ICT-317756

TRILOGY2

Trilogy2: Building the Liquid Net

Specific Targeted Research Project

FP7 ICT Objective 1.1 – The Network of the Future

D3.2 Use Case Development

Due date of deliverable: 1 January 2015

Actual submission date: 15 January 2015

Start date of project	1 January 2012
Duration	36 months
Lead contractor for this deliverable	Intel
Version	v1.0, January 15, 2015
Confidentiality status	Public

Abstract

This document describes a set of use-cases that have been developed within the Trilogy2 project. They allow to achieve previously impossible (or hard to achieve) functions by leveraging the software platforms and cross-liquidity tools as well as their controllers, described in the previous deliverables.
The use-cases are split in three categories, namely mobility, operator infrastructure and wide-area Internet services.

Target Audience

The target audience for this document is the networking research and development community, particularly those with an interest in 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.

Impressum

Full project title	TRILOGY2: Building the Liquid Net
Title of the workpackage	D3.2 Use Case Development
Editor	George Milescu, Intel
Project Co-ordinator	Marcelo Bagnulo Braun, UC3M
Copyright notice	© 2015 Participants in project TRILOGY2

Executive Summary

This document describes use-cases that are developed within the Trilogy2 project. The main purpose of this chapter is to prove that bandwidth, processing or storage liquidity can be achieved with the liquidity tools presented in the previous work. The use-cases are split in three categories, namely *Mobile Devices*, *Operator Infrastructure* and *Wide Area*.

The use-cases for *Liquidity in Mobile Devices* show solutions built for mobile devices with extensive use of Multipath TCP. Virtual machine migration for mobile devices enable mobile apps to be migrated between terminals without losing connectivity or affecting the user experience. Wifi channel switching exposes a novel way of pooling access point resources with MultiPath TCP; this provides an improved user-experience thanks to the additional capacity of the different access points. Finally, security concerns related to trusted resource pooling are addressed. The model proposed associates different trust-levels to different interfaces and different attack scenarios are discussed.

In the *Liquidity in Operator Infrastructure* area, use-cases mostly deal with Network Function Virtualization (NFV). The improvements brought in this section can benefit operators and clients. The control plane for MPLS VPN and IMS virtualized network functions describe the idea of pushing the NFV blocks in data centers, where they can be used in the most efficient way. In addition, the Minicache content delivery solution exposes the idea of using specialized high-performance virtual machines for streaming content. This would allow real time resource balancing and a more efficient use of cloud resources.

Finally, the *Liquidity in the Wide Area* area exposes solutions that allow data replication across the wide area. Irminsule allows hot-fixes and security related information to be propagated fast to production machines. By using a high amount of storage, it can achieve reduced latency which can ensure that the live system gets the data as soon possible. The cloud liquidity area exposes a mechanism that should simplify VM migration from one cloud to another by reducing the amount of data that needs to be transferred.

List of Authors

Authors	Pedro Aranda, Marcelo Bagnulo, Giacomo Bernini, Olivier Bonaventure, Julian Chesterfield, Jaime Garcia-Reinoso, Felipe Huici, Valentin Ilie, Simon Kuenzer, Anil Madhavapeddy, George Milescu, Catalin Moraru, Dragoș Niculescu, Christoph Paasch, Costin Raiciu, John Thomson, Ivan Vidal, Toby Moncaster
Participants	Intel, NEC, NXW, OnApp, TID, UC3M, UCAM, UCL-BE, UPB
Work-package	WP3
Security	Public (PU)
Nature	R
Version	1.0
Total number of pages	59

Contents

Executive Summary 3

List of Authors 4

List of Figures 7

1 Introduction 9

1.1 Structure of the document 9

2 Liquidity in Mobile Devices 10

2.1 Transparent Migration of Stateful Virtual Machines 10

2.1.1 Use-case design 10

2.1.2 Use-case implementation 11

2.1.3 Use-case challenges 12

2.2 WiFi Channel Switching 13

2.2.1 Use-case design 13

2.2.2 Use-case implementation 13

2.2.2.1 Making MPTCP and the WiFi MAC play nicely together 13

2.2.3 Use-case challenges 17

2.2.3.1 Client-side link quality estimation 18

2.2.3.2 ECN Marking 20

2.2.3.3 A mobility experiment 21

2.2.3.4 Static clients 23

2.2.3.5 Fairness 23

2.3 Trusted resource pooling with Multipath TCP 24

2.3.1 Use-case design 24

2.3.1.1 Use-case's interest 25

2.3.2 Use-case implementation 26

2.3.2.1 Protecting the initial handshake 26

2.3.2.2 Protecting the data 27

2.3.2.3 Using trusted interfaces 28

2.3.2.4 Performance evaluation 28

2.3.3 Use-case challenges 29

3 Liquidity in Operator Infrastructure 31

3.1 ClickOS migration 31

3.1.1 Use-case Design 31
3.1.2 Use-case Implementation 32
3.1.3 Use-case challenges 33
3.2 Control Plane for Liquid MPLS VPNs 34
3.2.1 Use-case design 34
3.2.2 Use-case implementation 34
3.2.2.1 The Quagga MPLS-VPN implementation on a Linux Virtual Machine . . 34
3.2.3 Use-case challenges 35
3.3 Providing resilience to IMS virtualised network functions 35
3.3.1 Use-case design 35
3.3.2 Use-case implementation 36
3.3.2.1 Signalling to add/remove virtualised network functions 36
3.3.2.2 IMS signalling procedures 38
3.3.3 Use-case challenges 43
3.4 Minicache video delivery 44
3.4.1 Use-case Design 44
3.4.2 Use-case Implementation 45
3.4.3 Use-case Challenges 47

4 Liquidity in the Wide Area **48**
4.1 Hot-fix propagation with Irminsule 48
4.1.1 Use-case Design 48
4.1.2 Use-case Implementation 48
4.1.3 Use-case Challenges 49
4.2 Cloud Liquidity 50
4.2.1 Use-case design 50
4.2.2 Use-case implementation 51
4.2.2.1 Migration Example 52
4.2.3 Use-case challenges 53

5 Conclusion **56**

References **56**

List of Figures

2.1 Migration Service Architecture 11
2.2 Scenario 1 (left): a client using AP_1 and AP_2 prefers AP_2 because of its more efficient use of airtime. Scenario 2 (right): moving all traffic to AP_2 with its better radio conditions is not the optimal strategy end-to-end 15
2.3 Client-side estimation of PHY delivery probability in 802.11a, fixed rate (54Mbps) 17
2.4 Throughput of a nomadic client at different position between AP1 and AP2 in 802.11n. MPTCP with ECN marking provides 85% of the optimal throughput. 18
2.5 The marking algorithm provides 80% of the optimal throughput over a wide range of RTTs. . 18
2.6 Varying the quality of the AP_1 link has little effect on the throughput experienced by the MPTCP client when marking is active. 19
2.7 Flow throughput across a wide range of parameters. Marking achieves on average 85% of the optimal throughput. 19
2.8 ECN Marking delivers near optimal throughput for scenario 2. 20
2.9 Connecting to more APs reduces the total available throughput. 20
2.10 Throughput of a client moving between AP_1 and AP_2 in 802.11n. MPTCP with ECN marking tracks reasonably close the best AP throughput. 20
2.11 Mobility experiment: indoor client walking speed. 21
2.12 Static clients experience performance variations outside their control; MPTCP offers predictable performance 21
2.13 Experimental setup to test fairness 21
2.14 Mobility simulation: mobile travels twice around a circle with a 30m radius, between 4 APs placed in a square of 80m. 22
2.15 Modified subflow establishment 27
2.16 Using `sendmsg` to force the utilization of trusted subflows 28
2.17 Boxplots show the ratio of page-load time between trusted and regular Multipath TCP; there is no significant difference between both. 30

3.1 System architecture. 32
3.2 Individual boot times for 10,000 ClickOS virtual machines on a single x86 server. 33
3.3 A simplified virtual Customer Premises Equipment (vCPE) architecture 34
3.4 Signalling to start and stop network function elements. 37
3.5 Overview of the proposed architecture. 38
3.6 Validation of the resilient IMS solution 43
3.7 Minicache architecture. 45

3.8 Minicache's main loop and memory allocation scheme. 46
3.9 Minicache's performance. 47

4.1 Overview of the Virtual Machine Migration use case 50
4.2 The list of VM clouds available in London before migrating 52
4.3 The drop-down box shows the possible targets for migration 53
4.4 The list of VM clouds available in Madrid before migrating 53
4.5 The list of VM clouds available in Madrid after migration 53
4.6 VM disk sizes requested on Cloud.net 54
4.7 VM RAM sizes requested on Cloud.net 55
4.8 Number of vCPUs per VM requested on Cloud.net 55

1 Introduction

The deliverables from WP 1 and 2 have described software platforms and tools that enable cross-liquidity as well as the control over these. These enable the implementation of new use-cases that are not possible in today's Internet architecture. In particular, *mobile devices* benefit from the resource pooling capabilities and the increased resilience to failures provided by Multipath TCP. In the *operator infrastructure*, Network Function Virtualisation as well as the usage of lightweight, flexible virtual machines enables the operators to scale their infrastructure more easily. Storage liquidity, provided by tools like Irminsule, enable new use-cases and improve user experience in the wide area.

The work done in deliverable D3.1 is continued in this deliverable. Deliverable D3.1 introduced several use-cases by illustrating their importance and mentioning the related liquidity tools. This deliverable goes one step further by explaining in detail how the liquidity tools are used to implement each use-case. Therefore the list of use-cases is unchanged from D3.1. Each use-case is described in terms of design, implementation and challenges providing an accurate technical view of the solution. The work done in D3.2 will be further continued in D3.3 with the results of deploying these use-cases on trials and testbeds.

1.1 Structure of the document

This document is organized as follows. Section 2 presents use-cases related to Mobile device; Multipath TCP is used in each component to ensure the required network flexibility. Section 3 details use-cases related to the operator's infrastructure by using Network Function Virtualisation (NFV). Finally, Section 4 uses Irminsule to trade storage capacity for reduced latency to quickly provide security and hot-fixes to a large number of end-users. It also introduces wide area block-migration to efficiently move virtual machines across the Internet.

2 Liquidity in Mobile Devices

2.1 Transparent Migration of Stateful Virtual Machines

2.1.1 Use-case design

Video and audio streaming continue to eat an increasing amount of Internet traffic year by year. Recent studies indicate that more than 50% of US traffic is directed to Youtube or Netflix [36]. With the increased popularity of mobile devices, people tend to consume multimedia content on tablets or phones via dedicated streaming apps. Currently, users don't have any mechanism that enable moving running apps between devices, while an internet connection is active. We propose a solution based on a thin virtualization layer between the apps and the underlying operating system that enables the flexibility of pushing applications between devices and, with the help of MPTCP, keeps the network connections active.

In deliverable D3.1, we described app liquidity in mobile devices using vM3 and MPTCP. We will now explore how this use-case is implemented and how well it performs in different real-life scenarios based on network quality factors and restrictions. Virtual machine migration for mobile devices (vM3) is designed to allow users to switch interaction between mobile devices without affecting the overall user- experience or causing network service interruption. Our current architecture allows the user to take his apps with him while on the go, without requiring extensive set-up or configuration to move the applications.

The client video player component benefits from HTML5 features. Its capabilities include the basic user action like playing, pausing, and stopping a live stream. The player ensures a responsive layout that auto adjusts its ratio and quality based on network conditions and measured throughput. Support for keyboard controls lets users control the video with a simple key press. By using the cutting-edge HTML5 API, the player allows adaptive bitrate based on detected network quality.

To ensure proper quality on the client side, multiple codecs need to be taken into account. This adds additional dependencies to our architecture (e.g. libffmpeg). For video live streaming, H264 and VP8 are the most common choices. In contrast with the progressive download technique, live streaming requires a bandwidth capacity somewhat bigger than the data rate of the encoded file. Streaming servers should encode at a date rate that should not create a bottleneck for the network infrastructure. Live streaming refers to content delivered over a network. The most common form of media source is a video camera, recording device or screen capture software. The content delivery network can reside on MPTCP-enabled servers, making it possible for the clients to benefit from the MPTCP mobility options. If the streaming servers aren't located on MPTCP-enabled machines, an MPTCP proxy will be used.

