript executions.
- Logging programatic DOM elements created from JavaScript. The change allows for more accurate logging of all the resources downloaded from within JavaScript executions.
- Recoding all activities even after the page Load Event fires. A crucial change, since many resource loads actually occur as a result of executing the Load Event, and its children.
- Logging Timers registered from JavaScript functions, and when the timer fires.
- Recording Events and when their handlers are executed. We log the event names, in addition to their targets. An Event target can either be an HTML tag, for example a load event called from an image tag, indicates that the load event's parent is the downloaded image Resource. The target may also be the JavaScript Document or Window objects, as is the case for the page load event.
- Distinguishing between parsing activities generated by the parser itself, or by programatically evaluating Document.write() in a JavaScript function. The distinction allows us to discount programatically generated tokens from the Parse Chunks.
- Attaching frame identifiers to each activity. Computations, resources and parse tag may belong to the page's main frame, or any one of the downloaded iframes within the page. Therefore these activities depend on loading the frame itself.

With the modified Chrome browser, we obtained logs for 60 of the Alexa top 500 websites [3], using the following procedure: for each web page, we navigate to that particular page and then wait 20 seconds after the page loads to capture any activities that occur after the load event, then navigate to the next page. We passed the raw logs to a post-processing python script, which iterates through each traced activity and generates the dependency graphs in JSON format. The script also generates a pdf representation of the graph, shown here in the figures.

Recall our example web page in Figure 5.1. After profiling the web page and constructing its dependency graph using the mechanism above, we have obtained the result in Figure 5.3. We can see that the graph matches our expectations and the dependency rules we have outlined in section 5.1.2. For example, the inline JavaScript depends on the Parse Chunk containing its `<script>` tag. The *.js* and *.css* resources depends on their respective Parse Chunks, and they trigger the evaluation of their content.

Figure 5.2 shows the result of applying our analysis tool on Amazon's home page. The figure is a 100 foot view, intended only to illustrate just how complex modern web pages and their dependencies have become. Amazon relies heavily on events and timers to download the images displayed in the website, which appear in the figure as the side graph branching off from the vertical main page.

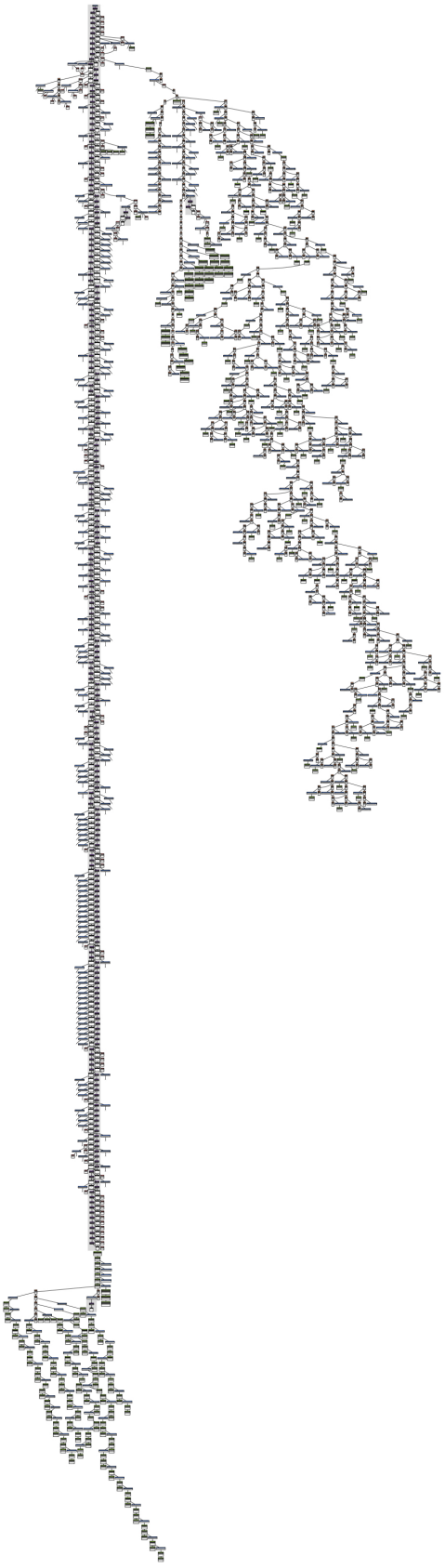


Figure 5.2: Zoomed out view of dependency graph of www.amazon.com.

With the dependency graphs in hand, we use the ns-3 network simulator [35] as the framework for reproducing a web page load, and examining its behaviour with different underlying web protocols. We've

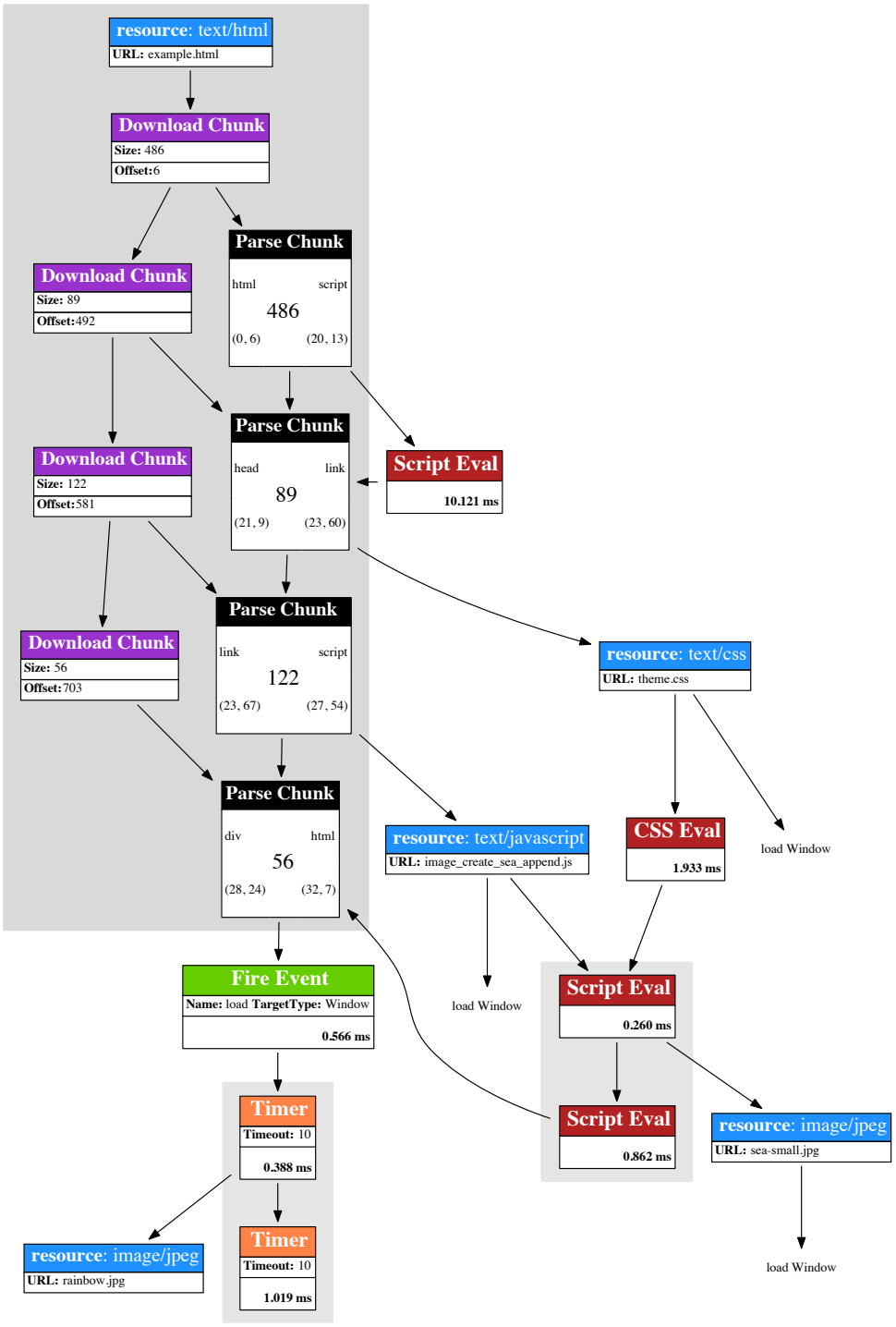


Figure 5.3: Dependency graph resulting for analysing the example HTML page. Grey boxes indicate a single frame, or a computation consisting the several sub-computations. Parse Chunks show their size, and the start and end HTML (row, column) pairs for the chunk. Computations display their duration.

added a Web Dependency module to ns-3, which simulates a web client communicating over a single switch with multiple servers (corresponding to the unique domains logged in our graphs). The web client consumes a profiled web page's dependency graph, and performs the correct actions: it waits the recorded time for each computation or parse chunk, and issues the requests for resource activities as necessary. Each time an activity in the graph completes, the client attempts to execute all of its children. A child is only executed

when all of its parents have completed. The simulation uses plugable web and transport protocols to issue the resource requests and receive responses over the network, such that we can substitute in prototype web protocols with ease.

