ICT-317756

TRILOGY2

Trilogy2: Building the Liquid Net

Specific Targeted Research Project

FP7 ICT Objective 1.1 The Network of the Future

D1.2 Initial Cross-Liquidity tools

Due date of deliverable: 31 December 2013

Actual submission date: 3 January 2014

Start date of project	1 January 2013
Duration	36 months
Lead contractor for this deliverable	Nextworks (NXW)
Version	v1.0, 3 January 2014
Confidentiality status	Public

© TRILOGY2 Consortium 2014

Abstract

This document presents the results of the analysis carried out on the interactions between existing techniques for creating liquidity in the Internet. In particular the focus is on possible combination and integration of liquidity mechanisms that collect and manage individual resources as pools at heterogeneous domains. These resources include storage, processing and bandwidth. The final aim is to enable, through these interactions, a converged architecture for the orchestration of provisioning, usage and control of these heterogeneous resources in the Internet. In this context, the deliverable describes the intial set of cross-liquidity tools and mechanisms that have been designed, and in some cases implemented, during the first phase of the Trilogy 2 project.

Target Audience

The target audience for this document is the networking research and development community, particularly those with an interest in the Future Internet technologies and architectures. The material should be accessible to any reader with a background in network architectures, including mobile, wireless, service operator and data centre networks. This document will also be of interest to those concerned with the interactions among heterogeneous network architectures and resource pooling mechanisms, although specialist expertise in these areas is not a pre-requisite.

Disclaimer

This document contains material, which is the copyright of certain TRILOGY2 consortium parties, and may not be reproduced or copied without permission. All TRILOGY2 consortium parties have agreed to the full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the TRILOGY2 consortium as a whole, nor a certain party of the TRILOGY2 consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

This document does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of its content.

Impressum

Full project title	TRILOGY2: Building the Liquid Net
Title of the workpackage	D1.2 Initial Cross-Liquidity Tools
Editor	Giacomo Bernini, NXW
Project Co-ordinator	Marcelo Bagnulo Braun, UC3M
Copyright notice	© 2014 Participants in project TRILOGY2

Executive Summary

The Internet is a concatenation of highly interconnected resources, such as bandwidth, processing, storage and energy, belonging to heterogeneous domains. The rapid growth of server virtualization, cloud services, Content Delivery Networks (CDNs), distributed processing and computational frameworks, are setting the basis for a new converged fluid architecture capable of creating and controlling heterogeneous resource pools as a whole liquid system, the Liquid Network.

Today's Internet resource pools and liquidity mechanisms have a limited scope, mostly bound to a single resource domain. Processing or storage resources are commonly pooled and used by cloud and content delivery applications independently from bandwidth resources, resulting in an overall inefficient utilization of resources. On the contrary, bandwidth is a key resource that allows all other resources to be grouped together and used as an heterogeneous and unified resource pool. A converged and seamless pooling of bandwidth, processing and storage resources is therefore fundamental to achieve actual liquidity in the current Internet.

In this context, Trilogy 2 aims to provide a converged architectural framework capable of orchestrating, provisioning, and controlling the usage of heterogeneous resource pools as demanded by emerging highly distributed applications. The overall architecture is described in Deliverable D2.1. The basic components of this architecture are the mechanisms and techniques that create liquidity at the different resource domains: bandwidth, storage and processing. Some of these mechanisms exist in the Internet today and several others are proposed by the Trilogy 2 project. The new mechanisms for creating liquidity in the different domains are described in Deliverable D1.1. In addition to creating new liquidity tools, Trilogy 2 provides the integration of these newly proposed tools with the aim of improving the interactions among the heterogeneous resource pools. The interaction between these new mechanisms, along with the interaction between the liquidity tools and the other components of the Internet architecture are explored in this Deliverable. On top of these cross-liquidity tools a uniform resource information model acts as the glueing description language for a converged and seamless control of these heterogeneous resource pools across the Internet. The initial version of the information model is described in Deliverable D2.2.

The focus of this document is on the cross-liquidity mechanisms that have been designed, and in some cases implemented, during this first year of the project in the context of WP1 activities. A cross-liquidity tool has to be considered as a set of mechanisms and procedures that regulate and facilitate the interactions among existing and well-defined resource pooling techniques. In particular we focus our work in the interaction between the new tools for liquidity discussed in Deliverable D1.1 and other elements of the Internet architecture. The interactions that have been investigated in this first phase of Trilogy 2 can be categorized as follows:

- interactions between mechanisms and tools that create bandwidth liquidity at different layers in the networking stack

- interactions between processing and bandwidth liquidity tools for a converged and integrated resource

pooling

- interactions between bandwidth and storage liquidity mechanisms
- interactions between storage and processing resource pooling techniques

This deliverable describes the initial Trilogy 2 cross-liquidity tools, mainly focusing on the first two categories mentioned above. Interactions between bandwidth and storage, as well as between storage and processing liquidity mechanisms, have been the primary focus during this first year of the project: this research work is still in progress and the results (in terms of new cross-liquidity tools) will be provided in the next WP1 deliverables.

Also in this Deliverable, a set of cross-liquidity mechanisms that operate in the bandwidth domain are provided: they differentiate according to the network segment they consider. Indeed, tools for enhanced liquidity, flexibility and optimization of bandwidth utilization in data centre and wireless networks are presented. Moreover, in the context of processing and bandwidth integrated resource pooling, a cross-liquidity tool for live migration of applications among mobile devices has been designed and also prototypes have been implemented.

The aim of this deliverable is to describe the above mentioned cross-liquidity tools as stand-alone components in the Trilogy 2 reference scenario. A more comprehensive and integrated architecture description is presented in Deliverable D2.1.

 © TRILOGY2 Consortium 2014

List of Authors

Authors	Giacomo Bernini, Gino Carrozzo, Nicola Ciulli, Alexandru Agache, Andrei Croitoru, Dragos Niculescu, Catalin Nicutar, Costin Raiciu, Pedro Andrés Aranda Gutiérrez, Primitivo Matas, George Milescu, Cătălin Moraru, Valentin Ilie, Rareş Dumitrescu
Participants	NXW, UPB, TID, INTEL
Work-package	WP1 - Creating Liquidity
Security	PUBLIC (PU)
Nature	R
Version	1.0
Total number of pages	50

Contents

 © TRILOGY2 Consortium 2014

List of Figures

 © TRILOGY2 Consortium 2014

List of Tables

1 Introduction

1.1 Motivations and scope

The Internet today is composed by a heterogeneous set of resources and distributed functions mostly interconnected, which span from the traditional network services up to virtualized ones. It includes classic network and security functions (enhanced forwarding, Quality of Service (QoS), firewalls, Network Address Translators (NAT), Deep Packet Inpsectors (DPI), traffic scrubbing, load balancing, etc.), as well as the virtualized infrastructures (servers and storage technologies), platforms (OS and middleware functions/primitives), and software layers (Software as a Service - SaaS) typically deployed within data centres.

Multiple technologies and solutions exist today that allow network operators and service providers to easily launch a variety of base on-demand services for the dynamic pooling of processing, storage, or bandwidth resources. Initial investigative work is being carried out for the joint orchestration of processing and storage, mostly within a single administrative entity and generally within the single data centre. Key for the evolution of these technologies and services is the possibility to treat the different resources as belonging to the same pool and consequently orchestrate their allocation and usage to better match the complex, multi-layer service chaining. In Trilogy 2, such a resource pool is the base of the Liquid Network concept. In fact, the resource pool identifies a collection of individual resources which together act as a single more capable and more tangible 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.

Based on the concept of resource pool, different types of liquidity can be identified depending on the specific service aspects and involved technological areas. In Trilogy 2 the following liquidity scopes are identified:

- **Cross-provider liquidity**, which points to the pooling techniques for orchestrating and controlling bandwidth, processing, storage and energy resources across different providers / resource owners.
- **Cross-layer liquidity**, which refers to the possibility to access more flexibly to those network liquidity functions that are operating in different layers of the stack e.g. MPTCP creates bandwidth liquidity at the transport layer and MPLS traffic engineering creates liquidity at the sub-IP layer.
- **Cross-resource liquidity**, which aims at understanding the trading between different types of resources. For example, would it be possible to trade off some storage resources in exchange for lower latency and higher bandwidth?

This document describes the initial Cross-Liquidity tools that have been designed, and in some cases implemented, during the first phase of the Trilogy 2 project. In the Deliverable D1.1 we have described new liquidity tools proposed by the Trilogy 2 project. The key objective of this deliverable is to identify the

 © TRILOGY2 Consortium 2014

interactions between the previously described pooling mechanisms running within a single domain while assuming cooperating endpoints. In particular, the main focus is in this document is on cross-liquidity tools that operate inside the bandwidth domain: the main motivation is that bandwidth can be considered as the bridging resource that enables all the others to be pooled together and be used and controlled in a unified way. The scope of this document is limited to describing the key aspects and components of the cross-liquidity mechanisms presented here. A more detailed and comprehensive architecture description is provided in Deliverable D2.1.

1.2 Structure of the document

This document is organized as follows.

In Section 2 the positioning of the resources pooling and cross-liquidity mechanisms in the Trilogy 2 reference scenario and architecture is provided.

In Section 3 the cross-liquidity mechanisms that result from the Trilogy 2 design and specification cycle (first year of the project) are detailed. Focus is on the interaction of the main bandwidth liquidity tool i.e. Multipath TCP (MPTCP) with other elements of the Internet architecture, namely:

- Multiprotocol Label Switching (MPLS) traffic engineering tools,
- wireless channel switching technologies,
- data center technologies, and
- Virtual Machine (VM) migration.

In Section 4 the future and planned specification steps and studies are discussed to highlight the path Trilogy 2 partners agreed towards consolidation and testing of the proposed solutions for the other workpackages.

2 Positioning in the Trilogy 2 architecture

Figure 2.1 shows the reference scenario for the Trilogy 2 Liquid Network concept. The basic components are those mechanisms that create liquidity inside each resource domain, namely bandwidth, processing and storage domains. We have presented the key liquidity techniques in Deliverable 1.1.

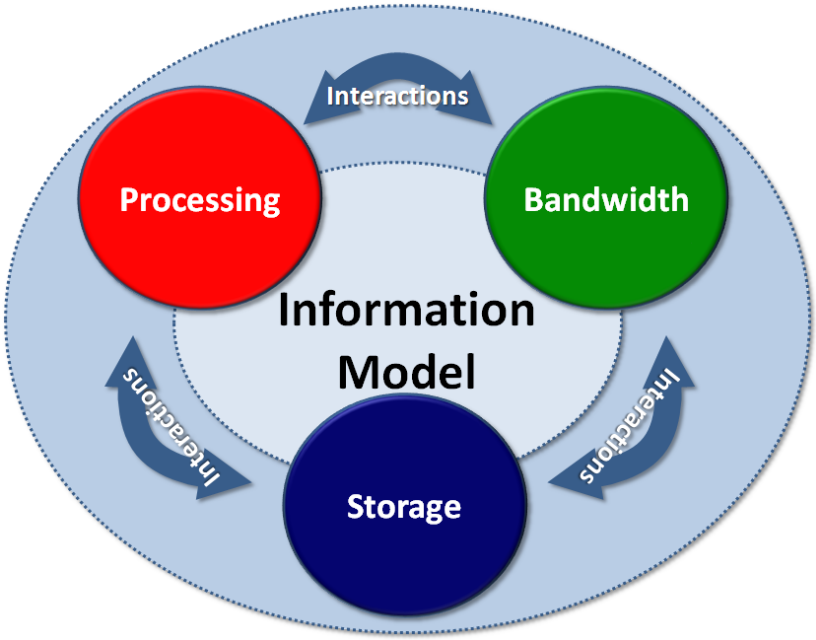

Figure 2.1: Trilogy 2 reference scenario.

One of the primary aims of Trilogy 2 is to connect these heterogeneous resource pools by developing novel liquidity mechanisms for converged interactions across resource domains. These cross-liquidity tools will enable the current disconnected and segmented Internet to evolve towards a Liquid Network where applications can have on-demand access to converged and seamless bandwidth, processing and storage resource pools. In other words, the Trilogy 2 architecture, which is described in D2.1, has been designed to provide mechanisms and tools to expose these heterogeneous resources as a unified pool, with the aim of orchestrating their provisioning and usage through seamless procedures. This is enabled by the definition of a uniform information model for the description and abstraction of heterogeneous resources belonging to different resource domains, which is described in Deliverable D2.2.

This deliverable is devoted to the definition of key aspects and components for the interactions among those existing mechanisms and tools which create liquidity at a single resource domain level. As a result, a set of cross-liquidity mechanisms have been defined in support of the interactions among heterogeneous resources depicted in Figure 2.1. In particular, the main focus here is dedicated on the one hand to the interactions between bandwidth liquidity tools (i.e. operating inside the bandwidth resource domain), and to the combination of processing and bandwidth liquidity techniques on the other. The research work for interactions between bandwidth and storage, and between storage and processing resource pooling mechanisms has been

 © TRILOGY2 Consortium 2014

also started and it is still in progress at the time of writing: the results of these investigations and related the cross-liquidity mechanisms will be presented in the next Deliverables, namely D1.3 and D1.4.

The next chapter describes the initial cross-liquidity tools that have been designed, and in some case implemented, in this first phase of the project.

3 Cross-Liquidity mechanisms

This section describes the results of the research and investigations carried out in the project regarding the interactions between the mechanisms and tools which create liquidity at different resource domains and network layers. The aim is to identify and possibly define novel combinations, interactions and integrations of such mechanisms to enhance current liquidity techniques.

The main objective here is to provide an overview of the analysis and investigations related to liquidity tools interaction carried out so far in the context of WP1 activities; experimental setups, simulations and tests are also reported where available. The focus at this stage of the project is to analyze the interaction of the main bandwidth liquidity tool presented in D1.1 namely MPTCP and other elements of the Internet architecture.