The MPTCP proxy resides between the client and the content delivery networks. It provides support for MPTCP capable hosts to communicate with non-MPTCP capable hosts and acts as an intermediary node that translates the MPTCP requests from the clients to TCP requests that are processed by the streaming server. When an application is being migrated from a device to another, MPTCP's brake-before-make capability will keep the network sockets connected. Based on whether the devices are in the same subnet or net or not, the set-up may need an MPTCP proxy or not. If all the devices are in a single subnet, the architecture is composed of two mobile devices, an access point, a network switch and a streaming server (MPTCP capable). If the streaming server is outside the local network either the streaming server needs to be MPTCP capable or a MPTCP proxy can be added. As nowadays it's quite common that ISPs filter MPTCP requests, the proxy set-up is preferred.

2.1.2 Use-case implementation

For the implementation we choose Android as the underlying operating system as it is open source and can be easily tuned to adapt to our solution. The Android implementation is composed from two main parts: Migration Services(MS) and VMSlot (VMs), as seen in figure 2.1. Migration services manages resources on the mobile device, and exposes capabilities to other nearby devices. Migration Services runs on each devices that implements vM3. MS identifies other devices on the local network that are capable of receiving apps and communicates with them to access their resources without any need for continuous user intervention.

The VMSlot is the app container solution for Android. The solution is based on QEMU and requires KVM

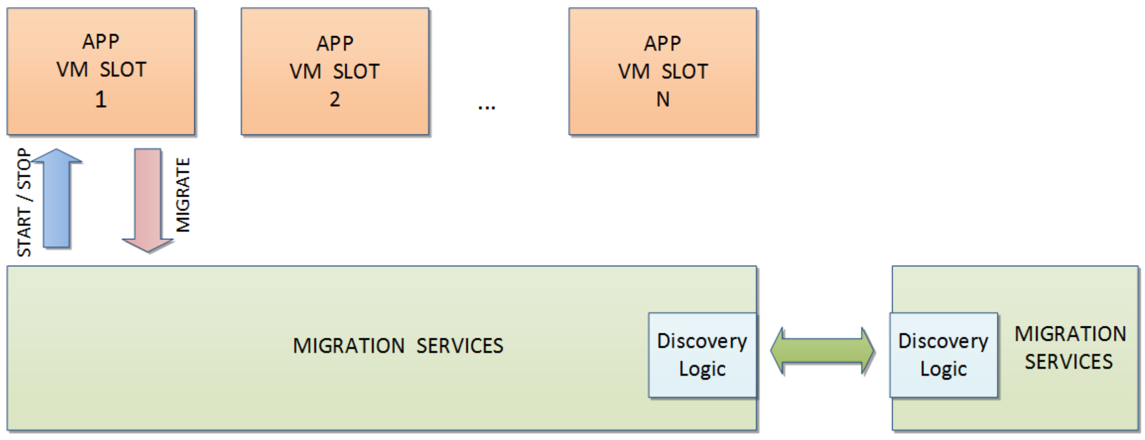


Figure 2.1: Migration Service Architecture

to be enabled in the host OS. By using hardware virtualization technologies, the app container will run with near-native performance, so the user won't see the difference between an Android app and the vM3 solution. The guest OS will start a strip-down version of the Chromium browser that will handle the streaming.

The MS will start and manage VMs, based on each specific app requirements and available hardware. When the user starts an app, a small footprint OS is booted and initialized inside a container, and the web app is initialized afterwards. The MS will manage the containers' IP address, routing and other migration parameters. VMSlot also accounts for apps isolation, based on the fact that in Android an app cannot have more than one instance running at the same time.

When the user wants to initiate migration to another device, a request is sent to MS asking for the list of neighbouring peers. The user can then choose a peer or manually enter the IP address. The MS will negotiate the migration parameters (e.g. throughput, maximum accepted downtime etc.) and will initiate the migration. During the migration, the app still runs on the first device. When the state is fully synchronized, the app shuts down on the first device, and starts on the receiver, resulting in a user- perceived downtime of less than one second.

The content delivery network is a Linux server with MPTCP features enabled. VLC is used to deliver the streaming through the network. To differentiate between multiple streams on the same server, vM3's server component will use different network ports with each stream. Each stream provider can build a custom Android app to enable users to watch their stream, or the stream can be passed manually on boot and start a blank VMSlot with default values of all the streaming parameters.

2.1.3 Use-case challenges

The vM3 use case allows a user to switch interaction between mobile devices without degrading the user experience. One of the main challenges in this solution is the app boot time, as every time the user launches a migratable app, an instance of the guest operating system is booted. In the early stages of the prototype, the booting process took about 20 seconds, which affected the user's perception about the utility and usability of the presented solution. To reduce this boot time, the OS has been stripped down of unnecessary components that were slowing down the process. The browser has been reduced to a tiny webcore and optimized for space to further reduce the overhead and hang time. Even though the app start time has been cut in half, it wasn't fit for production stage yet, as it still consumed more time to start than an average Android native app. To overcome the need to wait for a VM to boot for each app, we opted for a prebooted-VM solution. The system resumes the guest OS from a optimized state - just before the webapp starts, instead of booting the operating system each time. To assemble this solution, we paused the app container ready to render HTML5 apps, allowing it to properly resume from where it left off, without any user-perceived hung time. By using this technique, the overall size of the solution has increased, as a live snapshot of the guest OS occupies more space than a bootable disk. This issue is solved by splitting the live disk into multiple components. The main component contains only the kernel, whereas other components include other userspace items (e.g., X Server, browser etc). With this in place, migration can be tailored. For instance, for native Linux processes without any GUI, the vM3 architecture does not need auxiliary components related to GUIs.

To further improve the migration time, a steady and fast Internet connection needs to be ensured between the clients. Given that migration time mostly depends on the network throughput and app size, a reliable connection is needed. In our benchmarks, we could reach less than two seconds migration time with a 802.11ac connection. Furthermore, specific parameters need to be sent to the virtualization layer back-end to ensure maximum bandwidth is actually used.

Another factor that impacts the migration time is the allowed downtime during migration. In the current architecture, when a migration is initiated, the app on the senders' device continues to run, while pieces of its memory are sent to the receiving device. After a certain threshold, the app is stopped on the sender device, the app states are fully synchronized, and the instance is resumed on the destination device. This allows for the user-perceived downtime to be really small. By setting this time very small, the overall migration time would be big (as the apps need to be almost fully synchronized before stop and resume) but the user-perceived downtime would be absent. If this parameter is set to be high, the migration would be faster but the overall user experience would degrade (e.g. if the user migrates a live stream, a couple of frames would be lost in the process). In our experiments we managed to find specific values depending on the app type and its network/IO activity. As an optimization, a dynamic solution could be enforced. This requires determining the app type at runtime and setting this parameter before the migration, maximizing the overall experience based on user preferences and real-time network quality benchmarks.

2.2 WiFi Channel Switching

2.2.1 Use-case design

In deliverable D3.1, we outlined two methods of providing liquidity in WiFi networks, whenever MPTCP is available on the clients: to associate to all APs visible on one channel, and to switch channels in order to "see" APs on other channels.

2.2.2 Use-case implementation

As mentioned in D3.1, a client that associates to multiple APs and spreads traffic over them with MPTCP will receive close-to-optimal performance in a number of situations (hidden-terminal) but not in all. In particular, the performance achieved in carrier-sense environments is strongly dependent on the rate adaptation algorithms employed by the APs, and these are outside the client's control.

2.2.2.1 Making MPTCP and the WiFi MAC play nicely together

There are two reasons for the suboptimal interaction between the 802.11 MAC and MPTCP:

- Loss rate perceived by MPTCP on each subflow does not reflect the efficiency of the AP routing for that subflow. In cases where packet-level fairness exists between APs, MPTCP sees the same loss rate on all subflows, and is unable to properly balance the load.
- When subflows have shared bottlenecks, MPTCP assumes that sending traffic via the subflows will not affect the bottleneck capacity. This is not the case in single-channel WiFi setups, where sending traffic via a faraway AP will decrease the total throughput.

To fix these problems, we could request that all AP change their rate control algorithms or we could drop APs that decrease total throughput. The former is not only difficult to deploy but will also impact other clients, and the latter may perform worse with mobility.

Our key idea is to use the ability of the MPTCP congestion controller to direct traffic to the most efficient APs. The MPTCP client has access to local WiFi information that it can use to find out which APs are preferable to use, and all it needs to do is inform the sender. One way to relay this information is to have the client drop a percentage of packets but this is not work-conserving. Instead, we rely on explicit congestion notification(ECN) to tell the server to avoid, if possible, the bad APs.

The solution has two distinct parts that we discuss next: first, we present a novel client-side passive measurement technique that allows the client to accurately estimate the efficiency of each AP. Secondly, we present an algorithm that decides the amount of ECN marks that a client can set on a subflow that enables efficient airtime usage.

2.2.2.1.1 Measuring AP Efficiency When deciding which AP it should prefer, the client needs to understand how long it takes on average for that AP to successfully send one packet, assuming the AP is alone in accessing the wireless medium. We call this measure packet time and use T_i to denote AP i's packet time. Our metric purposely ignores the effect other WiFi clients may have on the average transmission time (e.g. by contending for the medium), or other clients that are serviced by the same AP. By comparing the resulting packet times, the client can decide which AP is preferable to use, and can signal the sender to steer traffic away from the other APs via ECN.

In contrast to previous work, we only care about the **hypothetical wireless bandwidth** from each AP to the client, as some of the interference from other APs is created by the client itself, so actual wireless bandwidth is not a good estimator.

To estimate the packet time T we measure at the client the bitrate b used by the AP and the physical loss rate p as discussed below. We feed them through the following analytical model of 802.11a/g where R is the number of retransmissions are made per packet so that the packet delivery probability is almost 1:

$$T = \sum_{i=0}^{R}[(\frac{MSS}{b} + K) \cdot (i+1) + C \cdot \sum_{j=0}^{i} \cdot 2^j] \cdot p^i \cdot (1-p) \tag{2.1}$$

In the model above, the first term measures the packet transmission time including the airtime used and K accounts for different WiFi constants such as SIFS, DIFS and the time needed to send the ACK at the lowest rate (2Mbps). The term $C \cdot ...$ measures the time spent due to the contention interval, and models its increase on successive frame losses. Finally, the whole term is moderated by the probability that i retransmissions are needed to successfully send the packet.

The client knows the bitrate used by the AP, however estimating the PHY loss rate is more difficult because it can only observe successful transmissions; for each successful transmission there may be an unknown number of retransmissions, which conceal the physical loss rate. Thus the obvious formula $delivery\_prob = \frac{N_{received}}{N_{total}}$ cannot be used at the client, as N_{total} is unknown.

We avoid this problem by leveraging the "retry" bit present in the MAC header of every frame, signaling whether the frame is a retransmission. The client counts the number of frames received with the retry bit set to 0, N_0. All the other frames reaching the client will have the retry bit set to 1, and it counts them as N_1. We

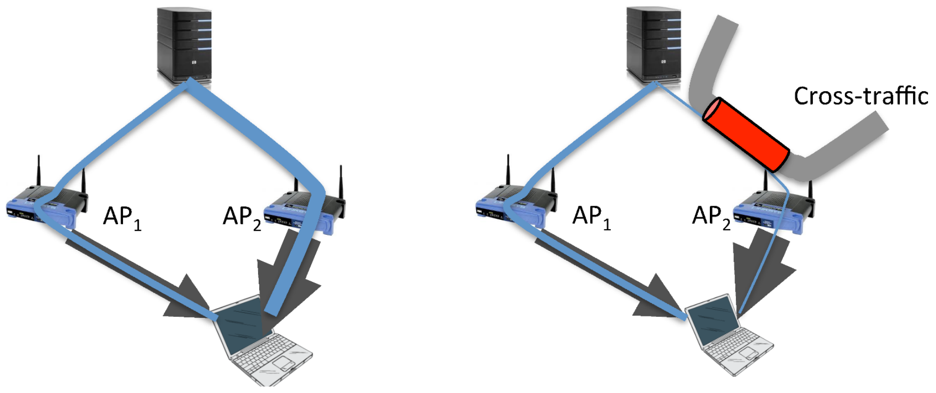


Figure 2.2: Scenario 1 (left): a client using AP_1 and AP_2 prefers AP_2 because of its more efficient use of airtime. Scenario 2 (right): moving all traffic to AP_2 with its better radio conditions is not the optimal strategy end-to-end

recast the previous formula to measure the delivery probability only for frames that are delivered on the first attempt:

$$delivery\_prob = \frac{N_0}{N_0 + N_1 + N_{lost}} \tag{2.2}$$

The term N_{lost} captures packets that were not delivered by the AP despite many successive attempts and can be measured by using the sequence number present in the MAC header. The client remembers the largest sequence number seen, and checks it against the sequence number in a newly received packet to find out the number of packets, if any, the AP failed to transmit since the last successful transmission.