Going back to our example from Figure 5.1, after simulating its dependency graph running over HTTP/1.0, with links of 1ms delay and 10 Mbps bandwidth, we obtain the waterfall plot shown in Figure 5.4. Since HTTP/1.0 creates a new TCP connection for every resource request, it incurs the overhead of the 3-way handshake, shown in the figure as grey wait times. The figure also shows that the activity labeled *comp_3*, corresponding to the page load event, cannot execute until the final resource, in this case "sea-small.jpg", completes downloading.

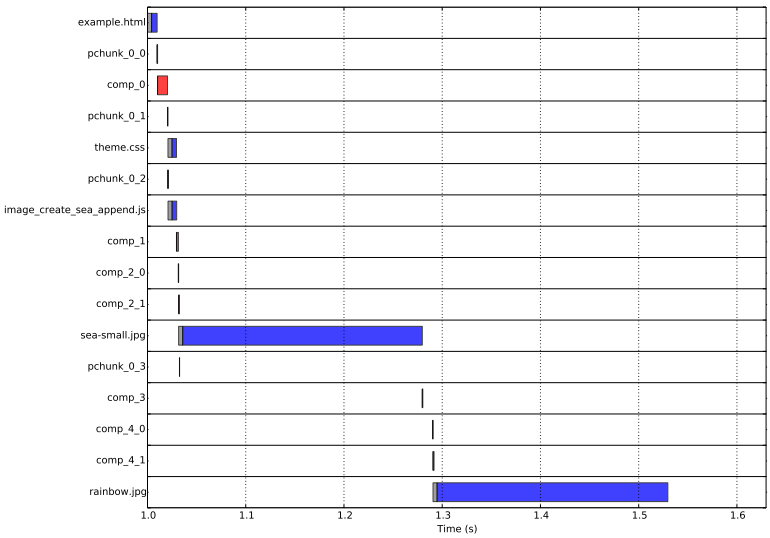


Figure 5.4: Waterfall diagram of downloading example.html over HTTP/1.0. Wait times are shown in grey.

To sum up, we've developed an accurate Web Dependency Graph Analyser, composed of a profiler and simulation framework. The analyser allows us to accurately model the activities involved in the page load process for real web sites. Furthermore, it provides a framework with which we can simulate loading these web sites over new multipath web protocols, in order to understand their behaviour in the context of the Liquid Net. Our code and some generated dependency graphs will be available as open source software for other researchers to use to evaluate web performance.

5.2 Symbolic execution for networks with Symnet

Modern networks deploy a mix of traditional switches and routers alongside more complex network functions including security appliances, NATs and tunnel endpoints. Understanding end-to-end properties such as TCP reachability is difficult before deploying the network configuration, and deployment can disrupt live traffic. Static analysis of network dataplanes allows cheap, fast and exhaustive verification of deployed networks for packet reachability, absence of loops, bidirectional forwarding, etc. All static analysis tools take as input a *model* of each network box processing, the links between boxes and a snapshot of the forwarding state, and are able to answer queries about the network without resorting to dynamic testing [53, 27, 31, 32, 39].

What is the best modeling language for networks? If possible, we should simply use the implementation of network boxes (e.g. a C program), as this is the most accurate and is easiest to use. If we view packets as variables being passed between different network boxes, static network analysis becomes akin to software testing. This is a problem that has been studied for decades, and the leading approach is to use *symbolic execution* [9].

Symbolic execution is powerful: it explores all possible paths through the program, providing possible values for each (symbolic) variable at every point. For static network analysis, the power of symbolic execution lies in its ability to relate the outgoing packets to the incoming ones: even if all the incoming packet headers are unknown, a symbolic execution engine can detect which parts of the packet are invariant through the network,

and can tell *how* the modified headers depend on the input when they are changed. Unfortunately, symbolic execution scales poorly: its complexity is roughly exponential in the number of branching instructions (e.g. "if" conditionals) in the analyzed program. Applying symbolic execution to actual network code quickly leads to untenable execution times, as shown in [17]. To cut complexity, we must run symbolic execution on models of the code, rather than the code itself. It is natural to program the models also in C, as previous works do [17, 2].

In this section we show how we can run symbolic execution on a snapshot of the network to understand a range of network properties that considerably simplify the job of network operators. The key novelty in our work is SEFL, a language we have developed to model network processing in a way that is amenable to fast symbolic execution. We have also developed Symnet, a tool that performs symbolic execution on SEFL models of network boxes, rather than their actual code. By relying on SEFL our tool can scale to large networks.

The remaining challenge is to accurately model network functionality in SEFL; this process is not trivial, and requires expert input. We have manually modeled a large subset of the elements from the Click modular router suite [28]. Our click models are particularly useful to third parties: they allow static verification of any modular router configurations built using existing Click elements. Finally, we have developed a testing tool that takes SEFL models and their runnable counterparts and checks that the model conforms to the actual implementation.

To evaluate Symnet, we have applied symbolic execution to understand a series of documented middlebox interactions[29, 25] and the Stanford backbone. Our results show that Symnet is more powerful than existing static analysis tools, captures most real-life interactions in networks with runtimes in the order or seconds.

5.2.1 Motivating examples

Static analysis tools are enticing because they can help network operators understand the operation of their deployed networks and they can inform the correct deployment of updates. Static network analysis is maturing: tools such as Header Space Analysis [27] and Network Optimized Datalog [31] have evolved from research and are now being rolled into production. However, all existing tools have limitations, as they choose different points in the tradeoff between the expressiveness of the policy specification language, the network modeling language, the ease of modeling and the checking speed. Furthermore, existing tools can't model widely used functions such as dynamic tunneling and encryption.

In this section we discuss three examples of network functionalities we want to statically analyze and highlight the difficulty in using existing tools for this purpose.

Modeling tunnels. Various forms of tunnelling are in widespread use in networks today, sometimes deployed by different parties. Can we statically analyze such networks? We provide a simple example below, where E1 and E2 perform IP-in-IP encapsulation and D1 and D2 the corresponding decapsulation.

$$A \rightarrow E1 \rightarrow E2 \longrightarrow D2 \rightarrow D1 \rightarrow B$$

Consider Header Space Analysis [27] (HSA), the most mature network static analysis tool today. With HSA, the packet header is modeled as a sequence of bits, where each bit can take values 0.1 or $*$ (don't care). Network functions are modeled as transformations of the packet header. For instance, the IP-in-IP encapsulation will be modeled as a 20 Byte shifting of existing bits to the right and adding the new IP header in the remaining space; the decapsulation will perform the inverse operation. Beyond reachability, we want to statically answer the following basic question: are packet contents invariant across this tunnel? The answer is obviously yes, but HSA cannot capture it: if the input header contains only * bits, the output will also contain $*$ bits, but this does not imply that individual packet contents may not change. We can always feed a specific packet to the model and check it is not modified, but to ensure the invariant holds in general we must try all possible packets—this won't scale. A symbolic packet is needed instead—if the symbolic packet doesn't change, the property holds regardless of its value.