This section is structured as follows.

- Section 3.1 presents the interaction between MPTCP and MPLS traffic engineering tools
- Section 3.2 presents the interaction between MPTCP and channel switching technologies.
- Section 3.3 describes the interaction between MPTCP and the Data center technologies.
- Section 3.4 covers the interaction of MPTCP and one of the fundamental tools for CPU liquidity i.e. virtualization, in the context of mobile devices.

3.1 MPTCP and MPLS-TE Interaction

The aim of multipath transport is to pool bandwidth resources by simultaneously making use of multiple disjoint (or partially disjoint) paths across a given network [8]. There are two benefits when compared with single path transport. First, it enhances the network connectivity resilience by providing multiple paths, preserving and protecting the end-hosts (and therefore applications running on them) from failure conditions in the middle of the network. Indeed, in principle the use of multiple interchangeable paths allows data chunks to be transmitted and re-transmitted, if needed, on any available path. Moreover, multipath transport natively increases the network resource utilization efficiency by implementing a concurrent use of disjointed paths, thus increasing bandwidth capacity, throughput and goodput available at the end-hosts.

3.1.1 MPTCP-aware MPLS-TE provider

Multipath transport enables what is called bandwidth liquidity by allowing individual and independent network resources to be used as single pool of resources for the end-hosts. Multipath TCP (MPTCP) [9] aims at providing the functions described above by extending standard TCP to pool multiple paths within a transport connection in a transparent way for the application. MPTCP is designed to be used when multiple end-to-end paths exists and are available at the end-hosts, and when one or both of them are multi-homed.

However, RFC 6182 [8] does not provide a clear mechanism for MPTCP path management in terms of routing aspects and disjointness of multiple paths across the network. In this context, dedicated mechanisms

 © TRILOGY2 Consortium 2014

and tools which can allow to create and tune Label Switched Paths (LSPs) in Multiprotocol Label Switching - Traffic Engineering (MPLS-TE) networks, on demand and bound to MPTCP flows and sub-flows need to be investigated. The main concept is to setup, control and modify multiple disjoint LSPs (potentially with different degrees of disjointness, such as node based or link based, depending on the application) to carry MPTCP subflows, above all in case of single homed end-hosts with the aim of overcoming and somehow mitigating the multi-homing base assumption of MPTCP.

3.1.1.1 Problem statement

The main objective behind the interaction between MPTCP and MPLS-TE is to improve the MPTCP benefits, including flexibility and redundancy, by leveraging the native MPLS-TE pooling functions and procedures. In particular, the idea is to design a Multi-Path (MP) box able to bind and correlate MPTCP and MPLS-TE procedures by providing a set of bridging functionalities to correlate subflows with LSPs, also in case of single homed end-hosts. A first set of requirements, in terms of functionalities to be supported by this MP box, has been identified:

- MUST be able to inspect TCP and MPTCP flows
- SHOULD setup and/or tune LSPs based on TCP and MPTCP flows
- MUST implement a decision/coordination entity to correlate subflows and LSPs
- MUST redirect MPTCP subflows in already established LSPs or trigger the creation of additional LSPs
- MUST implement a dynamic association of MPTCP subflows and LSPs, enabling a Forwarding Equivalence Class to Next-Hop Label Forwarding Entry (FEC-to-NHLFE) mapping in the Label Edge Router (LER) data plane
- Optionally, MIGHT originate MPTCP subflows in case of single homed end-hosts to implement multi-path transport

In addition to the above requirements, as a general consideration, this MP box should provide a non-invasive access to the LSP tunnel APIs on the legacy MPLS routers: this means that the MP box should be an enhanced network function deployed by network operators on top of MPLS routers without impacting any of their procedures, interfaces and behaviours.

The next sub-section presents a possible deployment scenario for the MP box, and also includes its functional decomposition along with a description of main functional modules and interfaces.

3.1.1.2 Reference scenario

Network providers usually tend to offer static connectivity services in their MPLS-TE networks, mostly based on off-line route computations and resource provisioning that meet bandwidth requirements of forecasted end-to-end traffic patterns. The result is a sub-optimal and inefficient usage of bandwidth resources, especially

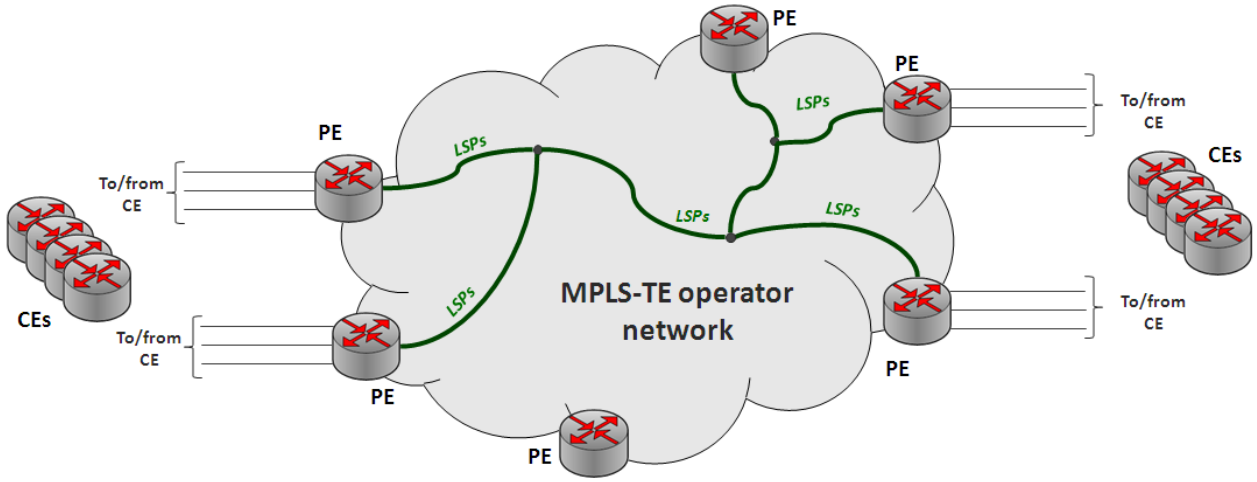

Figure 3.1: MPLS-TE operator network with pre-provisioned tunnels among PEs

when the actual traffic characteristics would encourage a more dynamic and flexible use of resources across these MPLS-TE networks. Figure 3.1 shows an example of MPLS-TE network operated by a provider, where a set of tunnels (i.e. LSPs) among the Provider Edge (PE) routers are pre-provisioned to serve the customers traffic (coming from correspondent Customer Edge (CE) routers) with general purpose connectivity.
In this context, the MP box can be deployed by network providers as a new MPTCP aware appliance to significantly enhance the flexibility and dynamicity of their MPLS-TE networks for MPTCP traffic generated by customers. The concept behind the MPTCP-aware MPLS-TE provider is twofold:

- *Data plane:* pre-pend at each PE router an OpenFlow enabled Layer2 device to tap TCP traffic
- *Control plane:* deploy on top of the PEs a Software Defined Networking (SDN)-based MP box running a set of enhanced network applications

Figure 3.2 depicts this twofold approach: for sake of simplicity, it shows the interaction of the SDN-based MP box with only two PE routers (left-side of the MPLS-TE network). In a real scenario, each PE in the MPLS-TE provider network should be pre-pended by an OpenFlow enabled device and controlled by the MP box. At the data plane, different options are available to tap the TCP traffic to be inspected by the MP box. Port mirroring on each PE router may be the first option; however, to keep the MP box functions as flexible, programmable and dynamic as possible, a preferable option is to use Open virtual Switch (OVS) or any OpenFlow Switch in combination with standard MPLS-TE PE routers. As shown in Figure 3.2 (where OVS is used), this allows the MP box to perform the inspection of TCP traffic via software through the OpenFlow protocol [16]. Indeed, the latest versions of the Openflow protocol (currently v1.3.3 is under ratification), allow special forwarding actions on reserved ports, such as the control port towards the SDN controller for tap operations. The SDN-based MP box can therefore be run as an on-line, flexible and programmable replanning engine for the MPLS-TE operator network, leveraging the benefits of an SDN architecture [23]. At each PE, TCP traffic coming from and outgoing the CE routers is tapped and inspected by the SDN-

 © TRILOGY2 Consortium 2014

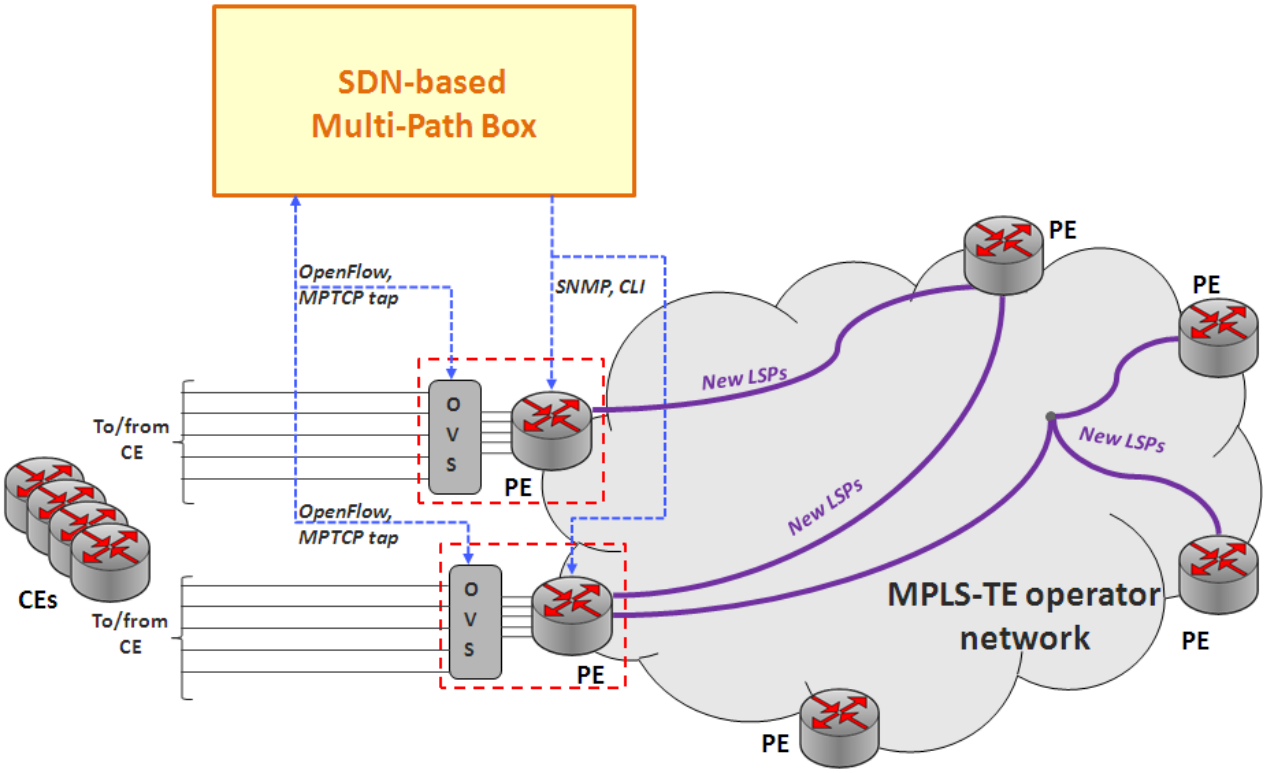

Figure 3.2: MPTCP-aware MPLS-TE provider: SDN-based MP box deployment

based MP box. Here, MPTCP flows are detected, and the information associated to the originating PEs (i.e. ingress/egress points of MPTCP flows in the MPLS-TE network) are correlated to replan MPLS-TE tunnels in the network with the aim of seeking for the maximum end-to-end MPTCP flows disjointness.

3.1.1.3 Functional architecture

The SDN-based MP box functional decomposition is depicted in Figure 3.3, that highlights its main functional entities and their interactions. It is built around an OpenFlow enabled SDN controller responsible for setting the TCP tap operations in each OVS, with a couple of network applications/functions acting as decision points for the MP box, in terms of MPTCP flows detection and MPLS-TE replanning.
The list of SDN-based MP box functional modules is provided below. For each of them, a high-level description of its main functionalities, roles and responsibilities is given:

- **SDN Controller:** It is used to implement tap operations via software, by leveraging the capabilities of the OpenFlow protocol. It controls all the OVSes at the edges of the MPLS-TE network and it configures proper rules in the OVS flow tables to retrieve TCP traffic crossing each PE router, either incoming (from CEs to PEs) or outgoing (from PEs to CEs). Its deployment allows to implement a software driven MP box, with flexible and programmable functions running on top of the SDN controller itself. In the context of the MP box the SDN controller provides basic functions, mainly focused on the configuration of few flow entries in OVSes for tap operations, without any embedded network function like Layer2 learning, routing, etc.