To estimate accurately the delivery probability for all APs on a channel, the client maintains a FIFO queue of packets seen in a chosen interval, recording for each packet the time of arrival, its sequence number and its retry bit (10B in total). When new packets are received, or when timers expire, the packets outside the interval are purged, and N_0, N_1 and N_{lost} of the corresponding AP are modified accordingly. Our implementation uses an interval of 500ms, which results in an overhead per channel of around 10KB for 802.11a/g, and 100-200KB for 802.11n with four spatial streams.

2.2.2.1.2 Helping senders make the right choice Consider the two scenarios depicted in Figure 2.2, where AP_2's packet time is shorter than AP_1's, and let's consider how the MPTCP sender behaves if the two subflows going via AP_1 and AP_2 do not interfere at the last hop. MPTCP congestion control [43] requires that it does no worse than TCP on the best available path, and it efficiently uses network resources. MPTCP achieves the first goal by continuously estimating the throughput TCP would get on its paths. The estimates use loss rates and RTTs of that path in a simple model of TCP throughput, namely $B = \sqrt{\frac{2}{p}} \cdot \frac{MSS}{RTT}$. Using this estimate, MPTCP adjusts its overall aggressiveness (total congestion window increase per RTT over all its subflows) to that it achieves at least the throughput of the best TCP. To achieve the second goal, MPTCP gives more of the increase to subflows with smaller loss rates.

In scenario 1, the throughput via AP_2 is higher than AP_1, resulting in a lower loss rate on the corresponding subflow and making the MPTCP sender send most of its traffic via AP_2. In scenario 2, other bottlenecks reduce the throughput available via AP_2, and the load balancing of traffic over paths will depend on the amount of loss experienced on AP_2. Either way, MPTCP will use its estimate of throughput available over AP_1 to ensure it achieves at least as much in aggregate.

Our target is to help MPTCP achieve the same goals when the two subflows via AP_1 and AP_2 interfere. For this to happen, we need to signal the sender via explicit congestion notification that AP_1 is worse and should be avoided when possible. Just making traffic move away from AP_1 is simple: the client will simply mark a large fraction of packets (e.g. 10%) received via AP_1 with the Congestion Experienced codepoint, which will duly push the sender to balance most of its traffic via AP_2. However, this approach will backfire in scenario 2, where MPTCP will stick to AP_2 and receive worse throughput.

To achieve the proper behavior in all these scenarios, the rate of ECN marks sent over AP_1 must be chosen carefully such that it does not affect MPTCP's estimation of TCP throughput via AP_1. Our goal is to ensure that the **MPTCP connection gets at least as much throughput as it would get via AP_1 if the latter is completely idle**. In particular, say the rate of ECN marks the client adds is δ. As the TCP congestion window depends on loss rate, the congestion window will decrease when we increase the loss rate. For the bandwidth to remain constant, we would like RTT_δ, the RTT after marking, to also decrease. In other words we would like for the following equation to hold:

$$B = \sqrt{\frac{2}{p}} \cdot \frac{MSS}{RTT} = \sqrt{\frac{2}{p+\delta}} \cdot \frac{MSS}{RTT_\delta} \tag{2.3}$$

We assume the subflow via AP_1 is the unique subflow at that AP; congestion control at the sender will keep AP_1's buffer half-full on average. Thus, the average RTT observed by the client can be decomposed as $RTT = RTT_0 + \frac{BUF}{2} \cdot T_1$, where RTT_0 is the path RTT, and the second term accounts for buffering. Note that we use T_1, the average packet delivery time for AP_1 estimated by our metric. If our client reduced its RTT to RTT_0 by decreasing its congestion window, it would still be able to fill the pipe, and more importantly it would allow the sender to correctly estimate the bandwidth via AP_1. Using these observations and simplifying the terms, we rewrite the equation above as:

$$B = \sqrt{\frac{1}{p}} \cdot \frac{1}{RTT} = \sqrt{\frac{1}{p+\delta}} \cdot \frac{1}{RTT - T_1 \cdot \frac{BUF}{2}} \tag{2.4}$$

Finally, knowing T_1 gives us an estimate of the maximum bandwidth $B = \frac{1}{T_1}$. We now have two equations with three unknowns: p, δ and BUF. Fortunately, we don't need to know the exact value of BUF; using a smaller value will only lead to a smaller value for δ, reducing our ability to steer traffic away from AP_1. To get an estimate of BUF, we note that nearly all wireless APs are buffered to offer 802.11a/g capacity (25Mbps) to single clients downloading from servers spread across the globe (i.e. an RTT of at least 100ms). This implies the smallest buffers should be around 2.5Mbit, which is about 200 packets of 1500 bytes. We

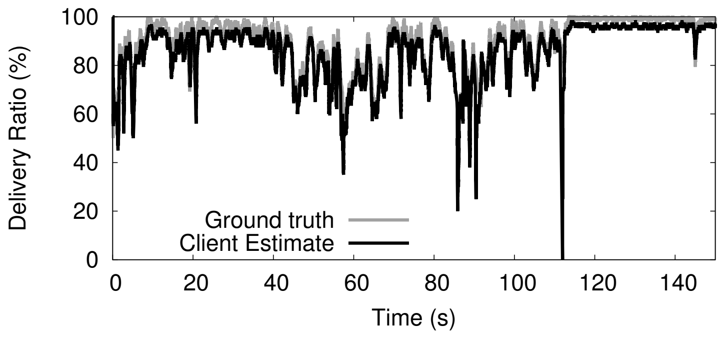


Figure 2.3: Client-side estimation of PHY delivery probability in 802.11a, fixed rate (54Mbps)

use 200 as our estimate for BUF, and can now solve the two equations for δ. The closed form we arrive at is:

$$\delta = \frac{1}{2} \cdot \left(\frac{50 \cdot T_1^2}{RTT \cdot (RTT - 50 \cdot T_1)}\right)^2 \tag{2.5}$$

δ depends on the interface (T_1) and the RTT of the subflow that will be marked, both of which are available at the client. Note that δ provides a maximum safe marking rate, and the actual marking rate used may be lower. For instance, marking rates in excess of 5% brings the TCP congestion window down to around 6 packets and makes TCP prone to timeouts.

In our implementation, the client computes the estimation of δ for every AP it is connected to. The client monitors the values of T_i for all of its interfaces, and sets ECN marks for subflows routed via interfaces with a packet time at least 20% larger than the best packet time across all interfaces. The 20% threshold is chosen to avoid penalizing good APs for short fluctuations in performance, while marking inefficient APs. The ECN marking happens before the L2 packets get delivered to L3 processing code at the client.

2.2.3 Use-case challenges

We have implemented our solution in the Linux 3.5.0 kernel, patched with MPTCP v0.89. Most of the changes are in the 802.11 part of the Linux kernel, and are independent of the actual NIC used. The patch has 1.3KLOC, and it includes code to compute the packet time for each wireless interface, the ECN marking algorithm, and channel switching support.

In this section we analyze each of our contributions in detail both experimentally and, where appropriate, in simulation. We first measure our client-side link estimation technique in a controlled environment. Next, we analyze the marking algorithm using 802.11n in the lab, and extensively in simulation to find it provides close to optimal throughput (90%) over a wide range of parameters. Next, we analyze different facets of our proposal including fairness to existing clients, performance for short downloads and measuring energy consumption. Finally, we run preliminary mobility tests "in-the-wild" using APs we do not control, finding that our solution does provide near-optimal throughput in real deployments.

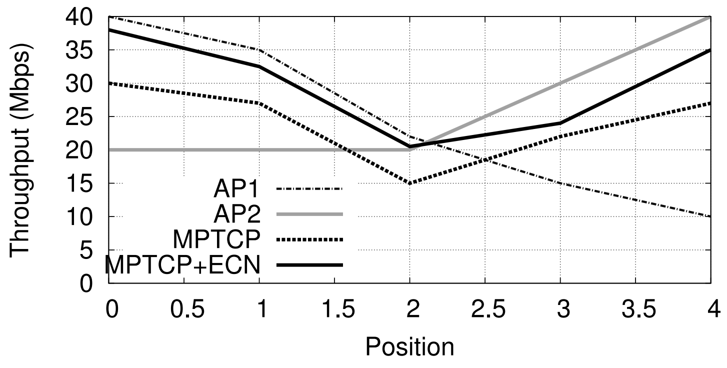


Figure 2.4: Throughput of a nomadic client at different position between AP1 and AP2 in 802.11n. MPTCP with ECN marking provides 85% of the optimal throughput.

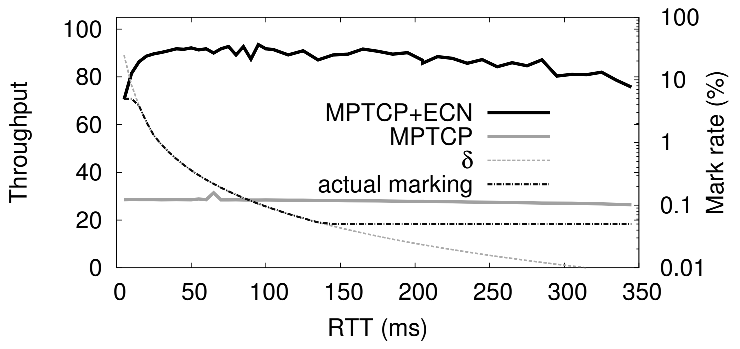


Figure 2.5: The marking algorithm provides 80% of the optimal throughput over a wide range of RTTs.

2.2.3.1 Client-side link quality estimation

To test the accuracy of our client-side estimation of PHY delivery probability, we setup one of our APs in 802.11a, place a sniffer within 1m of the AP, and place the client around 10m away from the AP. The AP's rate control algorithm is disabled, and we manually set the MCS to 54Mbps.

Both the client and the sniffer measure the average delivery ratio over a half-second window. The size of the window is a parameter of our algorithm: larger values take longer to adapt to changing channel conditions, while smaller values may result in very noisy estimations. Half a second provides a good tradeoff between noise and speed of adaptation.

The client downloads a large file and we plot the client's estimation of the delivery probability (relation (2.2)), and compare it against the ground truth, as observed at the packet sniffer near the sender. Two minutes into the experiment we increase the transmit power of the AP to the max, thus improving the delivery probability. The results are given in Figure 2.3 and show that the client's estimation closely tracks the actual delivery ratio, and the mean error across the whole test is around 3%. We ran many experiments with 802.11g/n and observed similar behavior: client side estimation closely tracks the ground truth, and the mean error rate was under 5% in all our experiments.

Our metric is based on the assumption that the delivery ratio is independent of the state of the packet (the number of retries). This assumption is reasonable when packet losses occur due to channel conditions, but

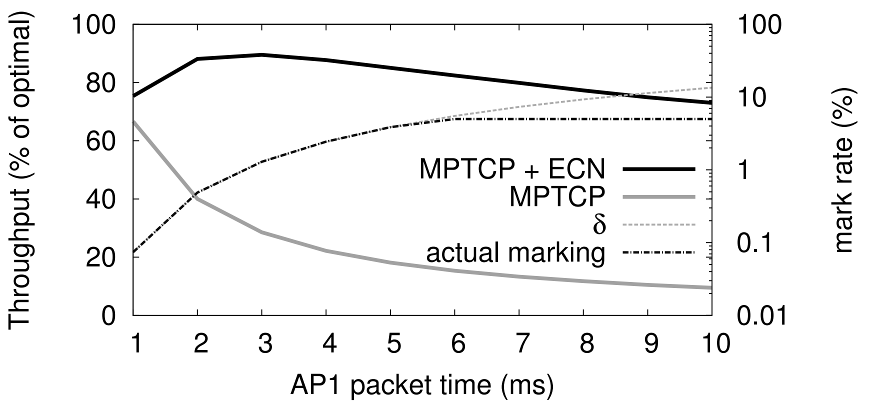


Figure 2.6: Varying the quality of the AP_1 link has little effect on the throughput experienced by the MPTCP client when marking is active.