We have also modeled the simple tunnel using the newer Network Optimized Datalog tool [31]. NOD can compute the invariant, but modeling is cumbersome and limiting in many ways. First, the models for D1 and D2 differ, despite the fact they are running the same operation. D2 takes a packet with six header fields (to "remember" the two layers of encapsulation): we cannot reuse the D1 model for D2, nor the one from E1 for E2: we need to create a new model instead. In fact, network models in NOD depend not only on the processing of the box, but also *the network topology and the processing of other boxes*. Additionally, models

```
int crt = 0;
while (crt>=0 && crt<length &&
                    options[crt]){
    switch(options[crt]){
        case 1:
            crt++; break;
        case 2://MSS
        case 3://WINDOW SCALE
        case 4://SACK PERMITTED
        case 8://TIMESTAMP
            crt += options[crt+1]; break;
        default:
            //unknown options, scrub
            int len = options[crt+1];
            for (i=crt;i<crt+len;i++)
                options[i] = 1;
            crt += len; break;
    }
}
```

Figure 5.5: TCP Options processing code for a middlebox that drops unknown options.

for boxes operating at a lower stack level must also include higher level protocol headers: for instance, a router will model not only its use of the layer 3 fields, but also the upper layer protocols such TCP, UDP or ICMP. In summary, NOD modeling is extremely cumbersome when network topologies are heterogeneous and run multiple protocols; this may not be an issue for datacenters where NOD's usage is targeted, but it will be a major issue in the wider Internet.

In Section 5.2.6 we show how symbolic execution can be used to model dynamic tunneling. However, the following examples show the limitations of running standard symbolic execution on C code.

IP Router. Dobrescu et al [17] show that symbolically executing an IP router implemented in C quickly becomes intractable: when a packet with a symbolic destination address reaches the router, the branching factor is at least as large as the number of prefixes. Such branching is prohibitive when analyzing core routers that have hundreds of thousands of prefixes in their forwarding tables.

To make symbolic execution tractable, we would like the branching factor to depend on the number of links of the router instead; this is feasible, but we must write an optimized router model for symbolic execution (see Section 5.2.6).

Parsing TCP Options. Assume a network operator has deployed a middlebox and wishes to know what options are allowed through in its current configuration; in particular, it wishes to understand whether a new IETF transport protocol might work in its network.

In Figure 5.5 we show a C code snippet that processes the TCP options header field. The options field is accessible via the `options` character array, and contains `length` bytes. The middlebox allows a number of widely used TCP options, and drops all other options by replacing them with padding.

HSA or NOD can't model this example, but the operator could run *klee* on the middlebox code instead, providing a symbolic `options` field. By examining all the paths resulting from the symbolic execution engine, the operator can tell, in principle, which options are allowed through and which not. In the table below we present the number of resulting paths and runtime of *klee*, as we vary the length parameter. The results below show just how costly symbolic execution on C code is, even on fairly simple code snippets.

Length	1	2	3	4	5	6
Number of paths	4	67	140	464	1095	3081
Runtime (s)	0.3	8	20	420	1500	9120

Towards a Solution

When applying symbolic execution to C code, the number of branches in the code exponentially increases the number of paths to be explored. To make symbolic execution feasible, we need to drastically simplify the code being symbolically executed. Unfolding loops and executing both branches of an "if" instruction are techniques that reduce the complexity of symbolic execution at the cost of increased runtime [50]. Such techniques, together with simpler data structures allow verifying small pipelines of Click modular router elements (ten or less) in minutes, as shown by Dobrescu et al. [17]. Our target scale is two orders of magnitude higher: we aim to verify networks containing hundreds or more elements in seconds.

Instruction	*Description*
`Allocate(v[,s,m])`	Allocates new stack for variable v, of size s. If v is a string, the allocation is handled as metadata and the optional m parameter controls its visibility: it can be global (default) or local to the current module. If v is an integer it is allocated in the packet header at the given address; size is mandatory.
`Deallocate(v[,s])`	Destroys the topmost stack of variable v; if provided, the size s is checked against the allocated size of v. The execution path fails when the sizes differ or there is no stack allocated for variable v.
`Assign(v,e)`	Symbolically evaluates expression e and assigns the result to variable v. All constraints applying to variable v in the current execution path are cleared.
`CreateTag(t,e)`	Creates tag t and sets its value e, where e must evaluate to a concrete integer value.
`DestroyTag(t)`	Destroys tag t.
`Constrain(v,cond)`	Ensures that variable v always satisfies expression cond. The execution path fails if it doesn't.
`Fail(msg)`	Stops the current path and prints message msg to the console.
`If (cond,instr1,instr2)`	Two execution paths are created; the first one executes instr1 as long as cond holds. the second path executes instr2 as long as the negation of cond holds.
`For (v in regex,instr)`	Iterates through all variables that match the name given by regex, executing instruction instr.
`Forward(i)`	Forwards this packet on exit port i. Used in the context of Click elements.
`InstructionBlock(i1,...)`	Groups a number of instructions that are executed in order.
`NoOp`	Does nothing.

Figure 5.6: SEFL instruction set.

Ideally, the number of paths explored by symbolic execution should be comparable to the number of paths in the network; in other words, the model of any network box should not produce more paths than the number of outgoing links from that box; for instance, in the middlebox code parsing TCP options in Figure 5.5, we should have one or two paths at most. This property would make symbolic execution tractable even on very large networks.

To achieve such scalability, a path in network symbolic execution must be tied to an active packet passing through the network: if a codepath does not result in packets, it should not be symbolically executed. Also, it follows that models of network boxes should only focus on the paths that decide the fate of packets, leaving out any logging, reporting, system checks, and so forth. Note that the C language does not have this property: a packet is just one of many variables handled by the program, and dropping a packet does not stop the execution of the box. Another fundamental problem is the poor handling of data structures, as shown in our TCP options example.

5.2.2 Design Overview

We take a radically different approach to enable large scale symbolic execution. We need a new verification-driven modeling language that is imperative —thus easier to program with, especially by network admins— and that allows us to harness networking domain knowledge to simplify, as much as possible, the task of the symbolic execution engine. To this end, we have designed a novel language called SEFL that makes it possible to consider symbolic execution as part of modeling rather than seeing it as a retrospective verification and validation activity.

A major design question regards the way packets are modeled. With Header Space Analysis, headers have a fixed size, and all possible layers have to be present at all time. This is fine for L2 boxes (such as Openflow switches), but won't work in large, heterogeneous networks. Network Optimized Datalog models packets as a collection of independent "variables"; it can capture tunnels to some extent, but it does not capture the physical layout of packets, and the problems that encapsulation may bring (e.g. interpreting the wrong part of the header, not knowing the higher level protocol, etc.).

SEFL models packet headers as variables too; however, each packet header has an absolute offset at which it is allocated. All SEFL allocation and deallocation commands include the explicit size of the header field; no symbolic sizes are allowed to ensure tractability. As header fields have predictable sizes, this is not a limitation in practice.

Packet encapsulation and decapsulation is performed by allocating more headers in front of the existing ones, or by deallocating existing headers. Each packet has a number of *tags* defined, pointing to the *start* and *end* of the packet. Other tags may be defined dynamically, pointing to addresses where layer two, three and four headers start. To get or set a header field, the programmer will use indexed addressing based on the existing

tags and a fixed offset. Tags are also variables, but they always have concrete values. Allowing symbolic values for the tags would create huge branching for no real benefit.
For instance, the IP source address field can be accessed by adding 12 bytes to the layer three tag. In Symnet, tunneling is very natural to implement: one simply needs to change the current layer three tag, making room for the new headers, allocate and assign the new headers. Decapsulation is the inverse process.
All header field accesses are only allowed if they refer to the beginning of a header field. In this way, Symnet offers a cheap and effective form of memory safety for packets, preventing creation of headers over existing ones, deallocation of inexistent headers, and unaligned access to allocated fields. This allows Symnet to catch buggy implementations of tunneling (e.g. wrong decapsulation applied) as we show in §5.2.7.
One last major decision is to offer native support for a map datastructure in the symbolic execution engine: SEFL programs can create or retrieve values based on string keys. This avoids the need to implement complex data structures in the models (such as those shown in Fig. 5.5) and their associated branching factor. Accesses to the map are only allowed with concrete keys (no symbolic accesses). This allows SEFL code to access and iterate over fields in the map without any branching, as shown by our model of the TCP options parser in Figure 5.9.