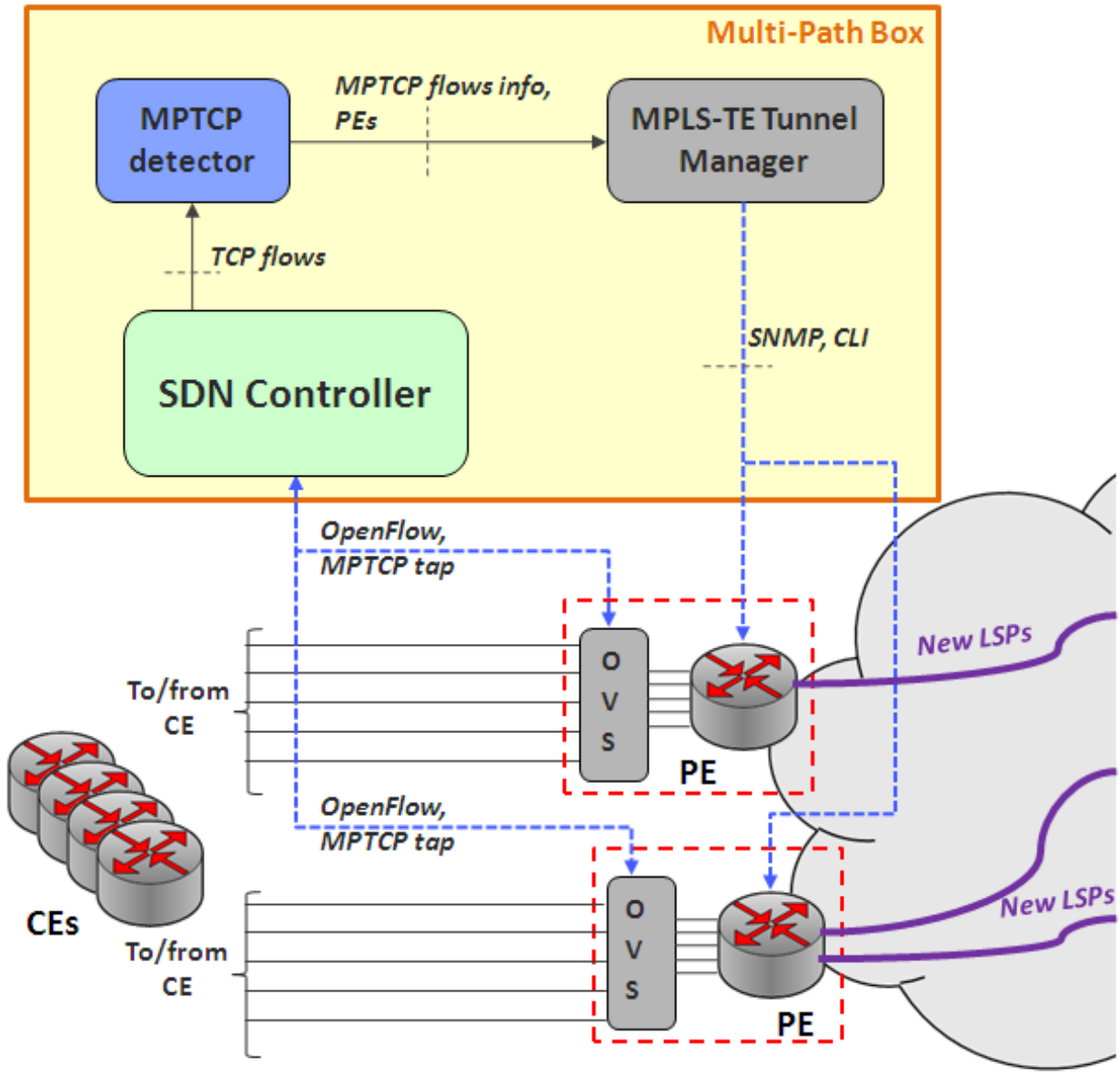

Figure 3.3: SDN-based MP box functional decomposition

© TRILOGY2 Consortium 2014

- **MPTCP Detector:** It is a network application that runs on top of the SDN controller, as an external enhanced function. It handles and inspects TCP traffic (gathered from the SDN controller northbound interface) with the aim of detecting MPTCP flows. Mainly it looks at TCP handshake to detect MPTCP subflows creation (i.e. SYN, SYN+ACK, ACK, JOIN, etc.); the flow information retrieved from multiple PEs are retrieved to correlate MPTCP flows crossing the MPLS-TE network at the same edge points (i.e. PE routers). The information regarding MPTCP flows, in terms of ingress/egress PE ports in the MPLS-TE network are then provided to the MPLS-TE Tunnel Manager to accordingly replan (if needed) the tunnels/LSPs.

- **MPLS-TE Tunnel Manager:** It is also a network application running on top of the SDN controller. It is triggered by the MPTCP detector, and it is in charge of checking tunnels/LSPs disjointness (between the given PEs) when new MPTCP flow notifications are received. This means that the MPLS-TE Tunnel Manager is aware of the tunnels/LSPs installed in the MPLS-TE network, and is able to evaluate their degree of disjointness. When needed, it triggers the re-configuration of tunnels/LSPs to seek for the maximum end-to-end disjointness; to limit the amount of replanning action specific thresholds may be configured by the network operator to let the MPLS-TE Tunnel Manager re-configure MPTCP flow aggregates (instead of single flows). Path Computation Element (PCE) functions, if available in the MPLS-TE network, may be reused for disjoint routing purposes [6]. The replanning actions can be triggered through Simple Network Management Protocol (SNMP) or Command Line Interface (CLI) interfaces exposed by the PE routers, mainly performing setup/teardown of LSPs and configuration of FEC-to-NHLFE.

3.2 MPTCP and channel switching interaction

WiFi channel switching is a technique used in order to achieve bandwidth pooling over WiFi by utilizing more than one Access Point (AP) using the same physical network interface. In order to achieve end-to-end connectivity through multiple APs, MPTCP has been used at the transport layer. Various possible interactions between channel switching policy and MPTCP have been investigated, mainly adjusting the policy and its various parameters to complement the load-balancing abilities of MPTCP's congestion control algorithms.

First, each AP is used in a round-robin fashion, spending a certain amount of time on each channel. This time interval, named T, is one of the parameters of the proposed solution. By increasing T, a higher level of delay is introduced to packets on each channel. Also, by increasing the time spent on each AP, the efficiency of the policy is improved mainly because the difference between time spent on each channel and the switching time is increased, which, with proper hardware and software modifications, is about 3ms. The value of T has been experimentally calculated at 100ms because it offers an acceptable tradeoff between RTT increase and throughput efficiency.

Second, a more complex policy, where the fixed time quota T is not the only switch parameter, has been

Wait time	MPTCP cubic	MPTCP coupled	MPTCP olia	TCP single flow	TCP no switching
1ms	9770Kbps	9280Kbps	8760 Kbps	9553 Kbps	19947 Kbps
5ms	16544Kbps	17027 Kbps	16280 Kbps	16262Kbps	21221Kbps
10ms	17057 Kbps	15644 Kbps	15797 Kbps	16172 Kbps	19875bps
20ms	15226Kbps	14242Kbps	14225Kbps	14554Kbps	19785Kbps

Table 3.1: Throughput measurements for different wait timers and MPTCP implementations.

considered. Given a situation where one of the APs gives better throughput than the other one(s), it doesn't make sense to split the time evenly between all APs. The aim is to use MPTCP to achieve good load-balancing, using its congestion control algorithms to split the traffic proportionally with the throughput on each subflow. The initial idea for achieving this is to specify a waiting time (Tw) for each channel. If, during the waiting time, no incoming or outgoing packets are detected on the sub-interface(s) on a certain channel, the switch is prematurely triggered. The Tw timer is reset on every incoming or outgoing packet.

Setting a value for Tw is a key element in this strategy. If Tw is too large, it doesn't achieve the purpose it is designed for. If Tw is too small, one or serveral APs could potentially starve by not waiting long enough for packets to arrive for that AP. Experiments have been performed by setting the Tw timer to values ranging from 1 to 20 ms. A value of 10ms seems the best for most scenarions, excluding a few edge cases. However, the objective is to create a policy which would work well for all possible cases, thus the wait time cannot be left fixed and static. It has to update itself dynamically for each channel, depending on different parameters.

3.2.1 Initial experiments

One of the cases that have been studied is a WiFi device associated to two APs on different channels, with the device near to one of them and far from the other. In this case, we would expect to receive most of the bandwidth from the closer AP, and a small part from the farther one. By modifying the Tw parameter, the results in Table 3.1 have been obtained.

Another interesting case is the one where the device is at a fairly large distance from either AP that it gets reduced throughput through either one. In this case, using channel switching and MPTCP, both APs can be outperfomed, and a higher throughput can be obtained by using both APs instead of any one of them. Further experiments also demonstrate that the congestion control used (e.g. cubic or coupled algorithms), does not affect the throughput achieved when moving from one AP to the other. This is one of the conclusions reached when testing this solution: since WiFi hides losses by retransmitting lost frames at the second layer, TCP does not register lost packets, so the congestion window does not vary according to signal strength. Only an increase in RTT is obtained, which does not alter the connection parameters by a large degree. The load-balancing is achieved at the second layer of the stack as opposed to the fourth layer, as we originally thought.

3.2.2 Dynamic timer adjustment

Another AP behaviour have been experimentally deduced: if a client sends a power-save frame to the AP while the AP has not finished sending the buffered packets, the power-save frame will be disregarded, and the AP will keep sending packets until it has finished delivering all the data buffered while the client was on

 © TRILOGY2 Consortium 2014

another channel. If the client switches channels after sending the power-save frame, the AP will keep trying to send the packets and, because the client is not replying with link-level ACKs, it will gradually increase the interval at which it sends packets, up to several milliseconds. Then, when the client comes back to the AP's channel, there will be a sizeable delay before the reception of the first packet (due to the backoff described above), thus reducing the throughput.

Because of this behaviour, a dedicated mechanism to detect when the AP has finished sending all the buffered packets has been designed. First, the client sends a power-save frame and turns off the TX queues. Then, it waits for packets from the AP up to a configured timeout (called Tws). After each received packet, the client waits for Tws milliseconds. If the timeout is triggered, the client sends another power-save frame and proceeds to switching channels.

Now, the algorithm has three variables that control its performance: T - the default time quota on each channel, Tw - the timeout for incoming/outgoing packets while on a channel, and Tws - the timeout for incoming packets after declaring power-save mode. Dynamically adjusting these timers is key to a good behaviour of the algorithm and to achieving good performance. How to calculate these parameters is described below:

- On each interface, try to compute the average interarrival time for downlink packets. Then, compute the mean of all interarrival times on each channel. Pick Tw as the minimum of the means, multiplied by a constant (to adjust for deviations from the mean). This value for Tw is then applied to the next time slot (to force a switch from the channels with suboptimal throughput).

- If the average interarrival time on a certain channel cannot be computed, it would be due to the fact that we did not receive enough packets on that channel during the time slot. Thus, that channel is considered underperforming, and Tw = T is set for the next channel.

- Tws is set to a fixed value (5ms), but it is only applied if, on the previous channel, the entire time quota (T) was spent, and if the entire time quota on the current channel was also spent. Only if both these conditions are true, it is assumed that the AP could have more packets buffered for the given device.

- T is set to a default value of 100ms. However, if a certain channel has significantly greater throughput that any other channels, T is increased in increments of 100ms, up to a maximum of 500ms. If any of the other channels starts offering more than zero throughput, T is reset to the default value.

Through these heuristics of adjusting the timers, a good throughput is achieved in all possible cases of operation, by giving more air time to the interface that have better performance. Tw is an efficient way of fine-tuning the load balancing between interfaces that offer similar throughput, while T is used when bandwidths on various interfaces differ by an order of magnitude or more.

3.2.3 Evaluating the performance of the switching algorithm

In order to measure the performance of the above algorithm, a set of experiments have been devised:

Experiment	AP1	AP2	Switch
AP1	21Mbps	0Mbps	20.6 Mbps
AP2	0 Mbps	21Mbps	20.7Mbps
AP1 & AP2 high bandwidth	21Mbps	21Mbps	19.95Mbps
AP1 & AP2 low bandwidth	6Mbps	6 Mbps	12Mbps
AP1 low signal	9.5Mbps	0Mbps	8.55Mbps
AP2 low signal	0 Mbps	11Mbps	10Mbps
AP1 & AP2 low signal	9.5Mbps	11Mbps	9.9 Mbps

Table 3.2: Performance of channel switching with MPTCP compared to a single AP.

(i) Client connected to a single AP(one interface, no switching):

  (a) Strong signal: maximum throughput

  (b) Weak signal(simulating distance to the AP or wi-fi contention)

(ii) Client connected to 2 APs(one interface, with channel switching)

  (a) Strong signal on both channels

    i. Simultaneous downlink using both APs

      A. Maximum bandwidth

      B. Small bandwidth

    ii. Downlink through one AP only

  (b) Weak signal on both channels

    i. Simultaneous downlink

    ii. Individual downlink

In these static cases, the performance of the proposed system connected to 2 APs have been measured compared to a single AP.

If the wired bandwidth is smaller than the wireless bandwidth, the throughput is measured again, compared to the theoretical maximum (the sum of bandwidths on the 2 channels). If the wireless badnwidth is smaller than the wired bandwidth, the throughput loss given by the channel switching is measured, with the goal to minimize it.

The results of these experiments are reported in Table 3.2:

3.3 MPTCP and data center networks

FatTree [3] and VL2 [10] are recently proposed data centre network topologies that are being deployed into production networks and offer full-bisection bandwidth: a high profile example is Amazon's EC2 Infrastructure-as-a-Service cloud that uses a topology resembling VL2 for their regular instances[19]. Other major players are using or deploying similar networks.