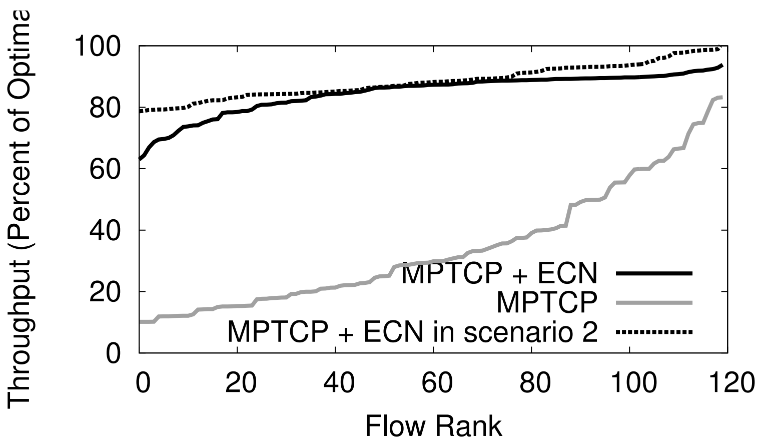


Figure 2.7: Flow throughput across a wide range of parameters. Marking achieves on average 85% of the optimal throughput.

breaks down in hidden terminal situations, where a collision on the first packet will most likely trigger collisions on subsequent packets. In such cases, our experiments show the metric's focus only on the initial transmissions will lead to errors, as follows:

- When competing traffic is sparse, our metric will overestimate the PHY delivery probability by around 10% in our tests.
- Under heavy contention, one AP may be starved completely, and our client's estimate will be unreliable.

This drawback does not affect our overall solution: we need client-side estimation only when the two APs are in carrier sense. When in hidden terminal, our experiments show that the interaction between the MAC and Multipath TCP leads to a proper throughput allocation, and no further intervention via ECN is needed. When a rate control algorithm picks a different rate to resend the same packet, that packet will not have its "retry" bit set despite being retransmitted. To understand whether this affects our results, we ran experiments as above but with rate control enabled; however the were no discernible differences in the efficacy of our algorithm.

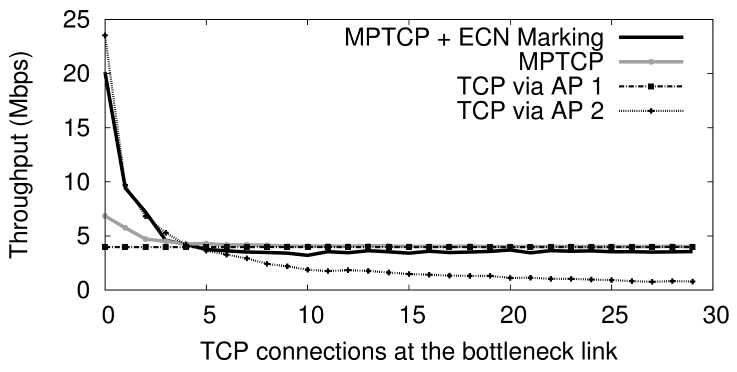


Figure 2.8: ECN Marking delivers near optimal throughput for scenario 2.

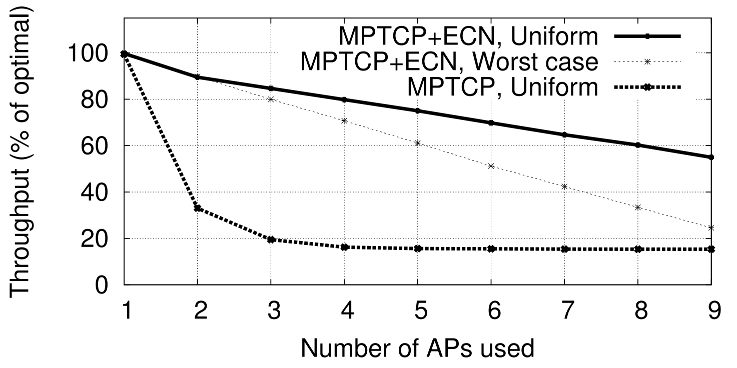


Figure 2.9: Connecting to more APs reduces the total available throughput.

2.2.3.2 ECN Marking

We reran all the 802.11a/g experiments presented so far with our client-side load balancing algorithm on. We found that the marking did not influence the actual results: in particular, we verified that marking was not triggered in the channel diversity setup we discussed before.

For a static 802.11n client, we applied the ECN marking as indicated by relation (2.5). The results shown in Fig. 2.4 reveal that our metric and ECN algorithms work well together, pushing traffic away from the inefficient AP. Using the same setup, we then moved the client at walking speed between the APs; the whole distance was covered in around 10s. The results (not shown) are much noisier, but show that the ECN

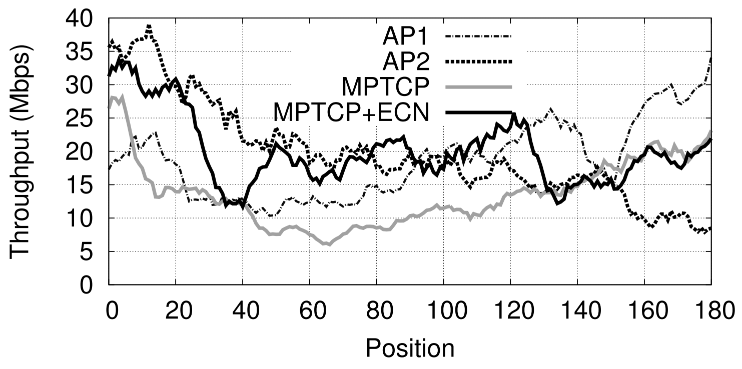


Figure 2.10: Throughput of a client moving between AP_1 and AP_2 in 802.11n. MPTCP with ECN marking tracks reasonably close the best AP throughput.

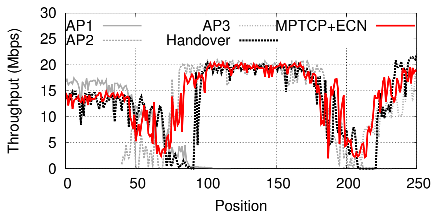


Figure 2.11: Mobility experiment: indoor client walking speed.

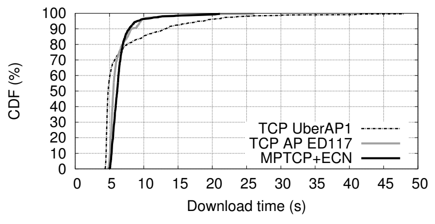


Figure 2.12: Static clients experience performance variations outside their control; MPTCP offers predictable performance

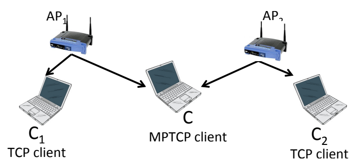


Figure 2.13: Experimental setup to test fairness

mechanism still works fairly well overall; a similar result with a mix of 11n and 11g is later discussed in Figure 2.11. All further experiments presented in this deliverable are run with the ECN algorithm enabled, unless otherwise specified.

2.2.3.3 A mobility experiment

We now discuss a mobility experiment we ran in a university building with mixed WiFi coverage: the user starts from our lab situated on the second floor of the building, goes down a flight of stairs and then walks over to the cafeteria. En route, the mobile client (a laptop) is locked on channel 6 and can associate to five different APs, each covering different parts of the route. We repeat the experiment by doing consecutive runs (each run takes around 1 minute or so), and in each run the client is either:

- using one AP at a time, with standard TCP.
- using MPTCP and associating to all APs it sees all the time.
- performing handover between the different APs: the client monitors the beacons it sees over a 2s period, and switches to a new AP when it receives Δ more beacons than the current AP; Δ was chosen experimentally to be 10 to avoid flapping between APs. MPTCP is still used to enable the connection to survive losing its current IP.

We run an experiment where the client slowly moves through the building at a speed of about 1km/h, and the results are shown in Figure 2.11. At the beginning of the walk, the client has access to the two APs in our lab (Linux APs, ath9k, minstrel), and these are also accessible briefly on the stairs (in the middle of the trace). Our departments' deployment of uncoordinated APs (Fortinet FAP 220B) are available both in our lab at very low strength, on the stairs and closer to the cafeteria. Our mobile client connects to at most three APs simultaneously.

Throughout the walk, the throughput of our MPTCP mobile client closely tracks that of TCP running on the best AP in any given point; the handover solution suffers because it is forced to be conservative at crossover points between APs. In such positions throughput drops to nearly zero for 5-10s. We also noticed that in the beginning of the trace our ECN-based load balancing algorithm penalizes the subflow over the department AP—if we disable it, the throughput of MPTCP in that area drops dramatically.

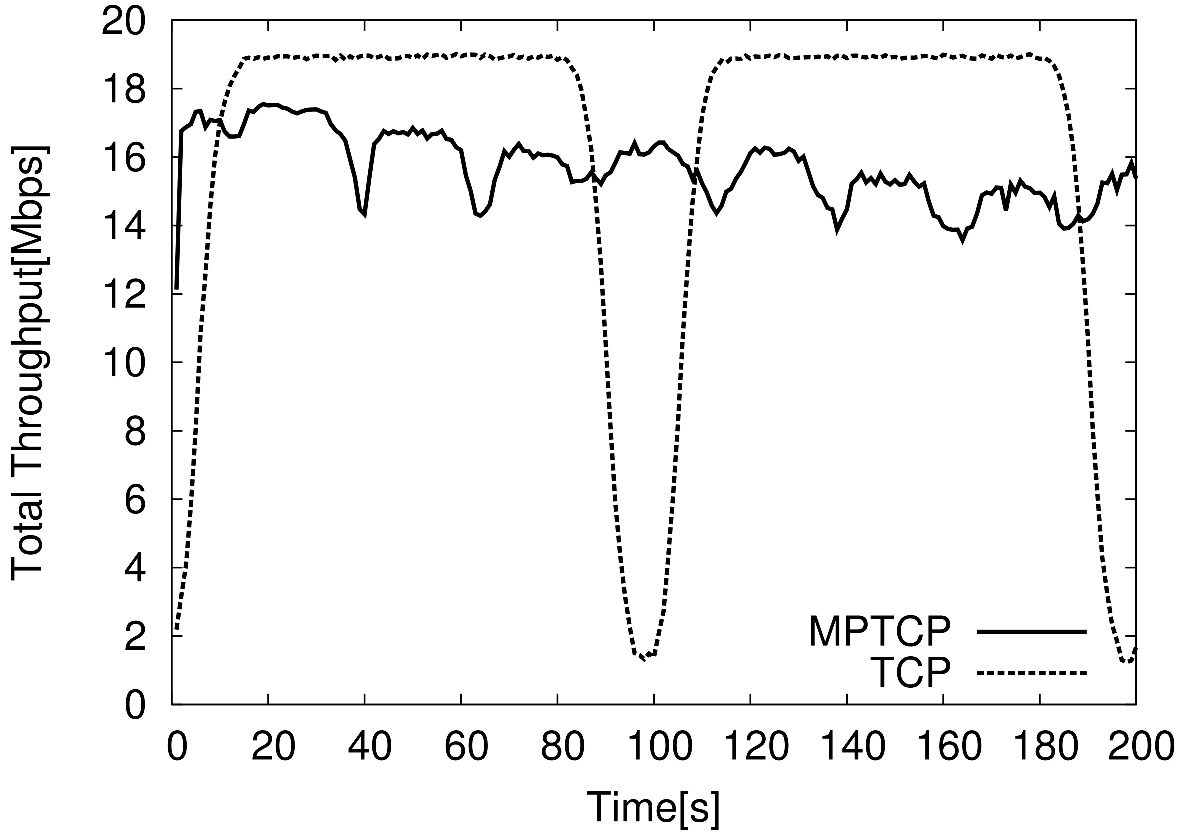


Figure 2.14: Mobility simulation: mobile travels twice around a circle with a 30m radius, between 4 APs placed in a square of 80m.