5.2.3 SEFL Language

In table 5.6 we list all the instructions provided by SEFL, together with their list of parameters and description. Every instruction implicitly takes as parameter the current execution state and outputs a new execution state. The state includes header variables and metadata together with their values and constraints.
The `Allocate` and `Deallocate` instructions create both header fields and metadata, depending on the parameter provided. If `v` is a string, the variable is metadata and is not aligned in any way, and memory safety checks do not apply. `v` acts as a key in a Map managed by the symbolic execution engine. If `v` is an integer (or an expression that evaluates to an integer), it is treated like a header field, and the associated memory checks are performed. SEFL also offers instructions to create and destroy tags. New tags can be created at absolute values (used when the packet is created), or relative to other tags (used for encapsulation of an existing packet).
Below is a code snippet that adds an IP header to an existing L4 header.

```
CreateTag("L3",Tag("L4")-160)
Allocate(Tag("L3")+96,32)    //IP src
Assign(Tag("L3")+96,ipToNumber("192.168.1.1"))
Allocate(IpDst,32)           //IP dst
Assign(IpDst,ipToNumber("8.8.8.8"))
```

The notation to access the IP source address field is rather wordy. To make programming easier, we have defined shorthands for all header fields that we work with: `Tag(``L3'')+96` becomes `IpSrc`. The code to initialize the destination address field uses this shorthand, and is also easier to read. The decapsulation code does the opposite: first, header fields are deallocated then the L3 tag is destroyed.
SEFL includes two instructions that constrain the execution of the current path, that have no direct correspondent in C. `Fail` stops the current execution path and prints an error message. `Constrain` applies a constraint to a variable, stopping the current path if the constraint does not hold. `Constrain` allows programmers to model filtering behaviour without branching. Below we show the SEFL and C code to drop non-HTTP packets.

```
Constrain(TCPDstPort,==80)     if (p->dst_port==80)
                               free(p);
```

The C code results in two execution paths if the dst_port field is symbolic, while SEFL only adds the dst_port==80 constraint to the current path.
`If` forks the current execution state. On one path, it applies the constraint and executes instr1. On the "else" branch is applies the negated constraint and executes instr2. If more than one instruction must be executed on any branch, an `InstructionBlock` should be used that groups more instructions into a single compound instruction. If any branch is empty, `NoOp` can be used instead.
`For` iterates over all variables that match the given regular expression. The match is computed when the for starts, and the loop in unfolded before it is executed, and creates no branches by itself.

SEFL properties. SEFL models (programs) are inherently bounded in space - there is no recursion or heap allocation - and are also bounded in time. All SEFL programs have bounded execution by construction. Loops can appear as a result of the network topology, but these are captured by Symnet.

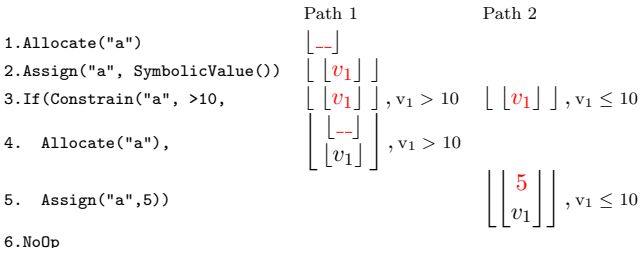


Figure 5.7: Symbolic execution with Symnet. The tool keeps a per-path value stack and assignment history for each variable.

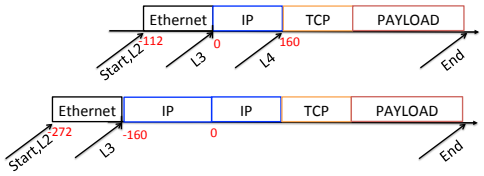


Figure 5.8: Symnet packet modeling uses the same physical layout as real packets.

5.2.4 Symbolic execution with Symnet

Our symbolic execution tool is called Symnet and takes as inputs programs written in SEFL. We have implemented a Symnet prototype in 13KLOC of Scala. The prototype also includes a frontend that parses Click modular router configurations and outputs the associated SEFL code.

Symnet execution starts by creating an initial execution path that contains an empty state (no variables are defined). Symbolic execution begins with this unique path and proceeds sequentially through the SEFL program. The execution path includes the state associated to all the variables.

As in other symbolic execution tools, Symnet instructions take as input an execution path and can modify its associated state, spawn new execution paths or both. However, Symnet is considerably more lightweight than existing symbolic execution tools such as Klee [9]:

- Symnet does not use heuristics to prioritize paths to explore because our target is finding all the possible execution paths through the network, not just covering all instructions of the network model with at least one execution path. Each path is explored until either an unsatisfiable condition was reached or there are no more instructions to execute.

- Symnet (via SEFL) only supports simple expressions (referencing, subtraction, addition, negation), and this greatly reduces state representation complexity.

Symnet differs from existing general purpose tools in two major ways. First, the state contains a map that stores variable names (or memory addresses) as keys and their associated *value stacks*, instead of simple values. Allocation and deallocation instructions push and pop a whole value stack. This allows programmers to quickly "mask" the current value of a variable and restore it later with ease. Secondly, Symnet keeps a complete trace of the values associated with a symbol which allows to check for field invariance across network hops.

Values in Symnet can be concrete or symbolic; each value has a unique identifier. For each value, on each path, Symnet holds a list of constraints that apply to that value. When these constraints cannot be satisfied, the path becomes invalid and is no longer explored. Assignment operations modify the top of the current value stack.

To better Symnet execution, let's look at the toy example in Figure 5.7. Execution starts with Path 1, and variable `a` first gets allocated a value stack and then assigned a symbolic value. At this point a new unconstrained symbolic value is created (v_1) and is pushed on the empty value stack. The `If` instruction then creates another execution path:Path 2. All the state of Path 1 is replicated to Path 2 (in fact, it is shared with a copy-on-write mechanism). Next, Path 1 gets a constraint added ($v_1 > 10$) which is checked by a constraint solver (Z3 [13], in our case). The constraint is satisfiable, so Path 1 is propagated further to the allocation instruction which creates a new empty stack for `a`. Path 1 is now done and Symnet will return to Path 2, first adding the negated constraint $v \leq 10$ and then asking the solver if it is satisfiable. Next, it assigns a constant value to `a`, finishing exploration; a complete history of variable `a` was kept on both paths.

5.2.5 Network Verification with Symnet

Reachability. It is straightforward to check reachability in a network modeled with SEFL. A symbolic packet is injected at the desired source node, and this packet is then propagated through the network by Symnet. At each node reached by the symbolic packet, we can inspect the values of and constraints on the header variables

to discover which packets are allowed, what input packets can reach the output, and how the packets look like at the output, on *all the execution paths that reach that node*.
In the example in Figure 5.7, two paths reach line 6, and variable `a` can be unassigned (on path 1) or have value 5 (path 2). If we take the union of the constraints of Path 1 and Path 2, it follows that line 6 is always reachable, regardless of the initial value of `a`.

Loop detection. The loop detection algorithm relies on the reachability algorithm with a twist: when a new node is visited, we save the current execution state. When the same node is revisited, the current state is compared to all previous states. A loop is detected when the current state is included in a previous state. State A is included in state B if every symbol in state A either a) has the same concrete value as the same symbol in B or b) the range of possible values in state A is a subset of those in state B.
The algorithm is generic and can capture different kinds of loops. If we apply it to the entire state (including header fields and metadata), the algorithm will not capture traditional forwarding loops because the TTL field will always decrease and thus the state will be different. To capture such loops, we apply the same algorithm but only consider destination and source IP addresses in the header.