 © TRILOGY2 Consortium 2014

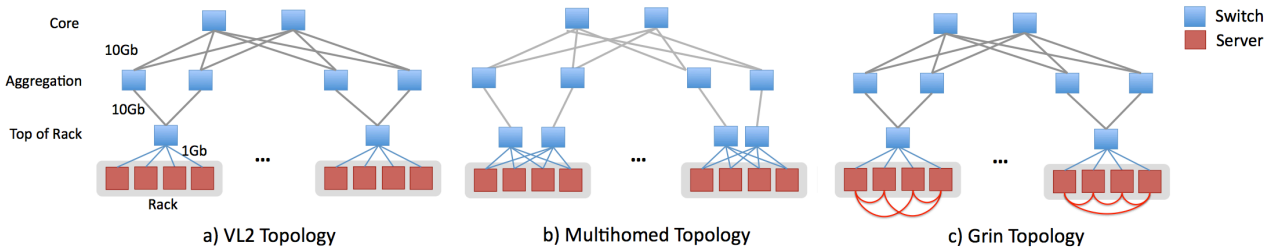

Figure 3.4: Enhancing a VL2 topology

Full-bisection bandwidth networks are appealing because they allow data centre operators and application designers to be mostly agnostic of network topology when deciding how to run distributed algorithms or where to place data. In theory, congestion can only appear on the host access links - by design, the network core should never become a bottleneck. For data intensive algorithms (such as the shuffle phase of map-reduce computation), a full-bisection network offers the best possible performance. On the downside, full-bisection networks incur a larger cost than oversubscribed networks.

Data Centres heavily rely on the concept of *resource pooling*: different applications' workloads are multiplexed onto the hardware, and any application can in principle expand to utilize as many resources as it needs as long as there is capacity anywhere in the data centre. In effect, the resources are pooled in time (when different users access the same machine at different times) and in space (where distributed applications can scale up and down as needed). Also, mechanisms are put in place to ensure their fair use.

Measurement studies [10, 11] show that data centre networks are underutilized. Many links are running hot for certain periods of time, while even more links are idle, which results in an underutilized core. The aim is to extend the resource pooling principle to data centre networks. The goal is simple: when a host wants to send flat-out, it should be able to use as much of the idle capacity of the network as it needs. It should be possible to leverage capacity from everywhere: any flow should be able to fill any part of the network, as long as it wants to do so and the network is underutilized. The effect such network pooling could bring is very appealing: either the network core is fully utilized or there is no application that is bottlenecked by the network. In both cases, the network is providing the best possible performance to the applications.

Achieving this goal is clearly not feasible in today's data centres where hosts connect using a single (typically gigabit) link to the network; this link becomes the bottleneck when hosts want to send flat-out. The question we answer is *how should data centre topologies change to achieve resource pooling?*

We seek solutions that are both cheap and deployable in today's production networks, and apply to any full-bisection network, not just FatTree or VL2. Solutions should preserve the same worst-case bandwidth guarantee provided by full bisection networks, and should ensure that hosts and applications are properly isolated.

A quick study shows that between 1 to 3 free ports should be commonly available in servers today, as all major equipment vendors provide either dual-port or quad-port gigabit NICs in their default blade configurations.

Leveraging this observation, we propose GRIN, a simple change to existing data centre networks where any free port existing in any *server* is randomly connected to a free port of another server in the same rack. Servers can then communicate using a path provided by the original topology, or via one of the servers they are directly connected to. To be able to utilize multiple paths simultaneously, and to advertise additional paths, the servers use MPTCP [18]. This solution can function seamlessly over existing data centre networks.

3.3.1 Problem statement

The main objective is to change existing topologies to allow hosts to utilize idle parts of the network when other hosts are not active. Good solutions share the following properties:

- **Ability to scale:** cost is a major factor that determines what is feasible to deploy in practice. Using more server ports should increase performance and incur little to no additional costs.

- **Fairness and isolation:** access to the shared resource pool must be mediated such that each server gets a fair share of the total bandwidth. Misbehaving servers should be penalized, and they should not adversely affect the performance of the network.

- **Widely applicable:** it should be possible to apply the solution to existing or future networks.

- **Incrementally deployable:** it should be possible to deploy the solutions on live data centre networks with the least possible disruption. This implies hardware or software changes to the network core (including routing algorithms) are out-of-scope. Further, upgrading only an existing subnet should bring appropriate benefits.

Full-bisection networks offer multiple paths between any pair of servers. In the VL2 network shown in Figure 3.4a there are at least four paths between any pair of servers in different racks. Data centre networks[1] give every host a single IP address and leave to the network the task of mapping traffic onto the available paths. The network runs an intra-domain routing algorithm such as OSPF on the aggregation and core switches (effectively these are routers). The algorithm's output consists of multiple equal-cost routes towards servers. When forwarding traffic, the switches use Equal Cost Multipath (ECMP) to hash each connection onto one of the available paths[2]

MPTCP is an evolution of TCP standardized by the IETF and recently adopted by Apple in IOS7 [1]: it takes a TCP connection and splits it across multiple paths, while offering applications the illusion they are still working over TCP. MPTCP has already been proposed as a replacement for TCP in data centres [18]. Its biggest benefit for VL2 and FatTree is avoiding collisions caused by ECMP when multiple connections are placed onto the same congested path, despite the existence of idle capacity elsewhere in the network.

[1] at least as seen from Amazon's EC2 cloud

[2] VLANs are another popular solution to expose multiple paths to hosts. Hosts will have as many addresses as possibly paths, and will implement flow placement on paths.

 © TRILOGY2 Consortium 2014

Any solution devised will have to use multiple NICs at servers. TCP is unable to split a flow across multiple interfaces. As there are few large flows at any time on a given server [10], adding more NICs will not bring performance benefits with regular TCP.

MPTCP, however, allows a transport connection to grow beyond the 1Gbps offered by one NIC port. It also provides by design some of the goals required—it enforces fairness across collections of links [24].

Barring extensive changes to the original topology, the most straightforward solution is to multi-home servers by using additional Top Of the Rack (TOR) switches. A TOR switch is added for every additional server port (see Figure 3.4b), so that each server is connected to each of the multiple TOR switches from its rack. In order to keep the rest of the topology unchanged, the uplinks of the original TOR switch are evenly divided between all the local TOR switches. The resulting topology is oversubscribed, but now each server can potentially use much more bandwidth.

Multihoming brings additional costs in terms of switching equipment, rack-space, energy usage and maintenance. As every additional server port could require an additional switch, this solution does not scale well with the number of server ports used.

To implement multi-homing, each server will receive an additional IP address for every uplink. After addresses are assigned, MPTCP and ECMP are enough to utilize the network: servers only have to choose the destination address for each subflow, and the routing will do the rest.

3.3.2 GRIN

A much better solution is to interconnect servers directly using their free network ports, while keeping the original topology unchanged. Each pair of servers that are directly connected in this manner become neighbors. Intuitively, when a server does not need to use its main network interface, it may allow one or more of its neighbors to borrow it, by forwarding packets received from them (or packets addressed to them) to their final destination. This solution is depicted in Figure 3.4c.

When a server wishes to transmit, it can use both the uplink and the links leading to its neighbors. Conversely, the destination can be reached through both its uplink and via its neighbors. Links used to interconnect servers are called horizontal (or Grin) links, and reserve the term uplinks for those that connect servers to the switch in the original topology. The network interface where the uplink is connected becomes the primary interface of the server, while the others are considered to be secondary interfaces. If every server has the same number n of available network ports, and all of them are used to connect to other servers, than the *degree* of the resulting Grin topology is defined to be equal to n.

3.3.2.1 Server interconnection

It is best to connect those servers that usually do not need to access the network at the same time, otherwise interconnection will not bring major gains beyond improving local throughput.

Many distributed applications have workloads that correlate the idle and busy periods across all the servers they are running on, as do the shuffle or data output phases of map-reduce computations. These applications

tend to distribute their work across many racks to acquire sufficient servers and to improve fault tolerance. Ideally, which nodes run which applications would be known beforehand and the servers would be interconnected running different applications in the hope that the usage peaks from the different applications are spread out in time. However, this is not possible as the same server is assigned different tasks over short time-scales.

Instead, it is assumed that servers in the same rack use their uplinks independently. If this holds true, it is sufficient to randomly interconnect servers in the same rack, which is very easy to cable. Otherwise, two options are possible: either connect servers from different racks, which increases cabling complexity, or change the application schedulers to spread servers across racks as much as possible[3]. Interconnecting servers from pairs of adjacent racks can be a good compromise, if the data centre layout allows it.

3.3.2.2 Path selection

There are a number of options to consider when choosing a path between two random servers, A and B:

- choose one of the paths available in the original topology
- a path may consist entirely of Grin links
- choose any number of intermediate servers, and form a path using the concatenation of all intermediate paths

In most if not all topologies encountered in practice equal cost paths or a small number of different path lengths have been found, depending on the relative position of the communicating hosts in the topology. If Grin is used, the set of paths can grow to include elements consisting of increasingly large number of hops, based on the aforementioned possibilities. A compromise must be found between limiting path length and trying to make the most of the available network capacity. The best way of keeping the path length as short as possible is to allow at most one horizontal hop both after the source and before the destination (giving the possibility of two horizontal hops maximum), which will be called *one hop routing*. This has the definite advantage of permitting the use of simpler routing schemes and requires the least amount of overall forwarding effort. On the other hand, one hop routing also appears to have the least potential of actually using spare network capacity. Thus arises the question of how much of an improvement, if any, can be achieved by increasing the length of horizontal segments in some of the paths candidate for usage.

First action is to see if there is a correlation between the maximum allowed path length and the amount of capacity that can be discovered in a Grin network seen as an abstract model. The proposed representation is agnostic of the original topology, so the idea is to look at it as groups of computers (a group representing one rack) connected to a single, sufficiently large switch. The computers from each group are interconnected in a random manner, having in mind the considerations from the previous section, and the entire setup is then represented as a directed graph. For every pair of elements A and B (where A is a server and B is either

[3] A Multi-homed topology would also rely on the same assumption

 © TRILOGY2 Consortium 2014

a server or the switch) there is one edge going from A to B and one edge going from B to A, each with a capacity of one unit.

For this graph and a given set of source-destination pairs (each pair standing for a connection) GLPK [2] is used to solve the maximum multi-commodity flow problem and a couple of its specializations in which we place restrictions on the number of horizontal hops. Since these tasks tend to be rather computationally intensive, the network model consisted of six groups of twenty servers. The main variables for each experiment were the number of available network ports for each server and the traffic pattern. The Grin interconnection was randomly selected in each case, as previously mentioned.

A minimum of two available network ports (as routing across multiple hops doesn't make sense for any less) and a maximum of six was considered. Larger values lead to a significant increase in evaluation time while not being very likely to appear in a real setting. Four types of traffic patterns were tested:

- *permutation traffic*: each active server sends data to a single destination, and each destination receives data from a single source. For this type of traffic pattern the number of active hosts has been varied from 10% to 70%

- *group traffic*: every server is randomly assigned to a group such that each group contains the same number of servers. A single server is chosen at random from each group as the destination for every other member. The group size ranges from 5 to 60 servers.

- *all-to-all traffic*: each active server sends data to every other active server. The number of communicating hosts varies from 10% to 30%

- *random traffic*: the endpoints of every connection are chosen at random. The number of connections ranges from 10% to 100% of the total number of servers.

The result of each experiment is the total flow in the network. As expected, this is proportional to both the number of active connections and the Grin degree. The same applies for the difference between the one-hop and two-hop restricted versions of the problem, which can go up to around 25% in a topology where six additional ports are used. Figure 3.5(a) shows the average of all experimental results for each case. The two-hop scenario was always within a few percent of the optimal solution, and often managed to provide the best result. This tells that with optimal placement and sizing of a sufficient number of flows, there is no need to consider more than two hops after the source and before the destination.

However, it's quite hard to form an expectation of real-world flow behavior based only on these results, as they do not take into account at least two important limiting factors. Any MPTCP connection will only use a certain number subflows in order to prevent performance degradation [24]. Also, it's not usually possible to make informed decisions about flow placement in real time; instead the hope is that spreading flows among multiple paths together with congestion control can get the most out of the network.

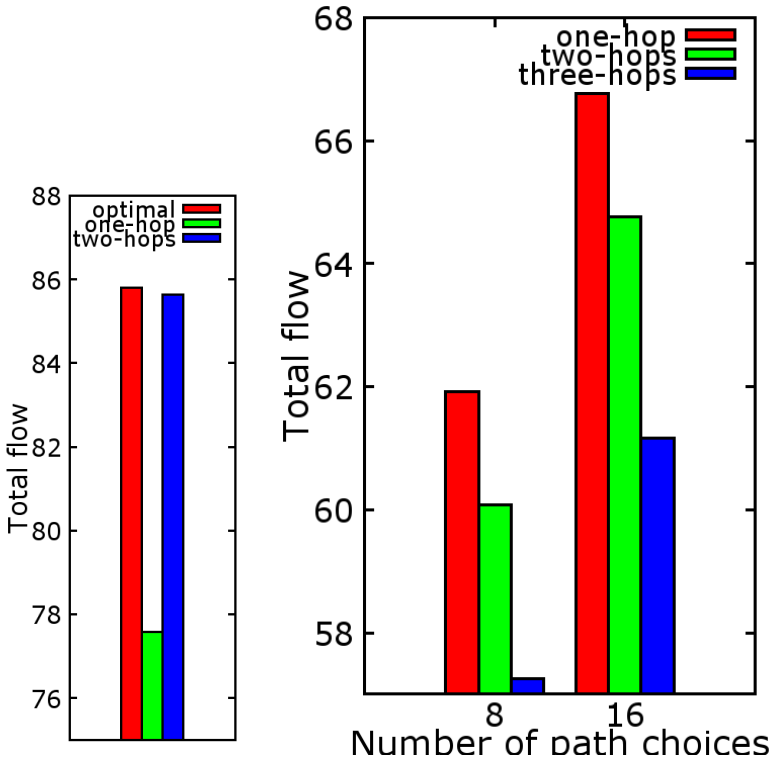

Figure 3.5: Hop Analysis - a) Max MCF (left-side), b) Simple evaluation (right-side)

Taking this into account, a much simpler performance estimation procedure has been devised, using the same network model as before. The first step is to build the complete set of paths between any source and destination from the current traffic matrix. The length of a path is defined as:

- $n - 1$ if the path is made up of n horizontal segments.
- the number of horizontal segments it contains, otherwise. The shortest such path between two servers is called the *direct path*.