C conn. to	AP_1-C_1	AP_2-C_2	C
AP_1(TCP)	5	10	5
AP_2(TCP)	10	5	5
AP_1&AP_2	7	7	7

Table 2.1: APs&clients in close range: MPTCP provides perfect fairness (802.11a, throughput in Mbps).

C conn. to	AP_1-C_1	AP_2-C_2	C
AP_1(TCP)	3.5	13	3.5
AP_2(TCP)	10	5	5
AP_1&AP_2	10	5	5

Table 2.2: Client close to AP_2: MPTCP client behaves as TCP connected to AP_2

C conn. to	AP_1-C_1	AP_2-C_2	C
AP_1(TCP)	3.5	13.5	3
AP_2(TCP)	14	3	3
AP_1&AP_2	8.5	6.5	5

Table 2.3: Client in-between APs: MPTCP client improves overall fairness

To evaluate how downloading from different APs works in a richer setup, we simulated (in ns3/dce with Linux 3.14 kernel, MPTCP v.89) a topology with four APs placed in a square with the side of 80m, and a mobile circling twice at walking speed inside the rectangle. As the mobile only uses MCS of 54Mbps, and starts near its main AP, there is a significant drop in the TCP throughput as it reaches the far side, as shown in 2.14. MPTCP is able to harvest bandwidth from the closest AP, and it also shows small bumps in performance whenever it closes to one of the APs.

2.2.3.4 Static clients

Our experiments so far show that connecting to multiple APs pays off when mobile. Is it worth doing the same when the client is static? We had our client connect to two APs (channel 11) visible in our lab and that we do not control, and the performance from both APs is similar. Our client repeatedly downloads the same 10MB file from one of our servers using either TCP over AP1, TCP over AP2 or MPTCP+ECN over both APs. We ran this experiment during 5 office hours, and present a CDF of the throughputs obtained in Figure 2.12. The figure shows there is a long tail in the throughput obtain via either AP because of external factors we do not control: other WiFi users, varying channel conditions, etc. The median download time for AP1 is 5s, 5.6s via AP2 and 6s with MPTCP (20% worse). However, MPTCP reduces the tail, cutting the 99% download time in half.

While connected to different APs, the solution adds the following per AP costs: association and authentication handshakes, DHCP, and the null frames required whenever the mobile goes in and out of power save. These are negligible, as the bulk of the cost is due to the radio processing and the TCP/IP stack [15]. The energy cost of our solution is therefore dependent on the actual throughput achieved, which is near-optimal.

2.2.3.5 Fairness

We have focused our analysis so far on the performance experienced by a client connecting to multiple APs, and showed that there are significant benefits to be had from this approach. We haven't analyzed the impact this solution has on other WiFi clients: is it fair to them? Does our solution decrease the aggregate throughput of the system? In answering these questions, our goals are neither absolute fairness (since WiFi is a completely distributed protocol and also inherently unfair, this goal in unattainable) nor maximizing aggregate throughput (which may mean some distant clients are starved). Rather, we want to ensure our solution's impact on other clients is *no worse than that of a TCP connection using the best available AP*.

The theory of Multipath TCP congestion control reassures us that two or more MPTCP subflows sharing the same wireless medium will not get more throughput than a single TCP connection would. Also, if an AP has more customers, Multipath TCP will tend to steer traffic away from that AP because it sees a higher packet loss rate.

We used the simple setup shown in Figure 2.13 to test the behavior of our proposal. There are two APs each with a TCP client in their close vicinity, using 802.11a, and an MPTCP client C using both APs.

The first test we run has both APs using maximum power: when alone, all clients will achieve the maximum rate in 802.11a, around 22Mbps. The results of the experiment are shown in table 2.1: when the client connects to both APs, the system achieves perfect fairness. In comparison, connecting to either AP alone will lead to an unfair bandwidth allocation. In this setup, MPTCP congestion control matters. If we use regular TCP congestion control on each subflow, the resulting setup is unfair: the MPTCP client receives 10Mbps while the two TCP clients each get 5Mbps.

We next study another instance of the setup in Fig. 2.13 where the APs, still in CS, are farther away, and while the TCP clients remain close to their respective APs, they get a lesser channel to the opposite AP. First, we place C close to AP_2. When C connects to AP_1, which is farther, it harms the throughput of C_1, and the overall fairness is greatly reduced. When C connects to both APs, its traffic flows via the better path through AP_2, offering optimal throughput without harming fairness (Table 2.2). When the client is between the two APs, traffic is split on both paths and the overall fairness is improved, while also increasing the throughput obtained by our client (Table 2.3).

In summary, by connecting to both APs at the same time and splitting the downlink traffic between them, MPTCP achieves better fairness in most cases, and never hurts traffic more than a TCP connection would when using the best AP.

2.3 Trusted resource pooling with Multipath TCP

2.3.1 Use-case design

Smartphones and tablets are becoming one of the most widely used devices to access the Internet. Today's smartphones are equipped with several wireless interfaces (WiFi, 3G/4G, Bluetooth, . . .). Faced with a huge growth of the data traffic [10, 19], mobile network operators are exploring alternatives to 3G/4G to provide Internet connectivity.

Researchers have explored the interactions between WiFi and 3G in the past. Several studies have demonstrated that there are performance and cost benefits in offloading data traffic to the WiFi network [23]. These findings encouraged network operators to roll out large WiFi networks. For example, FON[1] gathers more than twelve million WiFi access points, most of these being controlled by network operators.

From a pure cost viewpoint, users and network operators could wish to offload their traffic onto WiFi networks. However, WiFi networks also have some drawbacks. First, a WiFi network may be much smaller than 3G, in particular when many users are attached to a low bandwidth broadband link. Second, using WiFi may expose the users to more types of attacks and security problems than 3G/4G networks. Cellular networks

[1] `http://www.fon.com`

are typically controlled by the network operators and it is difficult for attackers to capture or inject packets inside these networks. On the other hand, WiFi started as a completely open technology and there are still many open access points where all data packets can be easily eavesdropped. Users attached to open WiFi networks are vulnerable to a wide range of attacks such as the Firesheep[2] Firefox extension that allows hijacking of HTTP sessions. Furthermore, many ADSL/cable routers that often provide WiFi access have often been the target of attacks, some having compromised hundreds of thousands of routers (see e.g. [28, 17]). Once compromised, such a WiFi router can easily mount various types of man in the middle attacks.

Application-level encryption with SSL/TLS is considered by many to be the best option to improve the security of the Internet [21]. However, these encryption techniques are still not deployed everywhere. In 2012, we surveyed the top 10,000 Alexa web sites and found that 38% of them supported HTTPS. A recent survey over the Alexa top 1,000,000 web sites revealed that only 45% of them supported HTTPS [37]. We clearly cannot assume that SSL/TLS will be used everywhere and measurement studies on smartphones show that plaintext protocols continue to dominate [10, 19, 12].

Given that usually the user can trust its mobile network operator (and 3G/4G includes protocols to authenticate the mobile network), in this use-case we propose to distinguish two types of network interfaces :

- **Trusted interface**. An interface is considered to be **trusted** when the user can expect that passive and active eavesdropping will be impossible on this interface given its nature (e.g. physical wire) or due to the utilization of encryption techniques.

- **Untrusted interface**. An interface is consider to be **untrusted** if an attacker can easily eavesdrop packets.

We thus suggest to schedule traffic across the different interfaces in such a way that the security-critical or privacy-sensitive data are always sent over the trusted interfaces, while the other traffic can be sent on untrusted interfaces. An attacker would need to be able to eavesdrop the traffic from the trusted interfaces in order to launch an attack. While this does not provide guaranteed security for the traffic, it makes the attacker's life a bit harder.

2.3.1.1 Use-case's interest

The bandwidth liquidity offered by multiple paths between two end-hosts has already been exploited in many ways by using Multipath TCP. This use-case adds another dimension to the network's liquidity by considering the different trust-levels one can have in each path.

Additionally, privacy concerns are rising nowadays. The Internet's users are demanding for more secure and private communication across the Internet. Leveraging Multipath TCP's capability to send traffic across different paths is an easy way to make an attacker's life harder.

[2]See `http://codebutler.github.io/firesheep/`

2.3.2 Use-case implementation

Multipath TCP and its Linux Kernel implementation (described in the Deliverable 1.1 on Software Platforms) was designed with resource pooling in mind [42] and aims at distributing data fairly over different interfaces. We use Multipath TCP in order to achieve the above described security goal. Multipath TCP allows to achieve this as it is able to transmit a single data stream via different paths. In the current Internet's protocol-suite, such multipath transmission is only possible with SCTP-CMT [20]. However, widespread deployment has still not been possible with SCTP due to its numerous issues with middlebox traversal.

The ability to steer traffic within Multipath TCP along a specific subflow is possible thanks to the scheduler framework, described in Deliverable 1.3. In our Linux Kernel implementation, we extend the interfaces table in the Linux kernel with one per bit interface that indicates its trust level. This trust level will typically be automatically configured by the application, usually the connection manager, that controls the utilization of the network interfaces. We expect that wired interfaces such as Ethernet could be considered to be trusted by default. However, some companies could prefer to consider that only the company's Ethernet is trusted and rely on 802.1x to verify that the device is attached to the company network. Virtual Private Network solutions built with IPSec, SSL/TLS or DTLS usually provide a virtual network interface. Such interfaces will be considered as trusted by the connection manager. For wireless networks, we expect that 3G and 4G networks will be considered to be trusted given the utilization of link layer encryption to secure the wireless channel. Mobile network operators often install tailored connection managers on the smartphones that they sell. This connection manager could easily recognize the operator's networks and consider them to be trusted. For WiFi networks, the level of trust could depend on the use of link-layer encryption (e.g. WPA2 could be considered trusted while WEP would not be) and also on the utilization of 802.1x (e.g. a corporate WiFi network using EAP-TTLS would be considered to be trusted once the network certificate has been validated).

2.3.2.1 Protecting the initial handshake

With the level of trust of each interface in mind, we need to reconsider how the TCP/IP stack is using the available interfaces. In the existing Linux TCP/IP implementation, one interface is considered to be preferred. On smartphones, this is usually the WiFi interface when active given that WiFi usually[3] provides a lower delay and higher throughput than cellular networks.

Our first modification to the Multipath TCP implementation is to take the trust level of the interfaces into account when creating a Multipath TCP connection. When a client opens a Multipath TCP connection, our implementation forces the transmission of the first `SYN` packet over a trusted interface. This protects the random keys that are included in the `MP_CAPABLE` option against passive eavesdroppers. Once the initial subflow has been created over a trusted interface, additional subflows can be established over the other available interfaces.

Configuring the trust level of each interface on the client is necessary but unfortunately not sufficient. Since

[3]This might change in cellular networks supporting 4G/LTE that provide lower delays [19].

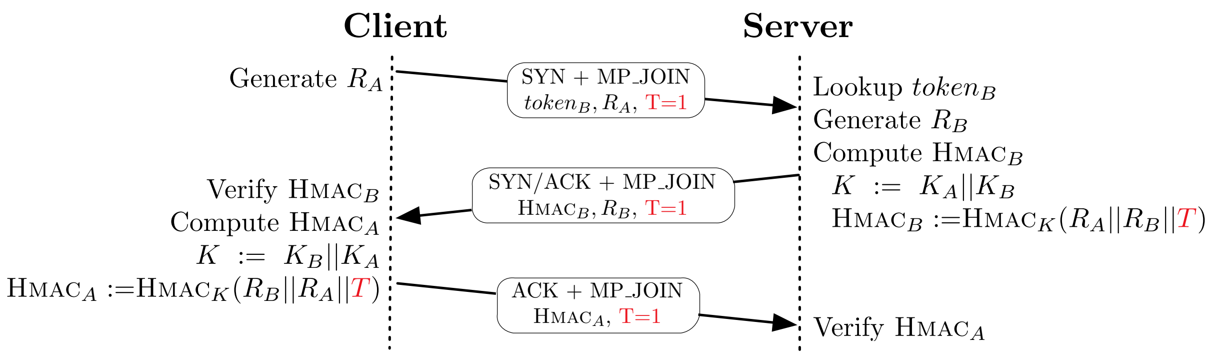


Figure 2.15: Modified subflow establishment

both the client and the server will exchange data, there must be a way for the client to reliably inform the server about the trust level of the different subflows. We first assume that the initial subflow is always established over a trusted interface. We further assume that the server only uses trusted interfaces. The second subflow could be established over a trusted or an untrusted interface. To convey the trust level of the interface used on the client to create the subflow, we extend the `MP_JOIN` option with the T bit. When set, this bit indicates that the subflow is created on an trusted interface on the client. Otherwise the interface (and thus the subflow) is considered to be untrusted. To prevent active attacks from e.g. a rogue WiFi access point, the value of the T bit is included in the HMAC computation used to authenticate the subflow.