Invariants. By checking the value stack of the destination address field, we find that it is bound to the same symbolic value that was set by the client. If the destination address were a constant value, invariance holds only if the variable was bound to the same constant at the origin.

Header visibility. By analyzing the value stack of a header field at an intermediate point, we can understand whether the value read is the same as that set by the source or seen by the destination. Such visibility tests allow us to check whether firewalls work on the correct fields.

Header memory safety. When creating or destroying header fields, accesses are indexed through tags. If the tags are set incorrectly, of if the program wrongly assumes the location of headers, the execution path will automatically fail. This allows us to catch various tunnel configuration problems or buggy network models.

5.2.6 Modeling networks with SEFL

We have modeled in SEFL a large subset of the elements of the Click modular router. This exercise has served two main purposes: first, it allowed us to understand whether SEFL's limited instruction set is sufficient to model a wide range of functionality. Second, we use the Click elements to implement more complex network boxes, ranging from routers, switches, firewalls, NATs etc. We use the models of individual Click elements to automatically derive SEFL models for entire Click configurations which are then used by Symnet.
Before we describe actual models, we present in more detail the way we model packets, shown in Figure 5.8. At the top of the Figure a TCP packet is encapsulated with IP and Ethernet headers. Packets always have the Start and End tags set; all the other tags are set as packets move through the (modeled) stack. Layer tags are always allocated relative to other tags; the start and end tags start at 0 by convention when a symbolic packet is first created.
The code below models a packet received from an Ethernet network interface; it first sets the L2 tag, and only IP packets are allowed through. Once the L2 tag is set, the L3 tag can be set by adding 112 bits to the L2 tag:

```
CreateTag("L2",Tag("Start"))
Constrain(EtherProto,==0x0800)
CreateTag("L3",Tag("L2")+112)
```

The bottom packet in Figure 5.8 is an IP-in-IP encapsulated packet. Note how the L4 tag is not set in this case; this will be set only in the IP decapsulation code. Any accesses to L4 fields before the L4 tag will fail, stopping the associated execution path.
SEFL network models support protocol layering natively, shielding lower layer models from the need to incorporate semantics from higher layers. This is in contrast to NOD, where lower layer models depend on higher layers, and even on the position of the box in the network.

Modeling switch behaviour. To model hardware switches, we have written a parser that takes a snapshot of CISCO switch forwarding tables and creates a SEFL model. The forwarding tables contain an output port and a destination MAC address. To reduce branching, our model simply groups all MAC addresses that should be forwarded on the same output port as follows:

```
If (Constrain(EtherDst==MAC11 |
   EtherDst==MAC12 | ... ),
```



```
Forward(port1),
If (Constrain(EtherDst==MAC21 |
   EtherDst==MAC22 | ... ),
     Forward(port2),
     Fail("Mac unknown"))
```

When run with a symbolic EtherDst, the model will result in as many execution paths as the number of output ports of the switch, which is optimal.

Modeling an IP Router. At first sight, it seems we should be able to use the same approach to model an IP router, but this is not true in the following forwarding table:

Prefix		Output Interface
192.168.0.1/32	→	If0
10.0.0.0/8	→	If0
192.168.0.0/24	→	If1
10.10.0.1/32	→	If1

If we simply group the rules per output interface and apply them using `If` instructions in the order above, the resulting forwarding will not use longest prefix match for destination address 10.10.0.1, which will be forwarded wrongly on `If0`. The most obvious solution is to have one `If` instruction for each prefix and ensure that for all overlapping prefixes, more specific matches are checked first. However, this creates as many branches as the number of prefixes in the routing table. In our example we would have four branches, but for core routers this means hundreds of thousands of branches.

A better algorithm is the following. If prefix *a* is more specific than prefix *b*, create the following constraint for `b`: `!a & b`. This ensures that the more specific prefix `a` does not match. We can now group all rules that have the same output interface as in the switch case; the number of resulting paths drops from the number of prefixes to the number of links of the router, which is again optimal.

Modeling TCP options parsing. Our models for routers and switches are exact—there is no simplification compared to the real code. To make the options parsing code symbolic-execution friendly though, we need to simplify it. The main problem is the for loop with an unknown number of iterations, and the code has branches in the loop body.

We first observe that the order in which options are placed in the options field does not matter for this box—all it does is to allow some known options and kill everything else. This suggests that we can modify the list data structure with Symnet's in-built map.

More concretely, each possible TCP option *x* (where x is in between 2 and 255) will have a corresponding metadata variable called "OPT-x" that can take values 1 or 0, modeling whether that TCP option is enabled or not.The option length and body will be held in metadata variables "OPTL-x" and "OPTB-x" respectively. In a sense, our model pre-parses the byte representation of the options and stores it in the packet metadata, allowing middleboxes to quickly access the options.

The options parsing code is given in Figure 5.9. First, the code allocates the options that it wants to allow through; if these options exist and are set, the allocation will "save" the existing values by pushing a new value on top of the stack associated to those symbols. Next, the middlebox can clear *all* options present in this packet by setting their "OPT-x" variables to 0. The last step is restoring the original values for the allowed TCP options by "popping" the top of the value stack with the `Deallocate` instruction. Note that this code does not branch at all, regardless of the values of the options, and is thus optimal from a symbolic execution viewpoint.

Modeling a Network Address Translator. NATs are ubiquiously deployed as operators come to grips with the IPv4 address space shortage. NATs modify the source IP address and source port for outgoing packets and apply the reverse mapping for incoming packets.

NATs are harder to model: they keep per flow state to ensure incoming traffic is only allowed if it is related to outgoing traffic the NAT has seen. In addition, the list of available ports at a NAT is a global variable, and the port assigned to a new connection will depend on many external factors, such as the number of active connections, the random number generator, and so forth.

To model the NAT we first observe that the exact port number assigned by the NAT is quasi random, and network operators treat it as such. Therefore it makes no sense to model the algorithm used to choose a port

```
//push new stacks for options we like
Allocate("OPT-2")
Allocate("OPT-3")
Allocate("OPT-4")
Allocate("OPT-8")
//disable all options
for (x in "OPT-*", Assign(x, 0))
//revert old values for allowed options
Deallocate("OPT-2")
Deallocate("OPT-3")
Deallocate("OPT-4")
Deallocate("OPT-8")
```

Figure 5.9: TCP options parsing code in SEFL; no new execution paths are created.

for a new connection; this would simply not scale. Instead, the newly mapped port will be a symbolic variable with allowed values in the NAT's port range. In the code below we assume for simplicity that the NAT always has available ports in the 0-10000 range; this assumption can be removed easily.

```
//only do TCP
Constrain(IPProto,==6)
Allocate("orig-ip",32,local)
Allocate("orig-port",16,local)
Allocate("new-ip",32,local)
Allocate("new-port",16,local)

Assign("orig-ip",IpSrc)
Assign("orig-port",TCPSrc)

//perform mapping
Assign(IpSrc,ipToNumber("..."))
Assign(TcpSrc, SymbolicValue())

Assign("new-ip",IpSrc)
Assign("new-port",TcpSrc)
Constrain(TcpSrc,<10000)
```

On the return path, the code restores the original mappings only if the metadata is present and matches the mapping the NAT has assigned to this flow:

```
Constrain(IPProto,==6)
Constrain(IpDst,=="new-ip")
Constrain(TcpDst,=="new-port")
Assign(IpDst,"orig-ip")
Assign(TcpDst, "orig-port")
```

The NAT uses local metadata to ensure that multiple instances of the code can be run cascaded. Local metadata will ensure each NAT instance stores and retrieves its own values. Our NAT does not create any branches - the return packet is allowed if it contains the mapping, or dropped otherwise. The NAT code is a faithful model of the real thing.

The technique we used to model the NAT—storing per flow state inside the packet—we also used to model other similar boxes including stateful firewalls and firewalls that randomize the initial sequence number of TCP connections. The same technique can be applied wherever the per-flow state is independent across flows. Under this (admittedly strong) assumption, symbolic execution can verify large networks with stateful middleboxes without state explosion.