For each connection a number of paths without replacement is randomly seleceted and added to the chosen path set; the direct path is always included. The largest possible flow is then assigend to each element of this set, in ascending order of length. This is done by finding the path segment with the least amount of available capacity, and then using that value to fill the entire path.

This method was applied to the same input data as in the previous experiment. An additional variable was the number of available choices for path selection, with two possibilities, either 8 or 16. Three-hop routing was also considered. The average results, represented in figure 3.5(b), show that the one-hop strategy is actually better under these circumstances. This is not entirely unexpected, because smaller Grin degrees make it more difficult for multiple-hop strategies to discover available capacity. This effect is compounded because the sampling was performed on a small subset of the numerous long paths.

Many tests were ran without finding any cases where one-hop routing performed sensibly worse than the other two, while the converse situation came up quite frequently. Adding bias in favor of shorter paths improves

 © TRILOGY2 Consortium 2014

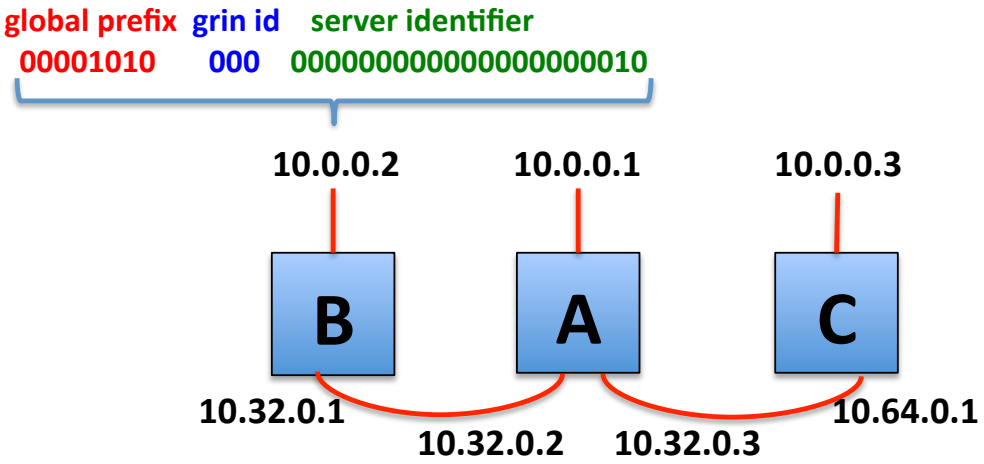

Figure 3.6: Grin Address Assignment Algorithm

the overall results, but doesn't do much to change the standings. The conclusion is that the one-hop strategy is preferable, especially when its advantages in terms of forwarding and complexity are considered.

3.3.2.3 Address assignment

Let assume that each server from the original topology uses an address from the $10.0.0.0/11$ network for its primary interface, and that any other address with the $10.0.0.0/8$ prefix is neither used nor meaningful (different non-routable prefixes can also be used). For any server s, the following are defined:

- $addr_s$ is the address used by its primary interface.
- u_s represents the number of ports that were used to connect s to other servers ($\forall s$, $u_s = 0$ when starting with the original topology).

Any network address $addr$ is split into three meaningful group of bits in the following manner:

- the most significant 8 bits are the *global prefix*, $P(addr)$. This is set to 0x0a for all interfaces.
- the next significant 3 bits represent the *Grin identifier*, $G(addr)$. The value of these bits taken together is going to identify each secondary interface connected to a particular server. Since all the primary interfaces are in the $10.0.0.0/11$ network, their Grin identifier will be equal to zero.
- the remaining 21 bits are the *server identifier*, $I(addr)$, as they represent the unique part of every primary interface address.

For each Grin interface, the corresponding address is determined during the interconnection process. If the interface e, belonging to server a, is connected to server b, then it is going to receive the address $addr_e$, with the following structure:

- $P(addr_e) = 0x0a$

- $G(addr_e) = u_b$
- $I(addr_e) = I(addr_b)$

The value of u_b increases with each additional interconnection involving server b, so it is used as Grin identifier for interface e, because it is guaranteed to be unique among interfaces connected to b. Lastly, the server identifier part of $addr_e$ becomes a reference to server b. For every pair of interconnected secondary interfaces, each one will be designated as default gateway for the other.

This particular addressing scheme is designed to help the routing process, because it allows us to identify the primary address of the neighbor through which a flow directed to a Grin interface must be sent. Thus, for any address, the neighbor function $N(a)$ is defined to have the value of a with the Grin identifier bits set to zero. Consider the example in Figure 3.6 where there are two ports available on each server, identified by e_s and f_s for any particular server s. The initial choice was to connect interfaces e_A and e_B belonging to servers A and B, respectively. Assuming that $addr_A = 10.0.0.1$ and $addr_B = 10.0.0.2$, then the address $10.32.0.2$ was assigned to e_A, and $10.32.0.1$ to e_B. Then servers A and C (with $addr_C = 10.0.0.3$) were connected using interfaces f_A and f_C. Since u_C is currently equal to zero, f_A is going to receive the address $10.32.0.3$, but $u_A = 1$ so the address of e_C will be $10.64.0.1$. This process continues until all available network ports are assigned IP addresses.

3.3.3 Implementation

Let's consider what happens if the base MPTCP implementation is used with a Grin setup. In the example in Figure 3.6, assume that there were no more Grin links except those shown, and that server B attempts to send data to server C. The connection begins like regular TCP using a single flow between the primary interfaces of the two servers. After the initial handshake is complete, server B will be notified via the MPTCP address advertisement mechanism of any additional addresses it can use to contact server C. The set of additional addresses for server C will consist of only one element, $10.64.0.1$. Having this information, server B will attempt to establish a full mesh of subflows between its own addresses (both primary and secondary) and those just received. In this particular case, no additional subflow can be successfully established, as the corresponding destination cannot be reached. This problem must be solved for Grin to work.

The simplest solution would be to use static routes. On every server s, for each secondary interface e in the rest of the network (except those directly connected to s), a route which designates $N(addr_e)$ is added as the gateway. Unfortunately, this approach doesn't scale very well and becomes unwieldy for any reasonably large network.

The best alternative is to use loose source routing. Whenever a TCP connection is initiated in the kernel, algorithm 1 is used to determine the content of the LSRR option for that particular socket.

The algorithm begins with a series of tests meant to determine whether adding the option is necessary. For the first condition on line 5, if the global prefix of the destination address does not match $0x0a$, the connection is considered to be outside the scope of the Grin topology, so no further action is taken. Next check is

 © TRILOGY2 Consortium 2014

Algorithm 1 Set socket IP options

1: $l \leftarrow local\_primary\_address$
2: $s \leftarrow sock.source\_address$
3: $d \leftarrow sock.destination\_address$
4: $H \leftarrow \emptyset$
5: **if** $P(d) \neq 0x0a$ **or** $N(d) = l$ **or** $N(s) = N(d)$ **then**
6: **return**
7: **end if**
8: **if** $G(s) \neq 0$ **then**
9: $H \leftarrow H \cup \{N(s)\}$
10: **end if**
11: **if** $G(d) \neq 0$ **then**
12: $H \leftarrow H \cup \{N(d)\}$
13: **end if**
14: $opt \leftarrow make\_option(LSRR, H)$
15: $set\_option(sock, opt)$

whether there is a direct connection between the destination and the local server. If the source interface is directly connected to the destination, then routing will simply work. If some other local interface is the one connected to the destination, then the present connection becomes invalid because it would contain a loop, so it is disabled by not adding the LSRR option. The last condition verifies whether the source and destination interfaces are directly connected to the same intermediate server. When true, this once more renders source routing unnecessary, as everything will work by default. On line 8 the format of the source address is verified. If it's the address of the primary interface then nothing is done, because it should be reachable from anywhere in the network. If not, then the primary address of its neighbor has to be added to the intermediate hops list. On line 11 this process is repeated for the destination address. Finally, the LSRR option is assembled and applied to the socket.

At first glance it may seem that statement from line 9 is unnecessary. When a connection is established between two secondary interfaces, the source will send packets to its neighbor anyway, because of the default gateway settings. However, the destination must honor the LSRR option by using the same intermediate servers, but in reverse order. Without adding the neighbor of the source to the option, packets on the reverse path are going to get stuck after the first hop, because there is no direct route from that server back to the source.

Source routing has the advantage of being already implemented in the Linux kernel. The overhead in terms of increased IP header size is at most 11 bytes (for two hops). This is considered to be an acceptable tradeoff considering the simplicity of the solution. The Grin implementation requires IP forwarding and source routing to be enabled. Another requirement, less apparent than the previous two, is that reverse path filtering must be disabled. This is caused by the addressing scheme, which can lead to situations where a server receives a packet from a subnet that appears to be accessible through another interface. While all these settings go against recommended general security practices, servers from a Grin network are assumed to be properly insulated against foreign interference.