2.3.2.2 Protecting the data

Protecting the initial handshake does not protect the data. This is particularly important with plain text protocols such as HTTP. On smartphones, HTTP remains the most widely used application [10, 19, 12]. Multipath TCP cannot encrypt the data to protect it, but applications can decide to send sensitive data only over trusted interfaces. For HTTP, we need to distinguish between the HTTP headers (request and response) and the web objects that are exchanged. The HTTP headers often contain sensitive data such as cookies or other forms of session identifiers [38].

To enable applications to request the transmission of sensitive data only over trusted interfaces, we extend the Multipath TCP implementation by overloading the `sendmsg` system call. This system call allows transmission of data on a socket while supporting scatter-gather. This system call allows the application to specify some flags when sending data. Our implementation defines the new `MPTCP_TRUSTED` flag. When this flag is set, the buffer passed through the `sendmsg` system call must only be transmitted over trusted interfaces. The `send` and `sendto` system calls can be modified in a similar way to support the `MPTCP_TRUSTED` flag.

When an application calls `sendmsg` and requires the transmission of data over a trusted interface, the stack iterates over the available subflows to find one that is trusted and has enough space in its congestion window to send the data. If necessary, the transmission of the data is delayed until a trusted subflow becomes available. Although our implementation forces the data sent with the modified `sendmsg` system call to be transmitted

```
...
/*This message must be sent on a trusted interface*/
msg.msg_iov->iov_base = secure_content;
msg.msg_iov->iov_len = length_of_secure_content;
sendmsg(sockfd,&msg,MPTCP_TRUSTED);

/*This message may be sent on any interface*/
msg.msg_iov->iov_base = unsecure_content;
msg.msg_iov->iov_len = length_of_unsecure_content;
sendmsg(sockfd,&msg,0);
...
```

Figure 2.16: Using `sendmsg` to force the utilization of trusted subflows

over a trusted subflow, it does not force this data to be transmitted in a (sequence of) independent packets. Other data may be attached before or after these packets.

2.3.2.3 Using trusted interfaces

SSL and TLS are much more secure than plaintext protocols, but they are far from perfect and there are situations where the Multipath TCP extensions discussed in the previous section could also be applied with SSL/TLS. SSL and TLS rely on server certificates to authenticate the servers. The validation of these certificates has always been a weak point in client implementations and has enabled researchers to implement software that can be used on WiFi access points to intercept, modify, replay and save traffic presumably secured by SSL/TLS[4]. Recently, several bugs in the certificate validation logic of popular SSL/TLS implementations (GnuTLS and Apple) have been identified and exploited by such interception software. Given the importance of the SSL/TLS certificates, some smartphone users might want to always use trusted interfaces for the initial TLS/SSL handshake and the certificate exchange.

The handshake is not the only vulnerable part in the SSL/TLS protocol from a cryptographic viewpoint. TLS/SSL supports different combinations of encryption and authentication algorithms. Some are known to be weaker than others, e.g. due to the length of the encryption keys that are used. RC4 is among the most widely used encryption algorithms on web servers. A recent survey[5] reveals that among 200,000 analyzed web servers supporting SSL/TLS, 89% of them support support RC4 and 36% of them would negotiate RC4 with recent browsers. This is despite serious concerns from the cryptography community that considers RC4 to be broken [29]. In particular, [29] showed that the first hundred bytes of the encrypted stream are more vulnerable to cryptanalysis than the remaining bytes. If a client still needs to interact with an SSL/TLS server that uses RC4, it could force the utilization of trusted interfaces for the SSL/TLS handshake and the vulnerable bytes in the beginning of the stream.

2.3.2.4 Performance evaluation

We experimentally evaluated the performance impact of forcing Multipath TCP to send the HTTP headers and responses only over a trusted interface. We use a simple network composed of one client, one router and one server. The client is a standard Linux server because it is easier to automate measurements on such a

[4]See e.g. `http://mitmproxy.org`.
[5]See `https://www.trustworthyinternet.org/ssl-pulse/`

server than on a smartphone, but our implementation also works on Android smartphones. The router is also a Linux PC that uses the `tc` software to emulate bandwidth and delays. For our evaluation, the bandwidth is set to 5, 10, 20 and 50 Mbps on the untrusted interface, emulating a WiFi access point, and 1, 5, 10, 20 Mbps on the trusted interface, representing a cellular interface. We set delays of 10, 100 and 200 msec on the trusted and untrusted interface. Our measurement scripts then iterate over all possible combinations of bandwidth and delays for the cellular and WiFi paths. Iterating over multiple conditions allows us to have a better view of the performance of Multipath TCP. In some combinations, WiFi is faster than cellular, in others this is the opposite. This reflects real-world deployments where both situations could happen.

We consider HTTP requests for objects of 1, 10, 100, 200, 500 and 1000 Kbytes. Each HTTP measurement is repeated ten times and for each bandwidth/delay combination we measure the average HTTP page load time. The client and the server use `sendmsg` to always send the HTTP headers over the trusted, "cellular" interface.7 Figure 2.17 compares the download times between Multipath TCP with a trusted interface and regular Multipath TCP. We divide the page-load time of the former by the page-load time of the latter. It is apparent in the graph that indeed, within our large range of environments, the difference between the both is negligible as the boxplots are centered around 1. There is no significant performance decrease due to the utilization of trusted interfaces. With larger filesizes the spread of the page-load times is a bit larger and thus a minor variance can be seen. Increasing the number of repetitions would reduce this variance.

2.3.3 Use-case challenges

Multipath TCP [13] was designed and implemented to provide the same API as regular TCP to applications. This objective was important to ensure the deployability of the protocol by given immediate benefits to existing applications. However, there are many applications and use cases, such as this one, where the traditional socket API is not sufficient anymore to fulfill the needs of the applications. To fully support this use case in the long term, Multipath TCP needs a more advanced API that exposes more information, but not too much information, about the underlying multipath capabilities. This work and the path manager described in Deliverable 1.3 (Multipath TCP path manager) are a first step in this direction. The next objective will be to combine them together and to propose an advanced API for Multipath TCP within the IETF.

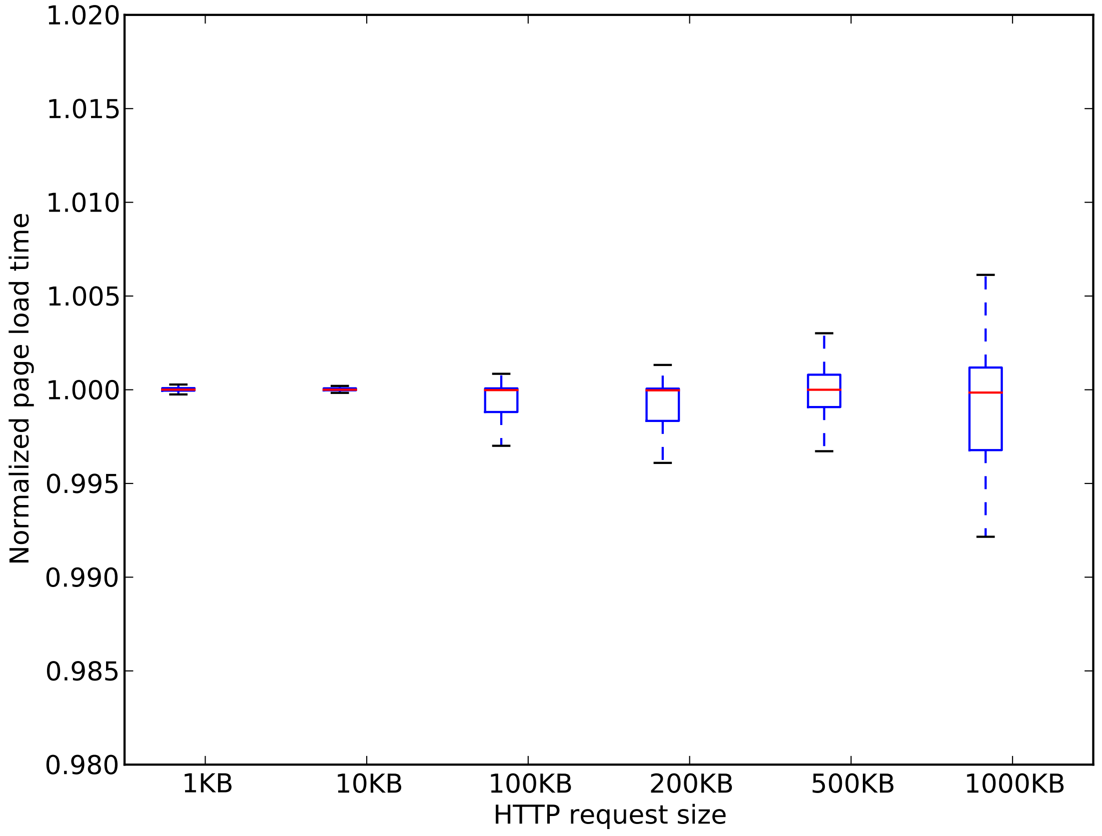


Figure 2.17: Boxplots show the ratio of page-load time between trusted and regular Multipath TCP; there is no significant difference between both.

3 Liquidity in Operator Infrastructure

3.1 ClickOS migration

3.1.1 Use-case Design

In recent years, cloud and data centers deployments have become pervasive, largely as a result of the availability of inexpensive commodity servers, higher capacity network links and ever improving virtualization technologies. Services such as Amazon EC2 [3], Rackspace [32], and Microsoft Azure [27] are now commonplace, and even network operators are looking to get into the game by deploying so call micro-data centers at Points-of-Presence sites [14] in order to run applications such as CDN caches or virtualized middlebox processing [11] close to end users.

Traditionally, the number of VMs running on each server has been low, in the order of ten or so, mostly because of the memory and CPU requirements imposed on the system by the full-fledged OSes that the VMs run. Further, while it is commonplace to speak of clouds being elastic because such VMs can be easily instantiated, suspended and migrated, their size and complexity means that performing such operations can take in the order of seconds or worse (e.g., when migrating a VM hundreds of megabytes or bigger in size).

More recently, work towards VMs based on minimalistic or specialized OSes (e.g., OSv [31], ClickOS [26], Mirage [24], Erlang on Xen [9], HalVM [18], etc.) has started pushing the envelope of how reactive or fluid the cloud can be. These VMs' small CPU and memory footprints (as little as a few megabytes) enable a number of scenarios that are not possible with traditional VMs.

First, such VMs have the potential be instantiated and suspended in tens of milliseconds. This means that they can be deployed on-the-fly, even as new flows arrive in a network, and can be used to effectively cope with flash crowds. Second, the ability to quickly migrate the VM and its state would allow operators to run their servers at "hotter" load levels without fear of overload, since processing could be near instantaneously moved to a less loaded server. Finally, these VMs' small memory footprints could potentially allow thousands or even more such VMs to run on a single, inexpensive server; this would lead to important investment and operating savings, and would allow for fine granularity, virtualized network processing (e.g., per-customer firewalls or CPEs, to name a couple).

Realizing such fluidity, however, poses a number of important challenges, since the virtualization technologies that these VMs run on (e.g., Xen or KVM) were never designed to run this large number of concurrent VMs. In the case of Xen [5], attempts to tackle *some* of the issues such as limited number of event channels or memory grants are under way, but these are still in their infancy and are not necessarily aiming to run the huge number of VMs we are envisioning.