Modeling Encryption. Encrypted tunnels are being deployed more and more. We need to capture two properties:

- Once encrypted, no network box can read the original contents of the payload.
- If we decrypt using the same key that was used to encrypt, we will retrieve the original payload

As in the NAT case, predicting the way the ciphertext will look is not important for our model. All that matters is that the original content is not available after encryption. We could use the following code snippet to encrypt with key K, where K is a parameter.


```
Allocate("Key")
Assign("Key",K)
Allocate(TcpPayload)
Assign(TcpPayload,SymbolicVariable)
```

The decryption only proceeds if the key matches:

```
Constrain("Key",==K)
Deallocate(TcpPayload)
```

Despite its simplicity, the code above has the two properties we seek. Any box reading the TCP payload after encryption will only see a novel unbounded symbolic variable, not the original contents. Only using the proper decryption will retrieve the original contents.

Are SEFL models valid?

SEFL models can be checked very quickly, but they are only useful as long as they accurately reflect the processing performed by the baseline code they mimic. As modeling is manual, inadvertent errors may be introduced.

To catch such bugs, we have developed a simple automated testing framework that relies on Symnet and compares the model to the actual implementation (be it a Click configuration or a hardware appliance). Our automated tool is similar in principle to ATPG [56] and proceeds in the following steps:

(i) We run a reachability test over the SEFL model, with a TCP/IP packet with symbolic fields. The output is a series of paths, where each path places a number of constraints on the header fields of the injected packet.

(ii) Pick a random execution path and use Z3 and the path constraints to generate concrete values for all the header fields, resulting in a concrete packet p; goto step 4.

(iii) Pick random values for the header fields to generate concrete packet p.

(iv) Packet p is used as input packet to the SEFL model and output packet(s) are generated via Symnet execution.

(v) Packet p is also injected into the real implementation and the outputs are saved.

(vi) If the two outputs of Symnet and the implementation match, repeat from steps 2 or 3, chosen randomly. Otherwise, report the error.

Our testing procedure aims to explore all paths, and it will be more accurate if it is left to run for longer. By using the symbolic execution paths, we aim to quickly cover most cases that might result in errors, but this is a heuristic. We have applied this testing tool to our models of Click elements and found it very useful. For instance, it helped us uncover the problem with grouping prefixes per output port in our IP router.

5.2.7 Evaluation

We ran experiments using Symnet on a quad-core Intel i5 machine with 8GB of RAM. Our evaluation seeks to understand whether Symnet scales symbolic execution to realistic networks, and compare its runtime to that of HSA. Next, we apply Symnet to real world scenarios to understand its usefulness in practice.

Performance micro-benchmarks. We first examine the number of execution paths resulting after injecting a purely symbolic packet into the previous examples (tunnel, switch, router and TCP options parser). We found that in all cases, the number of paths was equal to the number of outgoing links of the box, and thus optimal. The execution time of the tunnel and the TCP options parser was around 100ms. Compared to Klee, the TCP options parser runs 5 orders of magnitude faster. The execution time of Symnet grows linearly with the number of execution paths, the number of instructions per path and the complexity of the path constraint. In the case of a switch, the path constraint is simple; to explore 1000 paths only takes Symnet around 0.7s. Symbolic execution of a backbone router from the Stanford network that has 190.000 prefixes in its routing table takes around 15s. The number of execution paths here is very small (around 40), but the bulk to the execution time is used by Z3 to check the path constraints. If we disable per-port batching of prefixes, the number of paths jumps to 190.000, Symnet uses more than the available memory of our machine and starts swapping; the execution was still running 20mins later, at which point we stopped it.

	HSA [27]	HSA local	Symnet
Generation Time	2.5min	3.2min	8.1min
Runtime	18.6s	24s	37s

Table 5.2: Symnet vs HSA runtime comparison

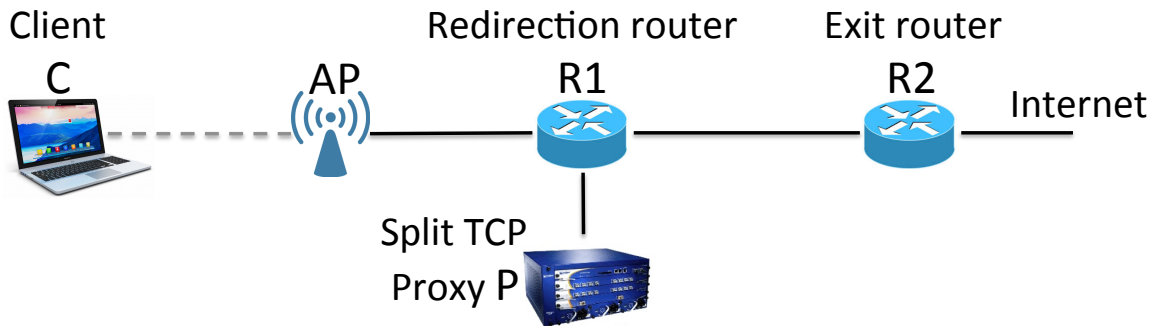


Figure 5.10: Split TCP Deployment, sideband mode [29].

Next, we seek to understand how Symnet compares to HSA, the most efficient static analysis tool today. We use the Stanford backbone network data [27] run reachability from an access router to all core routers with both Symnet and HSA. The results are given in table 5.2 and show that Symnet is within 50% of the execution time of HSA, despite its power. In comparison, NOD is reported to be 20 times slower than HSA [31] on the same benchmark.

Functional Evaluation

In this section we seek to understand what type of properties Symnet can verify that are useful in practice. To this end, we have selected two papers that report on various issues that are difficult to debug in live networks. The first paper describes operation experiences learned while deploying a Split TCP middlebox in ten enterprise networks serving thousands of users [29]. We have modeled the network topology that Split TCP uses and we show it in Figure 5.10. The Split TCP Proxy is deployed adjacent to router R3 which is configured to redirect traffic coming from both directions to P by rewriting the destination MAC address of the packet.
We model each box as a separate Click configuration composed of standard Click elements, and a separate file describes how the boxes are interconnected. Our model faithfully mimicks packet processing along the whole path, including Ethernet header encapsulation and decapsulation at each hop, routing and filtering. We now discuss how Symnet can be used to discover each of the issues observed in the paper.

Asymmetric routing. We run a reachability check from C to R2, and at R2 we use IPMirror to send the traffic back to C. Symnet shows that all execution paths from C to R2 and reverse cross via P, thus the setup is correct.

MTU issues. Router R1 is configured to drop all packets with size large than 1536B. We inject a symbolic packet at C with a symbolic IP length field. At R2, the IP length field has a constraint attached: $length < 1536$.
Next, we use IP-in-IP tunneling for traffic between R1 and P. This further reduces the available MTU, and was creating difficult to debug performance problems in the actual deployment: ping and TCP connection setup worked fine, but subsequent full MTU traffic from the client was blackholed because they exceed the MTU after encapsulation [29]. Running reachability on this new setup, the new constraint applied to length becomes: $length + 20 < 1536$, thus the client's maximum length must be smaller than 1516 for packets to go through.

Missing VLAN tagging. In one setup, P was removing VLAN tags before processing packets, and was not adding them back before pushing packets back ro R1. This caused R1 to drop those packets because it was expecting VLAN tagging [29]. A simple reachability check quickly highlights this problem: when R1 attempts to remove the VLAN tagging it finds the wrong EtherType and drops the packet.

Security Appliance. In one deployment, R2 acted as a DHCP server too, and it filtered packets where the

Ethernet source address, IP source address tuple was not in its assigned leases. We modeled the DHCP assignment by using two metadata variables set by C: "origIP" and "origEther". Both were set by the source to have the same symbolic value as the Ethernet and IP source address fields in the symbolic packet. R2 filters all packets where origIP≠pSrc or origEther≠EtherSrc. We then ran reachability again, and found that all packets were dropped by R2 because the source MAC was being modified by P and the second constraint didn't hold.

Extending TCP. Next, we wish to replicate the active testing performed by Honda et al. [25] to measure whether TCP extensions are possible. We have implemented a simple model of two TCP endpoints. We model the three way handshake, then the client sends a single data segment of unspecified size which is acked by the server and then a FIN exchange follows.