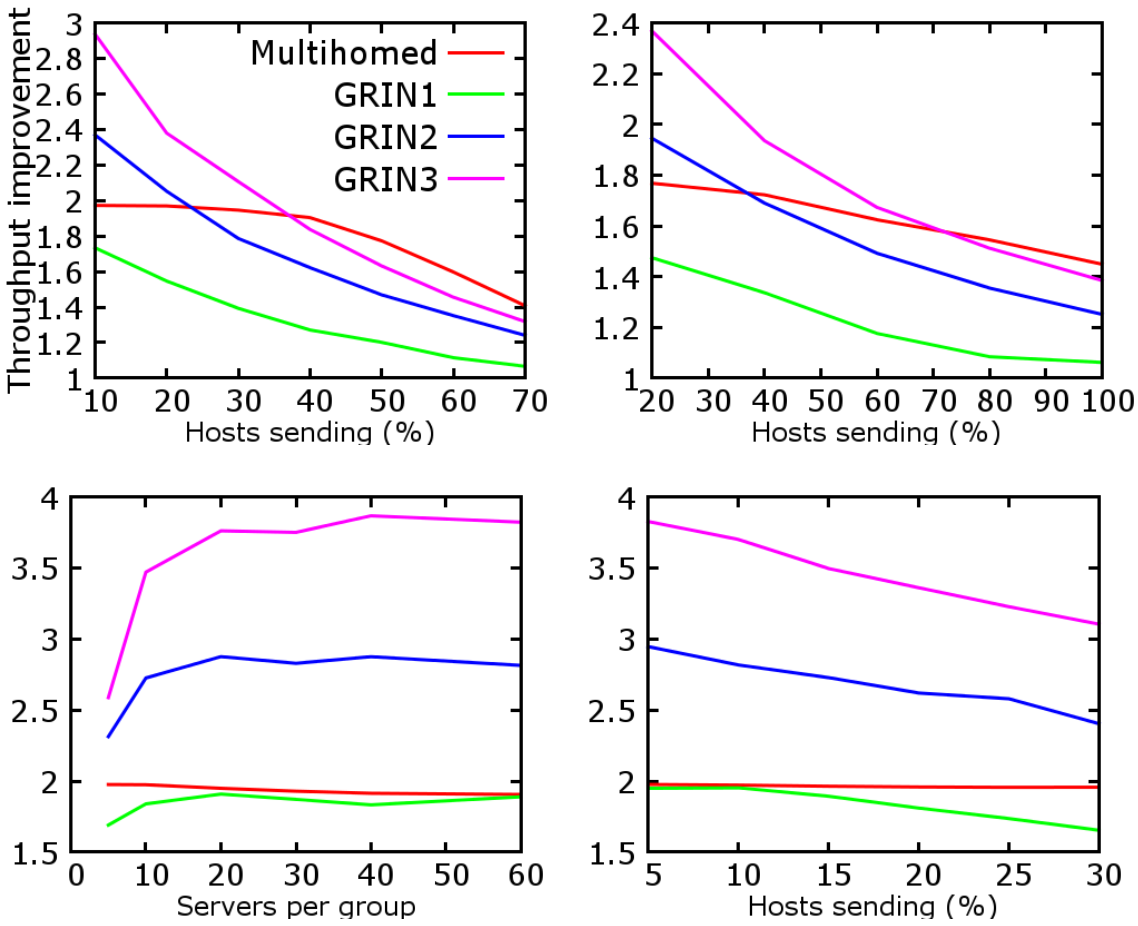

Figure 3.7: Simulation results - Permutation, Random, Group, All-to-all

© TRILOGY2 Consortium 2014

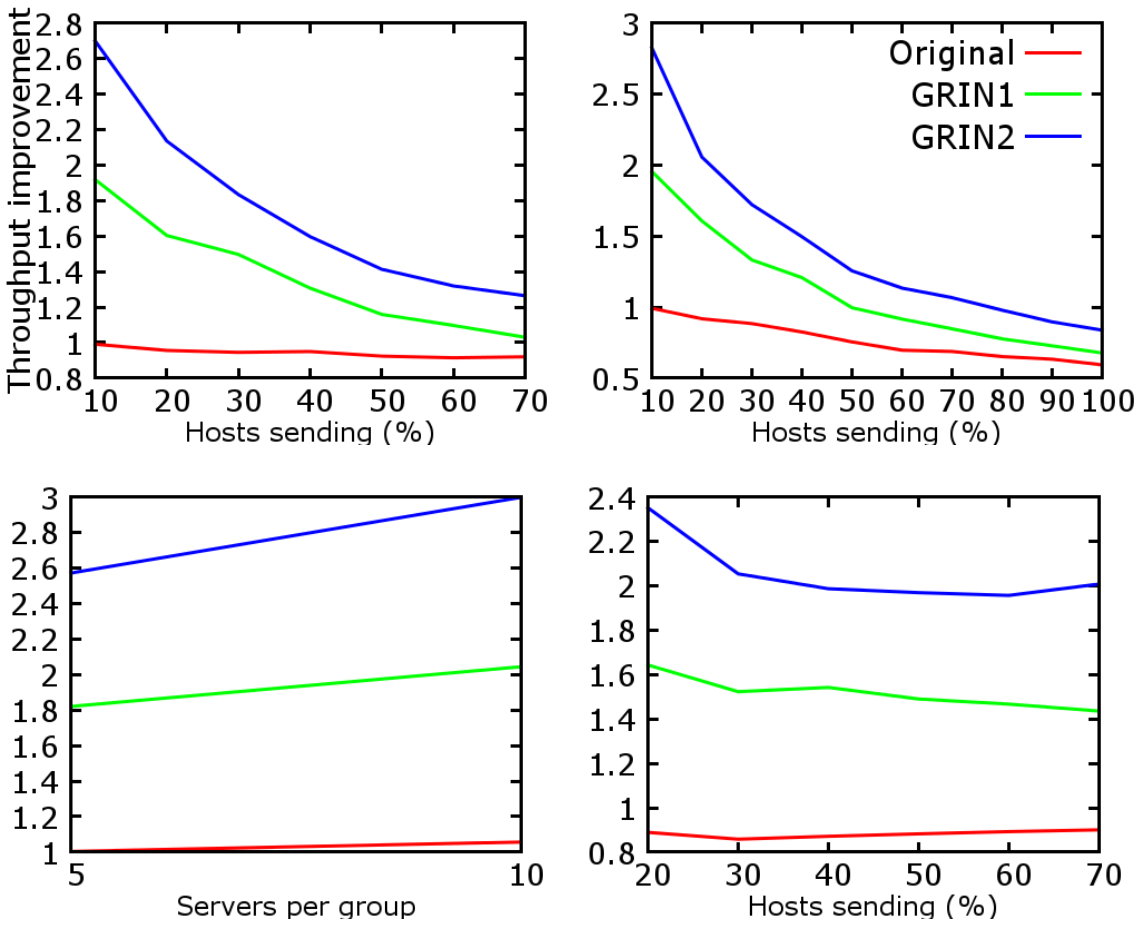

Figure 3.8: Validation results - Permutation, Random, Group, All-to-all

Some other small changes to the MPTCP kernel were also performed. The first is related to subflow initiation. Establishing a full mesh, especially for higher Grin degrees, would involve a very large number of connections. The default behavior for Grin is to establish the smallest number of subflows such that every horizontal link is used at least once. Thus, in a Grin topology with degree n, MPTCP will establish n additional subflows. There is also the option of specifying a certain number of subflows, which are going to be selected at random in a manner consisted with the goals of the default algorithm.

Another modification was to adjust the MPTCP subflow selection process, which decides what subflow to use when sending each particular packet. In most situations, it is desirable to send data using the direct subflow whenever possible, but the RTT estimation alone might not entice MPTCP to make this selection. Bias in favor of the direct connection were also added, based on a simple linear equation with configurable parameters. By default, its estimated RTT is halved for comparison purposes.

3.3.4 Evaluation

Grin has been evaluated using packet level simulations and by deploying it in a small cluster. Multiple Grin topologies are compared with a multi-homed topology in simulation, while on the cluster the simulation results are validated and examined with real-world application performance.

3.3.4.1 Simulation

The simulation studies used a topology consisting of 120 servers, connected to a single switch, with 20 servers in each rack. Increasing network size up to tenfold gave qualitatively similar results. Using the *htsim* simulator, the average data rate of each active receiver was measured and compared to the maximum possible value for the original topology. Each multipath connection is long lived and uses 16 subflows.

Three Grin topologies were tested, using 1 - 3 additional server ports, and one multi-homed topology that uses one additional server port. They are referred to as GRIN1, GRIN2, GRIN3 and Multi-homed, respectively. Multi-homed requires the same total number of ports as GRIN2, costs more and requires additional rack space.

Multi-homed and GRIN topologies were expected to perform significantly better than the original topology on average, and at least as well in the worst case. While traffic patterns usually found in data centre networks quite complex, the hope is to be able to provide at least some insight regarding how will this Grin approach behave in a real-life scenario. Intuitively, the greatest improvements should happen when either a small number of servers are using the network, or the ratio of senders to receivers increases considerably.

The traffic matrices used in these tests were once more permutation, group, all-to-all and random, with different percentages of active servers and numbers of connections per server. 20 random instances of every case were evaluated. Given the small size of the network, only connections between servers from different racks were allowed for permutation and random matrices. In much larger networks, connections chosen at random would seldom end up being local. The assumption is that the network interface is the only limiting factor for the amount of data that can be sent or received by any server.

 © TRILOGY2 Consortium 2014

For a permutation traffic matrix, as illustrated in figure 3.7a, lower percentages of active servers lead to significantly better results for GRIN topologies. The performance of these topologies, however, degrades quicker than that of Multi-homed, which provides better throughput when 25% and 40% of servers, respectively, become active.The group connection matrix simulates scatter-gather communication. This is the most favorable situation for GRIN topologies, because the large number of sources will fill every link of the receiving server. As seen in figure 3.7c, the performance benefit starts out at a minimum for the smallest group size, because of the significant probability of having two or more destinations in close proximity, and then increases to near the maximum value relatively quick. The all-to-all simulations (figure 3.7d) lead to a predictable outcome; the best results are found for the smallest number of active servers, and then gradually decline. The results for Multi-homed decline slower, but also start at a significantly lower value compared to GRIN2 or GRIN3. The final experiments were made for the random traffic matrix. The results, seen in figure 3.7b, resemble those found in the permutation case.

3.3.4.2 Small cluster deployment

Grin has been also deployed in a local cluster, consisting of 10 servers which were used to run the Grin implementation, together with a few other machines for management purposes. Each server has a Xeon E5645 processor, 16 GB of RAM and 4 x 1Gb network interfaces. One interface was used for network boot and another to connect each primary interface to the switch. A GRIN2 topology has been built using the two remaining network ports. By disabling one or both of them the downgrade to GRIN1 was possible as well as the revert to the original setup. It was not possible to deploy a multi-homed topology because of hardware constraints. Each server is also equipped with a slow HDD. This prompted us to rely on an in-memory filesystem whenever the need for persistent storage arisen in any experiment. One particular case involves the Network File System (NFS). NFS cannot export directories out of a tmpfs mount, so we just had to rely on the fact that the operating system will buffer as many previously accessed files as possible in memory. The results are thus an upper bound for performance when disks are involved.

Basic Measurements. The first attempt was to validate the results of the previous simulations, at least to the extent allowed at this much smaller scale. Inputs were generated trying to match the experiments described in section 3.3.4.1, and fed them to a script which would connect to the appropriate servers and launch iperf connections to their corresponding destinations. The duration of a connection is set to 5 seconds. Figure 3.8 shows the findings.

In order to relate to the simulated results, each value was normalized with respect with the rate of 941Mb/s, which is expect to achieve for a single-path, long running, UN-obstructed connection. The results for the permutation traffic pattern match very well, with the implementation being actually better in most cases. However, this is influenced by the fact that some servers will communicate more efficiently based on their proximity, which is not possible in the simulated scenario where intra-rack connections are forbidden. The same holds for the random traffic pattern, but here a somewhat steeper decline in performance is also encoun-

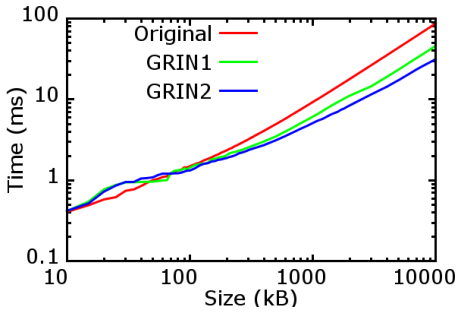

Figure 3.9: Improvements depend on the size of the transfer.

tered as more servers become active, caused by the higher negative impact of two or more sources choosing the same destination. In the all-to-all scenario, things look a bit different. One reason is the fact that the same percentages of active servers could not be replicated given the small size of the network. When taking this into perspective, the results for GRIN1 match reasonably well, while those of GRIN2 are only somewhat worse. The group traffic pattern related experiments were the hardest to enact. Since the network is so small, a single group of 10 servers and two groups of five was only considered. The former case is not really meaningful since it involves a single destination, so the latter is considered, which provides a better result, but once more proximity has taken into account.

The aim was also to find out when Grin starts bringing benefits if there are fixed-size transfers, assuming there is no contention anywhere in the network. A series of tests was ran using a simple client-server program which requests and then receives a certain number of bytes. The results are shown in figure 3.9, and reveal that there is need to transmit data on the order hundreds of kilobytes before any sizeable gain becomes apparent for a single connection. Also, for smaller transfers (between 15-75K), Grin may add at most 200μs to the completion time. This is an issue of the current implementation, caused by the way data is distributed among subflows.

Applications. The next step of the evaluation was to run a couple of real-world applications, starting with an **NFS**[17] server. The goal was to measure the time it took to read every file from an exported directory. The size of each file was variable from one experiment to another, but the contents of the directory always added up to 2GB. As can be seen in figure 3.10, throughput improvement is directly proportional to file size. After a certain point, each file is large enough to make the request overhead almost negligible in relation to the actual transfer duration.

Another application considered is HDFS [21]. This is a natural choice for any Grin setup because it involves handling large amounts of data, potentially causing a lot to gain in terms of performance. One server was used to host the NameNode, 8 were running DataNodes, and the last one acted as a client. The time needed to transfer a 4GB file from HDFS to local storage was measured. GRIN1 gave the best possible result, as the file transferred almost twice as fast. The switch to GRIN2 however, did not bring the expected threefold increase in speed. This was apparently caused by the java process becoming CPU bound. When two transfers in parallel are requested, each process ran on a different core and every network interface was almost fully

 © TRILOGY2 Consortium 2014

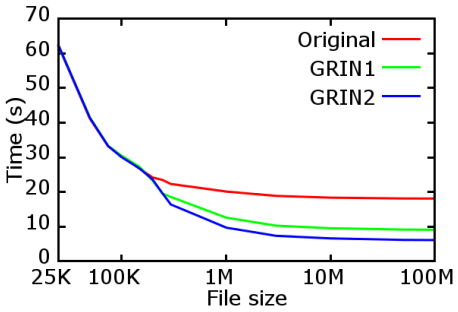

Figure 3.10: NFS

utilized.

Next, the goal is to see if Grin can bring any benefits to virtual machine migration. The Xen Hypervisor[4] version 4.2 was installed on several servers running the modified version of the MPTCP kernel. The virtual machine created for the experiments had 4GB of RAM, and its disk image was shared by an network block device server. The metric used during each test was the time required to migrate the VM to another server. The first attempts, using the xl toolstack, met with complete failure. By default, xl will use ssh to send migration data, which incurs a hefty overhead. Since the data rate hovered around 50MB/s, there was no reason to even begin to consider Grin. At this point we tried to find some way of circumventing the shortcomings caused by ssh in terms of speed. One possibility is to install an extension like High Performance SSH [20] which allows the use of a none-cipher together with various additions designed to improve the performance of ssh connections. Under these circumstances the migration of the VM was performed at a considerably higher speed, but still without using much more bandwidth than a single interface could provide. The best results are obtained by switching back to the xm toolstack, which uses plain TCP. In this case, using GRIN1 increased migration speed by around 60%, while GRIN2 doubled it.

The last application in the test suite was Apache Cassandra [12]. A Cassandra cluster consisting of 9 servers was set up, while the last one acted as client. The goal was to use the Cassandra-stress tool to measure the time required to write a constant amount of data in different circumstances. Default values for most parameters were used, only changing the number of columns to 10, and then varying the size of each column and the numbers of keys inserted such that the total transfer size was around 2GB. The relation between column size and request completion time is presented in figure 3.11. Somewhat unsurprisingly, the way parameters combine o determine the amount of data per row is what matters most in terms of performance. The operations complete much more quickly for a smaller number of larger rows. Increasing the number of columns while decreasing the number of rows will also lead to better results, but not to the same extent as using larger rather than more columns.

Incast. A Grin topology could also be used to lessen the effects of incast under specific conditions. The main idea is that instead of trying to use as many distinct paths as possible, a sender can choose a single path to its destination and only send data that way. With a sufficient number of senders and a random path selection algorithm, the flows should be spread reasonably even between the uplink used by the destination and those

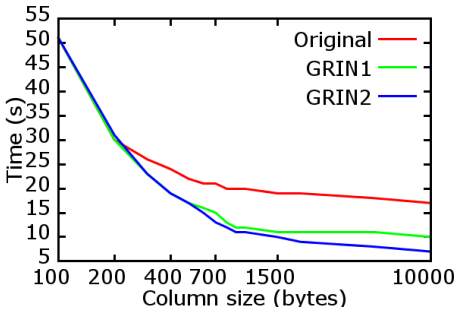

Figure 3.11: Cassandra

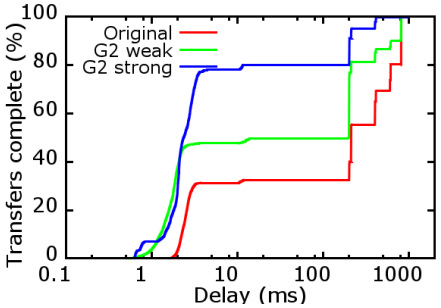

Figure 3.12: Incast mode

of its neighbors. We call this *incast mode*.

Any connection is initiated on the direct path between two servers, so incast mode cannot really help transient connections, because the first packets are always going to be received by the destination directly from the switch (unless we start using some kind of out-of-band address advertisement mechanism). On the other hand, persistent connections which amount to periodic message exchanges will represent the principal use case. The current Grin implementation comes with two version of incast mode for testing purposes: weak affinity and strong affinity. The former will pick a subflow as the preferred one, but is still willing to use other subflows whenever the first temporarily becomes unable to send any more packets. The latter will only use the chosen subflow.