The purpose of this use case is to demonstrate how to concurrently execute thousands of MiniOS-based guests [1] on a single inexpensive server. We will also show instantiation and migration of such VMs in

[1]MiniOS is a minimalistic, paravirtualized operating system distributed with the Xen sources.

tens of milliseconds, and transparent, wide area migration of virtualized middleboxes by combining such VMs with the multi-path TCP (MPTCP) protocol. In this way, this use case and its underlying technologies provide processing liquidity to the operator infrastructure by allowing VM-based network processing to be instantiated whenever and wherever, and transparently migrated.

3.1.2 Use-case Implementation

We base our system on Xen, a type 1 hypervisor (see figure 3.1). In order to keep the hypervisor efficient and secure, device drivers and the system's control plane are offloaded to a special virtual machine commonly running a standard Linux distribution and called the *control domain* or dom0 for short.

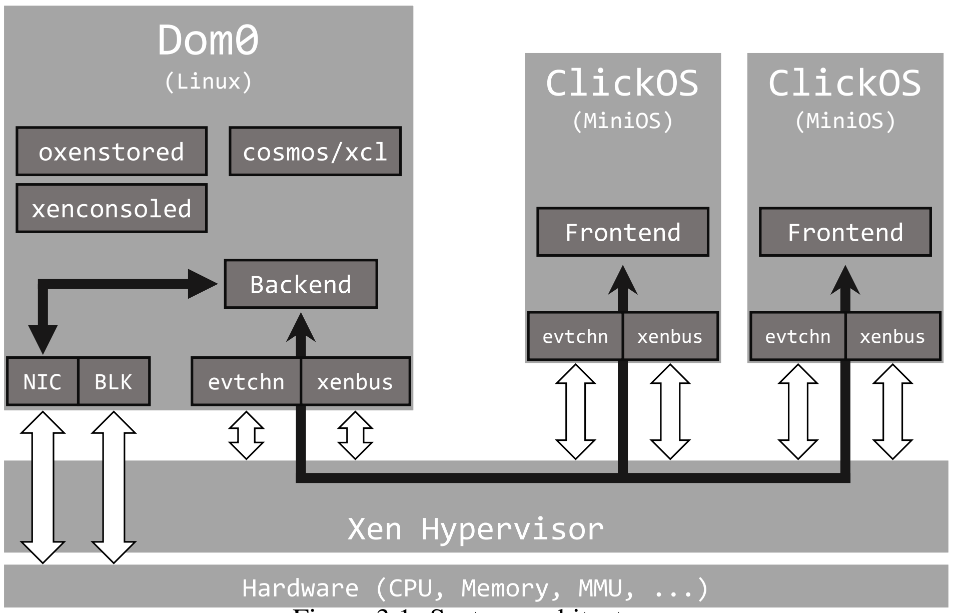


Figure 3.1: System architecture.

The control plane is based around the XenStore, a proc-like filesystem that keeps system and per-VM information such as virtual CPUs, virtual interfaces and memory assignments. Users control Xen (e.g., to create VMs) by means of its *toolstack*: the `xl` command line tool, which in turn uses the `libxl` library to interact with Xen components. In terms of network I/O, Xen uses a split-driver model: a virtualized *netfront* driver resides in the VMs, and a complimentary *netback* driver sits in dom0. dom0 further includes a back-end software switch to direct traffic from network drivers to the netback driver. Finally, event channels are essentially virtual interrupts used to signal the availability of packets.

For the actual virtual machine (also called a guest), we rely on MiniOS. More specifically, we use ClickOS virtual machines, which are basically guests running the Click modular router software on top of MiniOS (please see [26] for more details).

Our contribution: Our work uses these guests to analyze bottlenecks when attempting to run large numbers of VMs, and implements and configures a number of improvements to the Xen ecosystem. These include (1) a lean version of the `libxl` library called `libxcl` which reduces the number of write operations to the XenStore from 37 to 17 per VM and contains less generic code (e.g., it removes code that handles full virtualization guests); (2) a new command line tool called `cosmos` to make use of `libxcl`; (3) a simplified

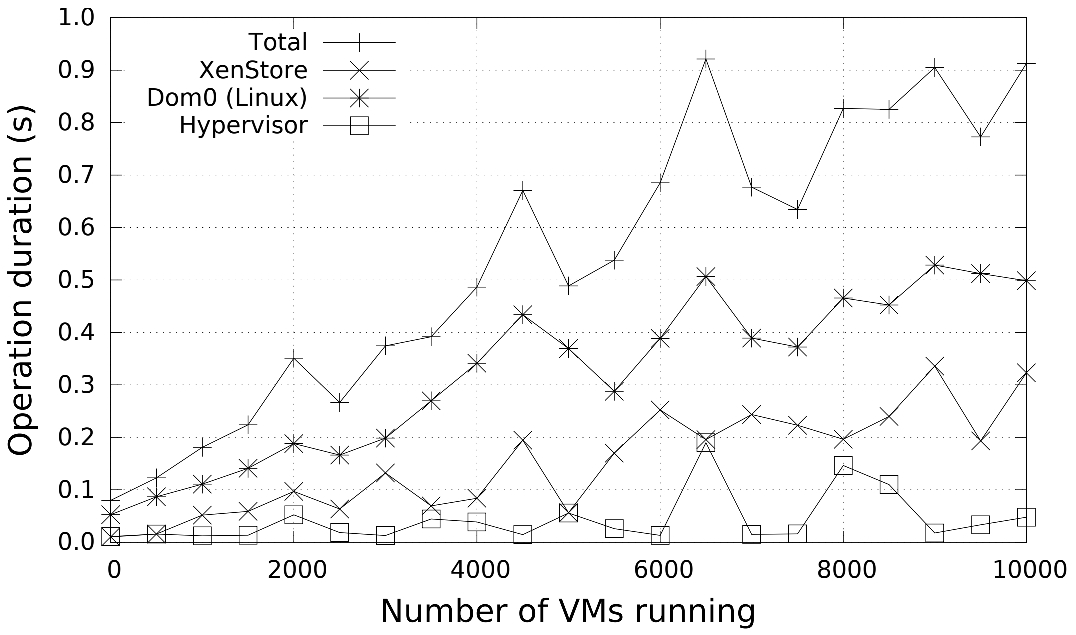


Figure 3.2: Individual boot times for 10,000 ClickOS virtual machines on a single x86 server.

virtual interface hotplug script for attaching interfaces to bridges; (4) placement of all relevant files (hotplug scripts, VM config files, VM images) on a RAM disk; (5) pinning of all vCPUs and reservation of one for the XenStore and another one for the XenConsole process; (6) addressing guests by ID rather than name (the latter causes toolstack code to sequentially go through all existing VM names); (7) use of multiple back-end switches to overcome a maximum number of interfaces per switch limitation; and (8) increasing the maximum number of open files and virtual terminals in Linux in order to support the VMs' consoles. Finally, we implement support for suspending and resuming MiniOS guests which enables migration scenarios.

Our early results are quite encouraging. Figure 3.2 plots the individual boot times of up to 10,000 ClickOS guests on a system with 4 AMD Opteron 6376@2.3GHz CPUs and 64GB of memory (costing about $3,500). As shown, the initial VMs take only tens of milliseconds to start, while the last ones still take under one second. The cumulative time to start all 10,000 VMs was about two hours, and we test the VMs by having them respond to a ping.

3.1.3 Use-case challenges

While the solution presented here has the potential to give operators unprecedented levels of processing fluidity, this is still early research. Going forward, we are now busy replacing the standard Xen store with a purpose-built, lightweight one that will allow us even faster boot times. Second, we are also looking into optimizing the toolstack in order to cut down on migration times. Finally, we are just now starting to look at potential scheduling issues when having large numbers of virtual machines processing traffic concurrently.

3.2 Control Plane for Liquid MPLS VPNs

3.2.1 Use-case design

Network Function Virtualisation is beginning to be deployed in the network based on use cases with high commercial interest. One of these use cases is the vCPE. The main value proposition is to centralise the Layer 3 functionality implemented in the Customer Premises Equipment (CPE) as Virtual Network Functions (VNFs) in a virtualised infrastructure and have simplified CPEs deployed, which basically implement the Layer-2 functionality at the client side.

Inside the virtualised infrastructure, isolation and separation of clients and tenants is a must. One of the control technologies to guarantee client separation in communication links is Multi-Protocol Label Switching (MPLS). In this use case, we explore how to implement the MPLS control plane functionality as a VNF itself, which can be mixed and matched to build different kinds of nodes within the vCPE architecture.

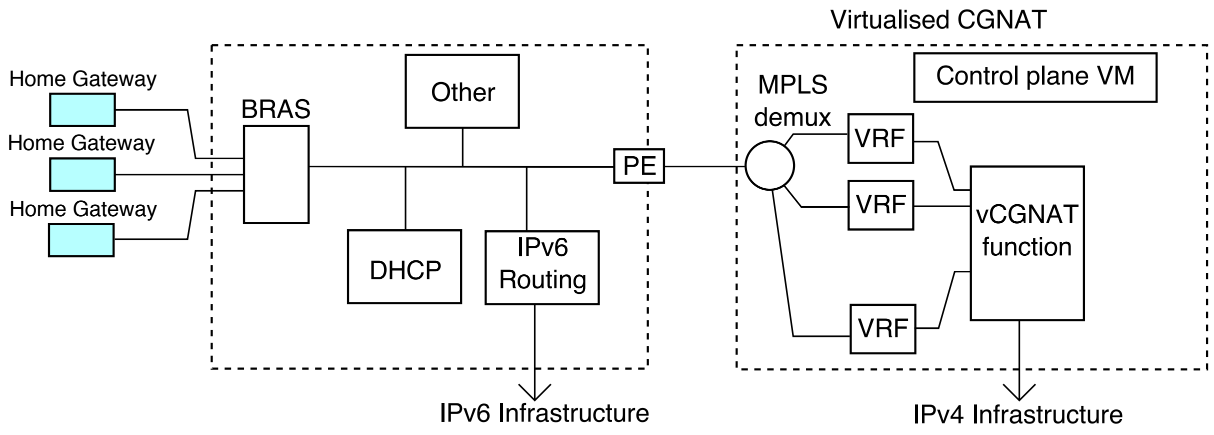


Figure 3.3: A simplified vCPE architecture

Figure 3.3 shows a simplified vCPE architecture. In our use case, we worked on the Virtualised Carried-Grade NAT (CGNAT) and concentrated on the control plane functionality. Our objective was to provide a virtualised MPLS control plane function, which we use to provide last resort of routing within the client's network. This control plane is implemented in a Virtual Machine (VM) that can be migrated between virtualised network nodes if needed.

3.2.2 Use-case implementation

3.2.2.1 The Quagga MPLS-VPN implementation on a Linux Virtual Machine

The objective was to provide a ready to use VM image, which could be used as a control plane for a software-based Network Address Translation (NAT). This image has to comply with stringent memory and disk size requirements in order to be usable in small to medium-size servers, as detailed in Table 3.1.

Extra packages developed in the scope of the Trilogy 2 project, that are required for the installation are:

- Patched Quagga routing daemon, with all patches from `https://github.com/paaguti/quagga-v4vpn`.

Component	Spec
Linux	Ubuntu server 12.04 LTS
VM Processor family	i686 (32-bit Intel)
VM RAM	512 MBytes
Image size	2 GBytes
Network interfaces	1 service + 1 out-of-band management

Table 3.1: Requirements of the Linux VM for the Broadband Access Server (BRAS) virtualised control plane

- Patched *scapy* package, with MPLS VPNv4 patches [4]

The patched Quagga routing daemon is described in earlier deliverables and includes fixes to the Route-Distinguisher (RD) handler, which allow it to inter-work with commercial MPLS Layer 3 VPN implementations.

The patched scapy package provides an easy way to dissect the BGP-4 session, detect new MPLS tunnels being brought up and implement automated detection of new clients using the *scapy* Python library.

3.2.3 Use-case challenges

At this point, the main challenge for this use case scenario is to coordinate the configuration of the different elements in the MPLS routing plane. On the one hand, the Control Plane VM terminates the MPLS control plane stack. To this avail, it has to be able to tap on the MPLS interface and divert Label Distribution Protocol (LDP) and Border Gateway Protocol (BGP-4) traffic.

The other challenge stems from the current MPLS model. Since it does not foresee bidirectional Label Switched Paths (LSPs), and in order to reduce the operations, we explore how to automate the routing configuration of the NAT function with a script that detects new client Virtual Private Networks (VPNs) from the BGP-4 routing information coming from the network.