We first use this model to check whether the option negotiation mechanism is sound when middleboxes exist on path that strip TCP options: after the three-way handshake both endpoints must agree on whether the new option is enabled or not. We tested this setup with asymetric paths where the option stripping middleboxes are on one path, both or none. The results confirm the findings in [25]: the standard TCP options negotiation mechanism when paths are asymmetric, and the option must also be echoed on the third ACK for correct negotiation.

Next, we want to understand whether TCP works through a box that randomizes initial sequence numbers by adding a arbitrary offset to sequence numbers of outgoing packets and subtracting it from the ACK values. We model this by setting the $offset$ to a symbolic value and saving it in the packet metadata. The ISN box adds $offset$ to the sequence number field of outgoing packets, and subtracts the metadata from the ACK field. The verification is simple: at the TCP active opener, $ACK = SEQ + 1$ must always hold. This model can be extended to show that Multipath TCP is resilient to sequence number rewriting, while other TCP extensions are not.

5.2.8 Conclusions

Symbolic execution is a powerful tool for network verification, but applying it to production networks is challenging.To allow scalable network symbolic execution we have proposed SEFL, a novel, minimalist, imperative language tailored by design for network symbolic execution. We have built Symnet — a fast symbolic execution tool for SEFL code.

To understand the expressiveness of SEFL, we have modeled many networking devices ranging from switches and routers to middleboxes that parse TCP options or rewrite sequence numbers. We have also modeled a large subset of the elements of the Click modular router, which allows us to verify Click configurations out-of-the box. Our evaluation shows that all our models have near-optimal branching factors per box and that Symnet seamlessly scales to large networks.

Finally, we have applied Symnet and SEFL to capture a number of middlebox behaviours described in the literature. Our experience shows that Symnet catches many interesting network properties and is very fast.

6 Conclusions

This deliverable described the tools for controlling liquidity developed by the Trilogy 2 project in its third and final year.

We started by describing the Trilogy 2 Information Model that is used to describe the different resources being pooled in the liquid network. This allows proper characterization of the resources involved. We then described the tools used for controlling liquidity from the end user side. We presented a new mechanism for encrypting web traffic, namely HTTPCrypt, motivated by a large feasibility analysis of deployment of encryption on the Internet. We also presented Kadupul, an incentive framework for users to create liquidity at the edge of the network and the MPTCP subflow manager, to assist the end user in the subflow creation strategy.

We then moved on to describe the tools for controlling liquidity from the operator's side. We described the incentive and enforcement tools we built in the Federated Market and then we described how operators can play with packet drops to affect the liquidity created by users using MPTCP.

Finally we presented two tools for understanding liquidity, a web dependency graph analyzer, to understand how modern web pages can benefit from liquidity tools, such as MPTCP and Symnet, a symbolic execution framework for network functions, to help reasoning about deployed network function and predict network behavior when network functions are dynamically deployed.

7 Appendix

7.1 DRaaS use-case listing

List of DRaaS entities (systems and users)

(i) **Source Cloud (or Client Cloud)** - the cloud that is running applications and virtual machines (Source VMs) of End Users. Once a replication is enabled for any of Source VMs the Source Cloud starts to stream any data changes happening on Source VMs to Provider Cloud. Source Clouds has to be configured before it could be used for DRaaS

(ii) **Provider Cloud** - the cloud that is running Shadow VMs - tiny helper VMs the single purpose of which is to receive data streamed from Source Cloud and save it locally. Each Source VM enabled for replication has corresponding Shadow VM on the provider side. Provider Cloud has to be configured before it could be used for DRaaS

(iii) **Dashboard** - a server that is controls and monitors replication between Source and Provider Clouds. Dashboard stores API credentials for all clouds it has access to and uses them to setup/teardown replications, provision VMs, create/remove disks and other VM resources via OnApp CP API.

(iv) **Cloud Owner** - the person that manages one or more Clouds. Cloud Owner can have an account in the Dashboard, and also user account on the Clouds that he manages

(v) **End User** - a user with an account on the Source Cloud, who is using the Source VMs running on the same Cloud.

(vi) **Cloud Owner Dashboard access role** - users with this role can manage the clouds that are owned by them in the dashboard, all vms that belong to those clouds and any End Users owning those VMs

(vii) **End User Dashboard access role** - user with this role can only manage VMs that are owned by them.

Workflow 1: Initial configuration When a Cloud Owner decides that he wants to provide DRaaS services for his End Users, he needs to go through process of configuration of his Cloud.

(i) A new Dashboard account with access rights of Cloud Owner is created for him by a member of Dashboard staff. Once a Dashboard account is ready, Cloud Owner recieves an email with a prompt to set up a password for his Dashboard account and start cloud configuration

(ii) Cloud Owner logs in to the Dashboard, and launches a Cloud Registration Wizard to register his Cloud within the Dashboard. Wizard prompts Cloud Owner to fill some description fields for the cloud (like name and description and subdomain - see more about subdomains below) and API access details (which can be obtained from the CP). Once those are entered (and verified to work), Dashboard generates a Dashboard API key/token pair and inserts those into the Cloud, so that the Cloud will be able to make authenticated API requests to the Dashboard

(iii) Next step for Cloud Owner is to specify which exact Zones on his Cloud will be available for DRaaS. This is done via Hypervisor Zone Registrations Wizard, which fetches list of all Hypervisor Zones from the Cloud. Once User selects a Zone, he needs to choose one of Provider Hypervisor Zones (which are previously registered by Dashboard staff), so that he has some control on where does he want the data of his End Users to flow. Once this Wizard is complete, Dashboard inserts an identifier token to the Hypervisor Zone on CP via CP API.

(iv) All VMs on the Clouds that have the identifier set for their Hypervisor Zones now have the ability to enable DRaaS replication.

Workflow 2: End user initiates replication of his VM

(i) Once a Source Cloud has the DRaaS enabled for it (as per Workflow 1), End Users of the same Cloud are now able to initiate the replication for any VMs that belong to them and run in the Hypervisor Zones that were configured for DRaaS.

(ii) Once End User presses “Enable DR” for a specific VM, several API queries are made by the Source Cloud to the Dashboard.

(iii) First API query creates a new Dashboard user account with End User access rights role. Once the account is ready the user will recieve an email with invitation to log in to the Dashboard and check the progress of DR of his VMs.

(iv) Second API query registers the VM within the Dashboard.

(v) Once the Dashboard successfully registerd the VM, it performs a set of CP API queries to the Source Cloud to get VM metadata (like ram, os type, disk size, networks, ips etc).

(vi) Once VM metadata is collected, the Dashboard builds a VM with similar metadata on the Provider Cloud side (it uses the Provider Hypervisor Zone that was associated with Source Hypervisor Zone that runs the VM on registration). This VM becomes a Shadow VM, e.g. even it has all the Source VM metadata, like 16GB of RAM, 5 networks etc, it is running currently in “shadow” mode, with only a little ram actually allocated, and a single public network.

(vii) Once the Shadow VM is ready on Provider Cloud side, Dashboard makes an API call to Source Cloud to prepare keys that are going to be used to initiate data streaming tunnel between the clouds

(viii) Once Dashboard has the tunnel keys, it make an API call to Provider Cloud to insert those keys into the Shadow VM, so that the VM will accept the tunnel. The Shadow VM is launched afterwards

(ix) Once the keys are prepared and the Shadow VM is running, Dashboard makes CP API calls to the Source Cloud to initiate the replication for each of the disks of the Source VM.

(x) Once previous step is complete the replication is considered to be established, and the Source CP periodically sends replication progress data to the Dashboard.

(xi) Once replication progress reaches 100% Dashboard enables the VM to be failed over to Provider side.

Workflow 3: VM failover

(i) When a Source VM is being replicated at 100%, Dashboard allows Failover, i.e. any user that can access the VM on the Dashboard (like Cloud Owner of the Source Cloud, or End User that owns the VM) can login to Dashboard and initiate VM failover.