A multi-threaded program was used to periodically request data from multiple sources. Figure 3.12 shows the behavior encountered when one server simultaneously requests 30KB of data over 27 persistent connections, evenly distributed among the nine remaining servers. There are 50 rounds of transfers, each being followed by a 300ms waiting period. Here the original topology is compared with a GRIN2 setup using weak affinity and another one using strong affinity. For each case, the percentage of transfers that were able to complete before a particular time interval has elapsed are represented. As expected, strong affinity gives significantly better results because it enforces the clear separation of connections.

3.3.5 Does Grin Slow Down Local Apps?

Fair Use of Uplinks. With basic Grin routing, every server is going to evenly share its uplink between subflows, whether they are local or not. While this may be acceptable or even desirable in some settings, there are also circumstances where it would be considered completely unfair to the forwarding server. This is especially true in a public data centre, where multiple tenants share the network. In such cases, the aim is to

 © TRILOGY2 Consortium 2014

discriminate against flows with foreign origin when the server is using its uplink.

In order to achieve this goal, a simple scheme based on DiffServ is employed. High priority code point (e.g. EF) is assigned to each direct subflow, while all the others retain the default value. There are two situations to consider. If contention happens at the sender, then packets are buffered locally and priority routing can be used on the primary interface (in Linux, the PRIO qdisc is used) to make sure they are treated according to their priority. On the other hand, when there is contention on the downlink, the network itself has to be able and configured to honor DSCP markings. Fortunately, most commodity switches nowadays support assigning traffic to at least two priority classes, if not more.

Thus, when a server needs to send or receive data flat out then it will be able to fully utilize its uplink if there is no other limiting factor. Also, direct subflows are only going to share the network between themselves, as it would have happened in the original topology. In public clouds, the proper assignment of priorities will be implemented in the hypervisor.

Latency. The goals of high bandwidth and low latency have generally found themselves at odds, so the sight of one while chasing the other must not be lost. Using a Grin topology may generate a number of dangerous situations in terms of latency. For example, a sensitive destination server can be suddenly overwhelmed by a large transfer requested by one of its neighbors. Another issue could be the increased buffer pressure caused by higher network utilization. Also, secondary paths will inherently exhibit higher latency caused by the extra hops. The general goal in this matter becomes to ensure that the latency characteristics of the original topology are not exacerbated; in other words, the aim is to *do no worse*.

Dealing with latency also depends a lot on the particular setting in which Grin is used. The best possible scenario would be a private data centre with priority-aware applications and network stack. This means that whenever data is sent, the application has the possibility of specifying a certain priority for that data, which is then also honored by the network. Under these circumstances, there is no need to do anything special because the most latency-sensitive flows already have the highest priority. In fact, Grin should in theory provide lower delay than the traditional topology, when there is heavy congestion: this was confirmed by htsim simulations.

In the testbed deployment flow completion times have been measured in heavily congestion both in the GRIN topology and the original network. The experimental results were variable, however, giving the same average performance for GRIN and the original topology, and high variance. This is most likely an artefact of the MPTCP Linux kernel implementation.

At the other extreme is a multi-tenant data centre where application priorities are unknown and fairness is paramount. In such cases, it is best to prioritize direct traffic over Grin-routed traffic. Most short connections will finish before an additional MPTCP subflow is setup, so the delay is exactly the same as in the original topology. For longer flows, MPTCP will open additional subflows and there is a danger that latency sensitive data may end up being sent on a low priority path. This will happen because the congestion window of the direct subflow is going to become full at some point, so the MPTCP subflow selection algorithm will have to

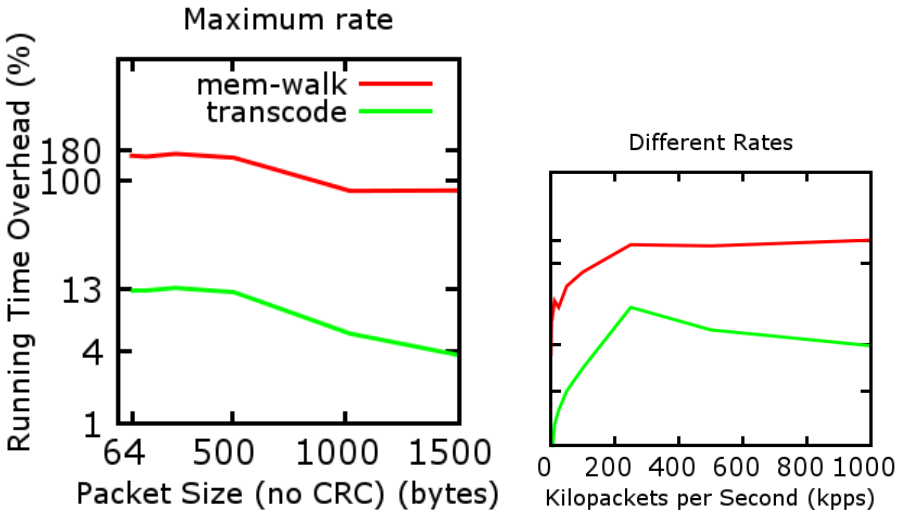

Figure 3.13: Forwarding overhead - 2Gbps throughput, 64-byte payloads

pick a secondary one to send outgoing data. These packets will face a much higher risk of being delayed or lost somewhere in the network. Therefore, in this case, only latency-sensitive flows up to a certain size will exhibit the same characteristics as in the original topology.

To accommodate larger latency-sensitive flows, a sysctl entry is implemented: it specifies the minimum amount of data that has to be successfully acknowledged by a connection before MPTCP is allowed to send data on secondary interfaces. The counter is periodically reset in order to deal with persistent connections. Presently, this is a global option that will affect every connection. However, it could also be made connection-specific, in order to eliminate any potential negative impact it may have on larger flows.

Forwarding overhead. With Grin, servers have to forward packets on behalf of their neighbors, and a valid concern is whether packet forwarding will slow down any running programs. Most of the time, a single server should only forward at most 2Gb/s. This is a positive consequence of using one-hop routing; since forwarded traffic is going to be sent or received through the uplink, congestion control will ensure that the total incoming and outgoing rate cannot exceed that value. [4]

Forwarding has two main sources of overhead: processing interrupts generated by the NIC when packets are received, and memory bandwidth usage to reading packet fields, forwarding table lookups and packet modifications (e.g. decrease TTL or implement Loose Source Routing). To understand these effects, it is not enough to track the CPU usage of forwarding on an otherwise idle machine (which is very low in the tests). Instead, real applications have been ran while measuring their completion time with and without packet forwarding. Experiments with a number of real-world applications gave, as expected, in I/O bound scenarios there are no significant negative consequences. CPU and memory bound applications are more interesting, and results for two representative programs have been collected: *transcode*, a multi-threaded video transcoder that is CPU-bound, and *memwalk*, a synthetic worst-case application that is bound by memory-bandwidth (it

[4] There are also some exceptions which may appear whenever two communicating servers happen to share a neighbor, but this is comparatively rare.

© TRILOGY2 Consortium 2014

has no cache locality).

Transcode heavily utilizes all the cores of the server. Figure 3.13(a) shows that its execution time increases by 4%-13% when the server is forwarding 2 Gb/s. The exact overhead depends on the packet size—larger packet sizes lead to smaller packet rates, and thus smaller overhead.

Memwalk allocates 100MB of memory and runs a long loop where it writes one byte, skips 63 bytes, writes another byte, and so on. Memwalk has no memory locality whatsoever, as every write forces another cache line to be loaded. Memwalk effectively measures the available memory bandwidth, and will be most affected by forwarding. The results in figure 3.13(a) show this effect: memwalk's execution time more than doubles when forwarding small packets.

The second experiment, shown in Figure 3.13(b), only uses minimum-sized packets, varying the packet rate and measuring the effects on the same applications. This experiment emulates an adversarial setup where a server attempts to disrupt the operation of its neighbor. The overhead induced on transcode is now almost entirely caused by interrupts, and peaks around 250kpps. This is where the interface automatically switches to polling mode, drastically reducing the number of interrupts. The difference to the previous experiment is due to the inter-packet gaps that delay the point at which the interface goes into polling mode. As the overhead with polling enabled is only around 13%, one solution could be to enable polling more aggressively. Memwalk suffers much more as the packet rates grow towards 250kpps; interrupt overheads also dominate up until 250kpps.

In summary, the overhead of forwarding on I/O bound and CPU-bound applications is negligible at gigabit speeds, regardless of packet size. Even for heavy CPU-bound apps, the overhead hovers between 4 and 13%.

Memory bandwidth-bound applications will suffer a lot more; while these are uncommon, they do exist: in-memory web-search is a likely candidate. To reduce the effects of packet forwarding, shaping is the obvious choice: packets exceeding a certain rate should be dropped. However, in the case of a denial-of-service attack, the box still has to process receive interrupts, and experiments show that this is just as expensive as doing the whole forwarding. Thus, the only solution is to turn off Grin altogether, by bringing down the interfaces corresponding to horizontal links.

When should Grin be disabled? The number of cache misses is a good indicator of memory performance [5]: by measuring the cache miss rate of the application with forwarding enabled and disabled one can tell whether the application is going to be impacted by Grin forwarding. For instance, the number of cache misses in transcode only varied by a few percent when forwarding packets. In contrast, memwalk generated 8 times more cache misses when forwarding was enabled. This profiling process can be manual (e.g. engineers decide to disable Grin on web-search machines) or even automatic, by measuring cache miss rates and runtime and correlating them with forwarding activity.

3.3.6 Deployment considerations

Grin can be deployed in most of today's data centres, where gigabit multi-port NICs are the norm. Grin has been assumed to run in a private data centre, but it can be easily deployed in public clouds too. In this setting, Grin routing and priorities will be implemented in the hypervisor; the tenant virtual machines only have to use Multipath TCP to be able to take advantage of the multiple paths.

A valid question is whether Grin would be applicable when hosts NICs are upgraded to 10Gbps. Running Grin over 10Gbps is obviously possible, but most applications today are unable to generate traffic at such speeds, and forwarding at 10Gbps is much more expensive for local applications. Running Grin in such settings today will only bring benefits for heavily optimized apps running in clusters where interference is not an issue.

Beyond enhancing existing networks, Grin opens up a new dimension in which data centre topologies can be designed. So far, the only knob has been to choose the size of the core, i.e. the amount of over-subscription. For instance, Jellyfish [22] is a recent proposal that advocates the use of random interconnections for data centre networks and allows provisioning the core capacity with fine granularity. Jellyfish could be used with Grin to build purpose-specific networks where the expected network load is used not only to optimize the size of the core, but also to decide on how many Grin links to use. For example, if a data centre deals predominantly with scatter-gather traffic, the number of Grin links becomes much more important than having a full bisection network.

3.4 Interactions between processing and bandwidth liquidity mechanisms

This section presents a set of mechanisms for accessing bandwidth and CPU resources across devices. The mechanism are based on VM migration, enabling an abstraction layer between user applications and hardware and operating system interfaces.

Virtualization tools enable CPU liquidity across devices, decoupling the operating system from the hardware platform. We can take advantage of these tools by running applications in a virtualized container. Migrating the container from one hardware platform to another also migrates the application, maintaining the application context between the source and the destination. The application context includes process state, CPU register state, memory allocation, open files etc.

Maintaining the network state of the application during migration requires a more advanced set of protocols that decouple the layer 4 socket state from the network configurations. The mechanism presented in this section take advantage of MPTCP to offer a seamless transition of the network state during the application migration.

The first part of this section details the proposed mechanism and the second part presents the prototype implementation of the architecture.

 © TRILOGY2 Consortium 2014

3.4.1 Moving VMs leveraging MPTCP

Since mobile devices have multiple network interfaces both wired and radio-based, the solution for enhancing the live migration of applications takes advantage of these features by using MPTCP. What makes this protocol the best choice for migrating applications is that it spreads the traffic over different interfaces of an end-host, making the transition between them possible without losing connectivity and seamless for the application and thus the user.

VM migration represents one common use-case scenario of MPTCP, that is a host running an application which initiates a fixed Internet connection with a server, issuing a request or downloading data. The Multipath TCP protocol creates one or more subflows, based on the known available interfaces, which appear like a regular TCP connection for the network. This only leads to the conclusion that each of the entities participating in the communication must be MPTCP capable.

The VM where the application is running can be viewed as a mobile host which has a Linux kernel with MPTCP extension. This enables to use the VM on every host device (phone, tablet, PC) which doesn't need to have its kernel modified to support MPTCP, since the connection with the server takes place inside the VM. Based on the number of virtual interfaces present in the VM and the changes in its physical network attachment point, results in acquiring or modifying IP addresses, thus in creating and terminating subflows. To be more precise, the VM attaches to the host through a virtual tap0 interface. Inside the VM, there can be more interfaces created which results in sending to the server, ADD ADDRESS packets with the known IP addresses.

The server hosting the video to be streamed also has to be MPTCP capable and inform the VM with its known IP addresses. If the server's kernel doesn't support MPTCP, a proxy can be used, following the same scenario. After the two end-devices know about their communication capabilities, the corresponding subflows are created between them.

Until this stage, a regular MPTCP connection was used, between a physical server which provides data to the running application and a VM present on a host, which will further be migrated to another host. What makes this migration seamless for the application and permits the connection to stay up, in other words, live migration of the VM, is the break-before-make feature of MPTCP. Break-before-make is a hand-off that permits to detach an interface used by the subflows and attach another one to them, without losing connectivity. This is exactly what happens in the proposed scenario since after migrating the VM on another host, another interface is present, thus another IP. When the migration starts, the connection to server remains open until a significant number of RAM pages are copied to the destination host and then the VM has to be stopped and started at the destination. This is the moment when the MPTCP connection is stalled and for a limited period of time then it can recover from this interruption by establishing a new subflow and continue the transmission. The VM sends other ADD ADDRESS packets informing the server that it has a new interface from the destination host and a new subflow is created, without disturbing the application. The

Figure 3.14: Decoupling applications from the hardware platform

initial MPTCP flow is dropped after a timeout because of inactivity.

This process enables the VM to maintain its active network connections during the migration process. It positively impacts mobility and also enables VM migration between different networks.

3.4.1.1 Prototype implementation

The proposed solution completely decouples the applications from the hardware platform by providing a seamless migration layer (see Figure 3.14). This is achieved by encapsulating each application into a small container composed of an virtual machine that will be live migrated between devices, regardless of type (phones, tables, laptops).

From an architectural point of view, the solutions components are divided into two main categories: VM manager and VM container. The following two sub-sections highlights their main functionalities: a further sub-section also provides a description of the test environment that has been setup for this prototype.

3.4.1.1.1 VM Manager

The VM Manager controls the VM life cycle events like: creation, migration, deletion. It must handle both the user input and interactions with VM managers on other devices.

The VM itself is run into qemu. Due to the nature and particularities of every platform, the implementation of VM Manger must take care of different software frameworks, mainly Linux and Android.

3.4.1.1.1.1 Android

Due to Android ecosystem particularities, the VM Manager will be an application on this platform. The project design in this particular case is similar the one of a regular Android application that has native code.

The SDL on Android provides the necessary abstraction layer between the Android framework graphics, sound and input and the qemu hypervisor. Qemu and depending libraries will be compiled as native code.

3.4.1.1.1.2 Linux

Due to the flexibility and good support of the linux enviroment, the VM Manager on linux is currently

 © TRILOGY2 Consortium 2014

composed of a set of scripts and library dependencies. For example, the SDL library support is requiered on the linux machine.

The qemu hypervisor is controlled from those scripts through a qemu specific protocol: QMP. When QMP is enabled, qemu opens a socket in witch it listens for commands and replies with different status information. This is the way the migration is initiated and also the way the migration process is verified.

3.4.1.1.2 VM Container

The VM Container represents the component that will be migrated. Currently it is represented of a small Linux machine with X11 and a stripped down version of chromium browser.

The Linux kernel is MPTCP capable, enabling seamless TCP connection migration. The browser has both HTML5 and video codecs suitable to open youtube movies and various video streams.

One idea for obtaining a smaller migration time, meaning reduced size of the VM RAM and disk, was to use a very lightweight browser which can run HTML5 applications out-of-the-box, but with limited features. The open-source Chromium project developed and maintained by Google is used: it is basically the base of Chrome browser without other additional third party licensed software.

After a checkout of the latest repo from their development branch, which was highly unstable and continuosly updated, the final decision went for the usage of the latest release (version 33). After compilation, the binary executable occupied 120MB of memory, 60MB of RAM with one tab opened and had a lot of plugins. The aim was to reduce the size of the memory by cutting as many unused features of chromium, maintaining only HTML5 capabilities and multimedia codecs.

The final product achieved is called chromium content shell, which is basically a stripped version of the initial chromium with single process for rendering, no tabs option, debug features and other limited capabilities. Buttons, menus and URL bar were also removed resulting in a white blank canvas which renders web pages. The ffmpeg codecs were installed and the browser scored 462 points for HTML5 capabilities. The final version we achieved occupied only 65MB with a RAM consumption of 25MB.

The solution is working on both Linux and Android operating systems. On Linux, the kernel versions used are 3.5.7 and 3.10.10, both KVM capable. For Android, the IA port on Intel platforms was used, patched to have KVM enabled. The version used is 4.2.2 on Acer Iconia tablets and 4.1.2 on RAZRi phone.

3.4.1.1.3 Proof-of-concept setup

In this section the proof-of-concept setup is described. Two Acert Iconia w700 tablets with Android IA version 4.2.2 have been deployed. Those two tables are connected to a gigabit ethernet switch through USB 3.0 to ethernet gigabit adapters. The tablets are controled through adb from a third station a laptop, as you can see in the following picture. There is also a webserver connect to the topology. This server serves a page that displayes a video stream. The stream is created by vlc from a local video file.

In order to have precise control over the migration process, the virtual machine is managed from command line scripts.

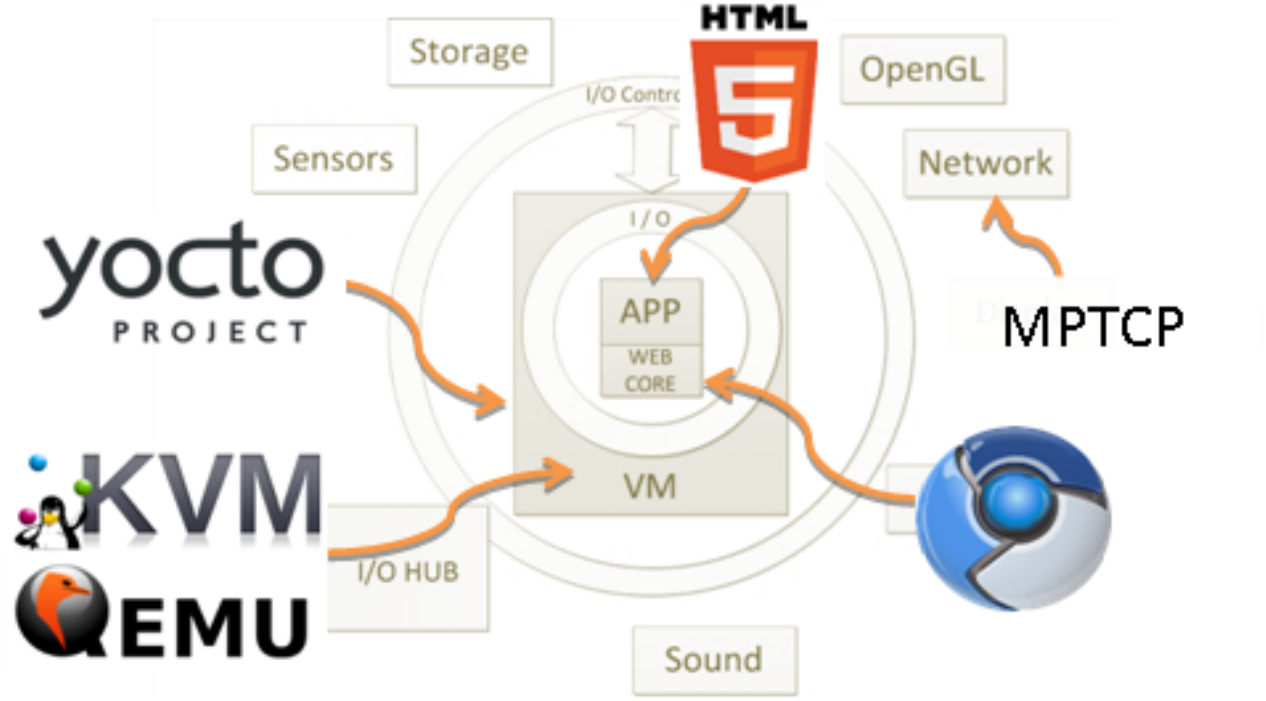

Figure 3.15: Prototype implementation of VM migration across devices

The VM Manager starts on both tabled. The application that is migrated is a video rendering application. On the first tablet the VM manager fires up a VM with the content shell (stripped down version of chromium) browser that opens a web page transmiting a live video stream.

The VM running the video rendering app starts with 320 MB RAM and actually uses 250MB of RAM. The tablets are connected through USB3.0 to gibabit ethernet adapters to the gigabit ethernet switch. The migration lasts 8 seconds. The testing is done using the pre-copy migration technique. Thus, in this 8 seconds interval, the video is continually displayed in the source machine. After the migration,the display is switched to the destination machine and the stream progress continues uninterrupted.

Figure 3.16: Proof-of-concept Topology

© TRILOGY2 Consortium 2014

4 Next steps and conclusions

4.1 Future work

The research work on the cross-liquidity mechanisms described in this document will continue within Trilogy 2, and a lot of future work is planned in two main directions. First, the tools presented in this deliverable will be implemented (if not already complete), upgraded to support new functionalities and also further experimentally evaluated. Moreover, new cross-liquidity mechanisms are currently being investigated, with main focus on the interaction and combination of storage liquidity tools (such as Trevi, described in Deliverable D1.1) and bandwidth pooling mechanisms (mainly MPTCP). The next WP1 deliverables, D1.3 and D1.4, will report the results of this new ongoing research work. With regard to the cross-liquidity tools described in this document, the following are the key future activities and tasks:

- *MPTCP-aware MPLS-TE provider*: The next steps mainly refer to the implementation and deployment of the MP box. Some of the functional components are already available, like the OVSes and the SDN controller. For the latter, a couple of options are being evaluated for deployment in the MP box, mainly OpenDaylight [15] and Floodlight [7] due to their native modular and extensible architecture and full support of OVS and OpenFlow. The internal design and implementation of the MPTCP detector, as a network application on top of the SDN controller also needs to be finalized. The deployment and evaluation of the whole MP box in an emualated MPLS-TE network provider environment will be also considered: a candidate option is to leverage the GMPLS and PCE stacks owned by Nextworks (namely NXW-GMPLS [13] and NXW-PCE [14])

- *MPTCP and channel switching*: The next steps concern the mobility experiments; the plan is to to measure throughput while a mobile client moves between two APs, and compare it to an unmodified client. Measurements of wi-fi handover efficiency will also be considered. Another aspect to be tackled would be the migration of the channel switching code from a laptop to a mobile phone. As the code is only in the generic 802.11 layer in the Linux kernel, porting it to Android is the next logical step. Moreover, measurements to evaluate the power cost of current implementation are also planned.

- *GRIN*: The most interesting and pressing direction for future work regarding GRIN is the transition to 10G networks. This provides some unique challenges as server forwarding effort will greatly increase, and ways to deal with this or to prove that there are circumstances where the tradeoff is acceptable should be found. Morever, using GRIN to optimise topologies for certain dominant traffic patterns is a further interesting topic.

- *MPTCP-enabled virtual machine (VM) migration in mobile devices*: Future work activities include both development and performance tunning activities. From the development point of view, the plan is to enable the usage of the SDL Library on Android. By providing an abstraction layer to graphics,

sound and input, SDL can be used as part of the I/O Controller architecture. From the performance tunning perspective, the plan is to reduce migration time to under 5 seconds, in order to improve the application migration experience for the user. The current project status denotes a migration time of 8 seconds for a VM playing a video stream in content shell. The only performance improvements carried out so far were to obtain a stripped down but functional version of a browser. Future research will include network migration patterns, memory allocation enhancements for qemu and application profiling.

4.2 Concluding remarks

This deliverable provided a description of the initial cross-liquidity tools that have been defined so far in the project. The primary focus has been dedicated to those tools that operate in the bandwidth domain, mainly because bandwidth is the key resource in the current Internet architecture that acts as a glue among all the existing distributed functions, enabling the usage of of heterogeneous pool of resources. These tools fit in the Trilogy 2 architecture by enabling interaction, integration and combination among resource pools in the bandwidth, processing and storage domains. The cross-liquidity mechanisms described in this document, set the basis for the development of a converged Liquid Network architecture for seamless creation and control of heterogeneous resource pools.

© TRILOGY2 Consortium 2014

Bibliography

[1] Apple seems to also believe in multipath tcp. http://perso.uclouvain.be/olivier.bonaventure/blog/html/2013/09/18/mptcp.html.

[2] GNU Linear Programming Kit. http://www.gnu.org/software/glpk/.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM 2010*.

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *In SOSP (2003*, pages 164–177.

[5] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[6] A. Farrel, J.P. Vasseur, and J. Ash. A Path Computation Element (PCE)-Based Architecture - IETF RFC 4655, August 2006.

[7] Project Floodlight. http://www.projectfloodlight.org/floodlight/.

[8] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. RFC 6182: Architectural Guidelines for Multipath TCP Development, March 2011.

[9] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. Rfc6824, IETF, 2013.

[10] Albert Greenberg el al. VL2: a scalable and flexible data center network. In *Proc. ACM Sigcomm 2009*.

[11] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of ACM IMC 2009*.

[12] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[13] Nextworks. NXW-GMPLS. Standard GMPLS stack owned by Nextworks and compliant with the GMPLS architecture specified by IETF and ITU-T ASON project.

[14] Nextworks. NXW-PCE. Standard centralized PCE owned by Nextworks and compliant with the IETF PCE architecture.

[15] OpenDaylight Project. ,http://www.opendaylight.org/.

[16] OpenFlow Switch Specification. https://www.opennetworking.org/sdn-resources/onf-specifications.

[17] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. The nfs version 4 protocol. In *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[18] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. In *Proceedings of ACM SIGCOMM 2011*.

[19] Costin Raiciu, Mihai Ionescu, and Dragos Niculescu. Opening up black box networks with CloudTalk. In *Proceedings of USENIX HotCloud 2012*.

[20] Chris Rapier and Benjamin Bennett. High speed bulk data transfer using the ssh protocol. In *Proceedings of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids: Understanding the spectrum of distributed computing requirements, applications, tools, infrastructures, interoperability, and the incremental adoption of key capabilities*, MG '08, pages 11:1–11:7, New York, NY, USA, 2008. ACM.

[21] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[22] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: networking data centers randomly. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 17–17, Berkeley, CA, USA, 2012. USENIX Association.

[23] Open Networking Foundation white paper. Software-Defined Networking: The New Norm for Networks, April 2012.

[24] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proc. Usenix NSDI 2011*.

 © TRILOGY2 Consortium 2014