(ii) Once Failover is initiated, Dashboard makes a series of API calls that perform the failover:

API call to Source Cloud to teardown the replication

API call to Provider Cloud to stop the Shadow VM

API call to Source Cloud to stop the Source VM

API call to Provider Cloud to remove the “Shadow” status fo the VM. Shadow VM now becomes Failover VM

API call to Provider Cloud rebuild network on the Failover VM

API call to Provider Cloud to start the Failover VM

(iii) After those operations the VM is considered to be failed over and running on the Provider Cloud

Workflow 4: VM failback

(i) When a VM is running in Failover mode, any user that can access the VM on the Dashboard can decide its time to Failback the VM.

(ii) Once Failback is initiated, Dashboard makes a series of API calls that move the VM back to Source Cloud

API call to Source Cloud to add the Shadow status to Original VM

Series of API call that copy any data collected in Failover VM back to original VM - this is the same as decribed in Workflow 2, only the direction of replication is now from Failover VM to Original VM, which is now running in Shadow mode

Once replication progress reaches 100

API call to Source Cloud to power off the Original VM (in Shadow mode)

API call to Source Cloud to remove the Shadow status of the VM

API call to Source Cloud to rebuild the network of the VM

API call to Source Cloud to launch the Original VM

(iii) After those operations the VM is now again running on the Source Cloud

(iv) Replication to the Provider Cloud is now re-established as per Workflow 2

Bibliography

[1] Uci cloud owl ontology, February 2009.

[2] Scalable Testing of Context-Dependent Policies over Stateful Data Planes with Armstrong. http://arxiv.org/abs/1505.03356, 2015.

[3] Alexa top 500 global sites. http://www.alexa.com/topsites.

[4] Darko Androcec, Neven Vrcek, and Jurica Seva. Cloud computing ontologies: a systematic review. In *Proceedings of the third international conference on models and ontology-based design of protocols, architectures and services*, pages 9–14, 2012.

[5] Mike Belshe and Roberto Peon. SPDY protocol: Draft 3.2. http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-2.

[6] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015.

[7] Luca Boccassi, Marwan M. Fayed, and Mahesh K. Marina. Binder: A system to aggregate multiple internet gateways in community networks. In *Proceedings of the 2013 ACM MobiCom Workshop on Lowest Cost Denominator Networking for Universal Access*, LCDNet '13, pages 3–8, New York, NY, USA, 2013. ACM.

[8] Antonio Brogi, Ahmad Ibrahim, Jacopo Soldani, José Carrasco, Javier Cubo, Ernesto Pimentel, and Francesco D'Andria. Seaclouds: a european project on seamless management of multi-cloud applications. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–4, 2014.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[10] Peter Pin-Shan Chen. The entity-relationship modeltoward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[11] M. Coudron. Mptcp netlink. https://github.com/teto/mptcpnetlink, Feb 2014.

[12] Andrei Croitoru, Dragos Niculescu, and Costin Raiciu. Towards wifi mobility without fast handover. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 219–234. USENIX Association, 2015.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] G. Detal and S. Barré. Flexible path managers for MPTCP. http://www.tessares.net/path-manager/.

[15] CIM DMTF. Schema, 2015.

[16] Anhai Doan et al. Crowdsourcing systems on the world-wide web. *ACM*, 2011.

[17] Mihai Dobrescu and Katerina Argyraki. Software dataplane verification. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 101–114, Berkeley, CA, USA, 2014. USENIX Association.

[18] E. Dumazet and Y. Cheng. TSO, fair queuing, pacing: three's a charm. Presented at IETF'88, Nov. 2013.

[19] Philip Eardley. Survey of MPTCP Implementations. Internet-Draft draft-eardley-mptcp-implementations-survey-02, IETF Secretariat, July 2013.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[21] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, January 2013.

[22] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[23] Mark Handley. Why the internet only just works. *BT Technology Journal*, 2006.

[24] Seppo Hätönen, Aki Nyrhinen, Lars Eggert, Stephen Strowes, Pasi Sarolahti, and Markku Kojo. An experimental study of home gateway characteristics. In *IMC*, pages 260–7, New York, New York, USA, 2010. ACM Press.

[25] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 181–194, New York, NY, USA, 2011. ACM.

[26] A. Huttunen, B. Swander, V. Volpe, L. DiBurro, and M. Stenberg. UDP Encapsulation of IPsec ESP Packets. RFC 3948 (Proposed Standard), January 2005.

[27] 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.

[28] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[29] Franck Le, Erich Nahum, Vasilis Pappas, Maroun Touma, and Dinesh Verma. Experiences deploying a transparent split-tcp middlebox in operational networks and the implications for nfv. HotMiddlebox'15, 2015.

[30] Yeon-sup Lim, Yung-Chih Chen, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. How green is Multipath TCP for mobile devices? In *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, & Challenges*, pages 3–8. ACM, 2014.

[31] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 499–512, Berkeley, CA, USA, 2015. USENIX Association.

[32] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Sigcomm*, 2011.

[33] Francesco Moscato, Rocco Aversa, Beniamino Di Martino, Teodor-Florin Fortis, and Victor Munteanu. An analysis of mosaic ontology for cloud resources annotation. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 973–980. IEEE, 2011.

[34] Andreas Müller, Florian Wohlfart, and Georg Carle. Analysis and topology-based traversal of cascaded large scale NATs. In *HotMiddlebox*, pages 43–48. ACM Press, 2013.

[35] Ns-3 network simulator. `https://www.nsnam.org`.

[36] M. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

[37] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. Exploring Mobile/WiFi Handover with Multipath TCP. In *ACM SIGCOMM CellNet workshop*, pages 31–36, 2012.

[38] Christoph Paasch, Sebastien Barre, et al. Multipath TCP in the Linux Kernel. available from `http://www.multipath-tcp.org`.

[39] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. Verifying Isolation Properties in the Presence of Middleboxes. Tech Report arXiv:1409.7687v1.

[40] Qiuyu Peng, Minghua Chen, Anwar Walid, and Steven Low. Energy efficient Multipath TCP for mobile devices. In *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '14, pages 257–266, New York, NY, USA, 2014. ACM.

[41] QUIC, a multiplexed stream transport over udp. `http://www.chromium.org/quic`.

[42] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *ACM SIGCOMM 2011*, 2011.

[43] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol. RFC 3549 (Informational), July 2003.

[44] M. Scharf and A. Ford. Multipath TCP (MPTCP) Application Interface Considerations. RFC 6897, March 2013.

[45] Philipp S. Schmidt, Theresa Enghardt, Ramin Khalili, and Anja Feldmann. Socket intents: Leveraging application awareness for multi-access connectivity. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 295–300, New York, NY, USA, 2013. ACM.

[46] Stelios Sotiriadis and Nik Bessis. An inter-cloud bridge system for heterogeneous cloud platforms. *Future Generation Computer Systems*, 2015.

[47] Stelios Sotiriadis, Nik Bessis, and Euripides GM Petrakis. An inter-cloud architecture for future internet infrastructures. In *Adaptive Resource Management and Scheduling for Cloud Computing*, pages 206–216. Springer, 2014.

[48] SPDY: An experimental protocol for a faster web. `http://www.chromium.org/spdy/spdy-whitepaper`.

[49] Yeon sup Lim, Yung-Chih Chen, E.M. Nahum, D. Towsley, and Kang-Won Lee. Cross-layer path management in multi-path transport protocol for mobile devices. In *INFOCOM, 2014 Proceedings IEEE*, pages 1815–1823, April 2014.

[50] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. Overify: Optimizing programs for fast verification. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 18–18, Berkeley, CA, USA, 2013. USENIX Association.

[51] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with WProf. In *Proc. of the 11th USENIX symposium on Networked Systems Design and Implementation (*NSDI '13*)*, 2013.

[52] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, 2011.

[53] 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.

[54] Sami Yangui. *Service-based applications provisioning in the cloud*. PhD thesis, Evry, Institut national des télécommunications, 2014.

[55] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. IEEE, 2008.

[56] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 241–252, New York, NY, USA, 2012. ACM.