3.3 Providing resilience to IMS virtualised network functions

3.3.1 Use-case design

Network function virtualisation allows a network operator to reduce costs in CapEx and also in OpEx, and allows the creation of new business models where this network operator could rent, on demand, virtual machines to run virtualised network functions. In such scenario, an infrastructure operator could provide different services to network operators like dynamic and elastic scaling of services, interface for the configuration of a full chain of virtual network functions, automatic placement of virtual network functions, reliability, etc. However, these services should be provided transparently to the users of the network providers, i.e. to the end-users, to provide a real network liquidity. Nowadays, several network elements store the state of traffic flows (TCP and UDP), which obstruct their virtualisation.

In this respect, in this use-case we propose a solution to handle the open issues to virtualise IMS functional elements, included in one of the use cases covered by the ETSI NFV working group. In particular, our main focus is to add resilience by proposing signalling procedures to transparently transfer users among these functional elements after a failure. In addition, our proposal plays an important role to tackle network liquidity problems, when given virtualised functional elements cannot be scaled-up (over-loaded) or deactivated (under-loaded): in those cases, users state should be transferred among functional entities, similar to the failure event. With our approach, a network operator can better adapt its active resources to the instantaneous load, with evident advantages in terms of efficiency and economic costs.

3.3.2 Use-case implementation

Our proposal is focused on providing a reliable virtualised IMS deployment, which will be transparent for end-user applications. It is mainly based on two different signalling procedures: (1) the signalling to add or remove virtualised network functions, and (2) the IMS signalling to transfer the load caused by users from an IMS functional element to another. While the former procedure can be considered generic, and can be adapted to other scenarios, the latter is specific for virtualised IMS deployments. Both procedures will be explained in detail in next subsections.

3.3.2.1 Signalling to add/remove virtualised network functions

This section is devoted to provide a solution to control the instantiation of new virtualised network functions, as well as the mechanisms to stop active VFNs. Figure 3.4 presents a flow diagram showing the signalling exchange to create new instances of virtualised functional elements (Figure 3.4.a) and to remove an IMS functional element instance (Figure 3.4.b). The first three steps are similar in both cases, where the corresponding EMS element triggers the event. This trigger is forwarded to the OSS/BSS, which generates the proper command depending on the received trigger.

Adding a new virtualised network function. As shown in Figure 3.4.a), in step (4), the OSS/BSS generates a command towards the Orchestrator in order to request the instantiation of a new VNF with similar functionalities as the failed or overloaded machine. In (5), the Orchestrator translates this request towards the VIM, including an identifier of the failed or overloaded VNF, in order to select an image in the virtual machine pool. With this information, the VIM instantiates a new VNF. After the new VNF is active, the VIM sends an OK message towards the Orchestrator in step (6), which, in turn, informs the OSS/BSS about the availability of this new VNF in step (7). In the particular case when the affected VNF is an IMS functional element, the OSS/BSS contacts the Control Function Selection (CFS), which provides the functionality to transfer users between those functional elements. This signalling is generated in step (8), which includes, at least, three parameters: the affected VNF, the target load for such VNF , and the new VNF that can be used to transfer users to. With this information, the CFS transfers users between those VNFs, using the procedures that will be explained in next section. After the appropriate users are transferred, the CFS answers back to the OSS/BSS in step (9) with an OK message.

Removing a virtualised network function. Figure 3.4.b) shows the steps to remove an instance of a given VNF after all its users are transferred to other VNF. As explained before, steps (1), (2) and (3) are

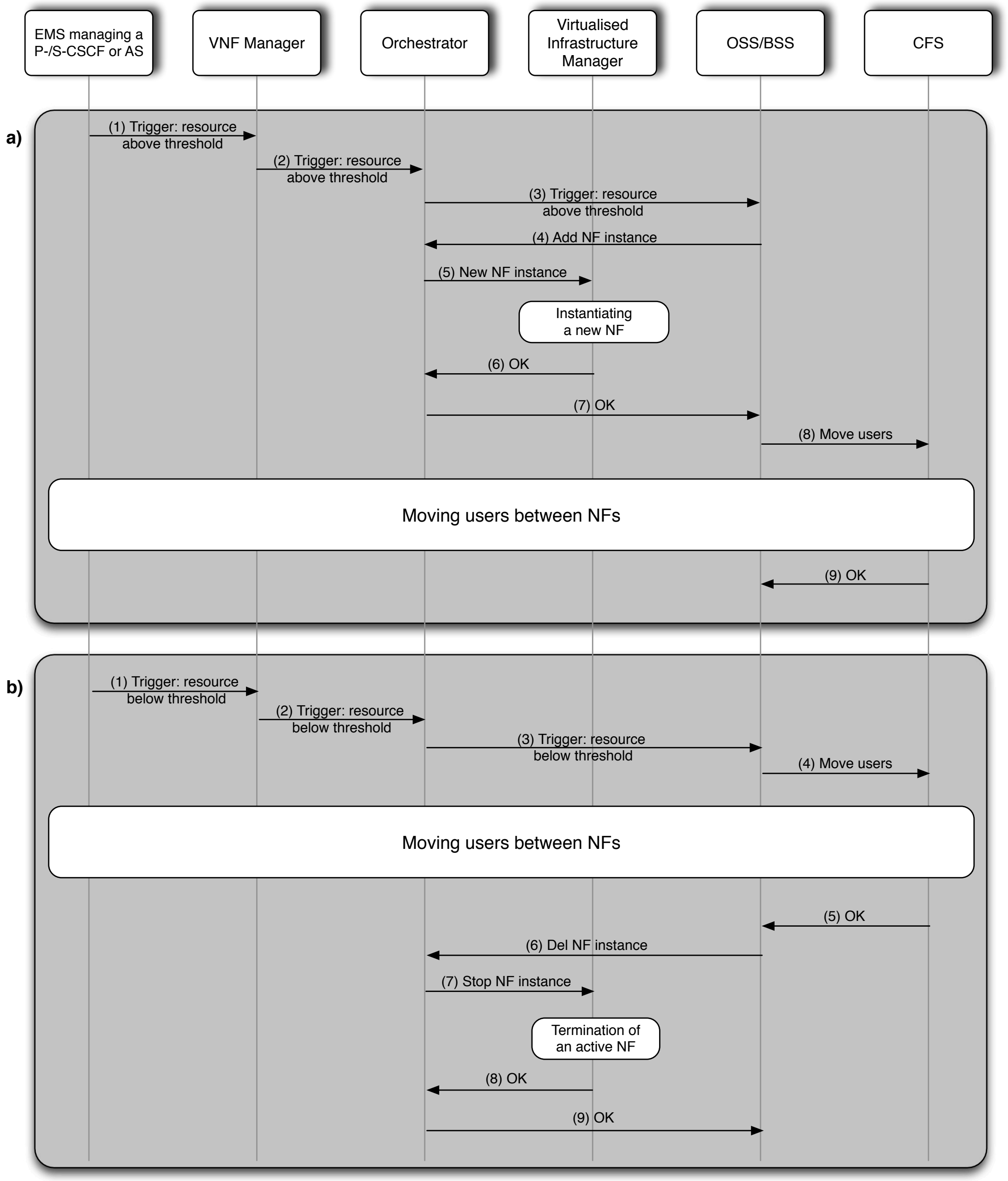

EMS managing a P-/S-CSCF or AS
VNF Manager
Orchestrator
Virtualised Infrastructure Manager
OSS/BSS
CFS
a)
(1) Trigger: resource above threshold
(2) Trigger: resource above threshold
(3) Trigger: resource above threshold
(4) Add NF instance
(5) New NF instance
Instantiating a new NF
(6) OK
(7) OK
(8) Move users
Moving users between NFs
(9) OK
b)
(1) Trigger: resource below threshold
(2) Trigger: resource below threshold
(3) Trigger: resource below threshold
(4) Move users
Moving users between NFs
(5) OK
(6) Del NF instance
(7) Stop NF instance
Termination of an active NF
(8) OK
(9) OK

Figure 3.4: Signalling to start and stop network function elements.

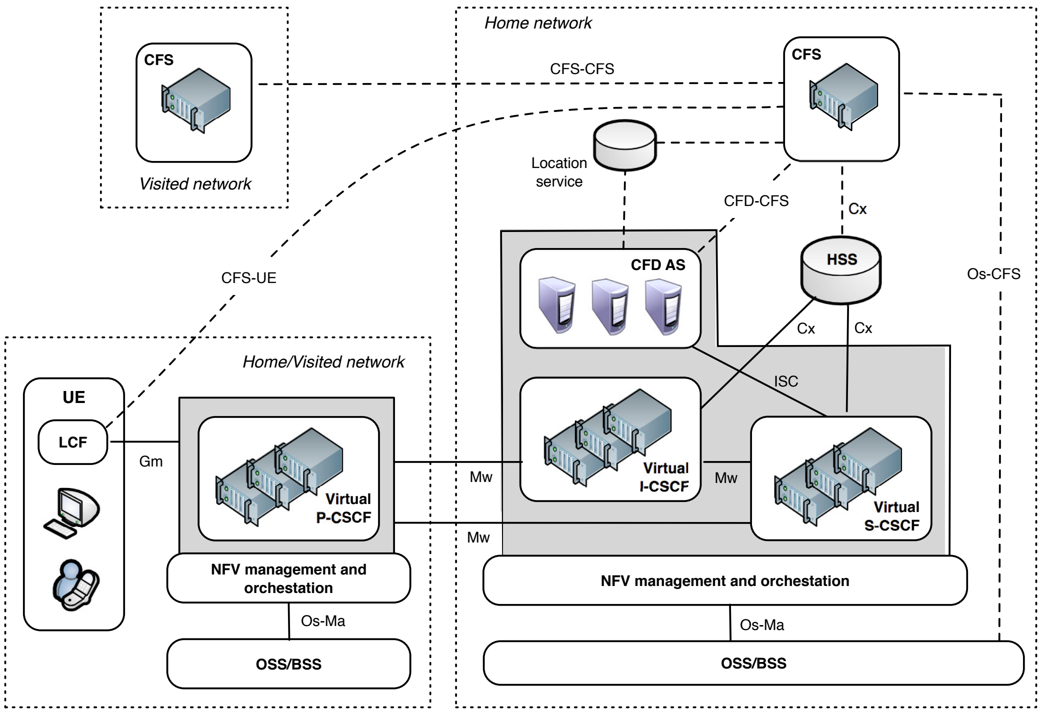


Figure 3.5: Overview of the proposed architecture.

similar to the ones presented in the previous procedure. Because the underutilised VNF is an IMS functional element and another similar element is already running, in step (4) the OSS/BSS contacts the CFS to transfer all users from the former machine to the latter. After all users are transferred, in step (5) the CFS answers back the OSS/BSS with an OK message, which in (6) orders the Orchestrator to stop a VNF. In step (7) the Orchestrator orders the VIM to stop such VNF. Steps (8) and (9) show the OK messages to confirm that the VNF is stopped.

3.3.2.2 IMS signalling procedures

This section describes the functional entities, as well as the IMS signalling procedures, that are proposed in this deliverable to support an appropriate allocation of users to the available CSCFs in a virtualised IMS deployment. These entities and signalling procedures allow a home operator to dynamically change the allocation of the P-CSCF or S-CSCF of any of its users, transparently to the end-user applications running at the IMS terminal. This way, our architecture enables the home operator to maintain an appropriate distribution of the load among the virtualised CSCFs, which are instantiated and removed by the NFV management and orchestration system. Figure 3.5 shows the new functional elements proposed in this section, and their relation with the IMS core and the NFV architecture.

The Control Function Discovery (CFD) is an application server acting as a SIP user agent. The SIP URI of this AS is configured in the user profile, in initial filter criteria that will be matched during the registration. This way, after the successful registration of the user in the IMS, the CFD application server is notified by the S-CSCF with information corresponding to this particular registration, which will include, although it is not limited to, the addresses of the P-CSCF and the S-CSCF allocated to the user. The CFD AS stores this information in a location service, which is a database in the home network domain that maintains all the information related with the IMS control functions serving the users of the home operator. From the performance point of view, the CFD AS can also be virtualised by the NFV management and orchestration system, which can dynamically create and delete instances of the CFD AS to adapt the processing capacity under varying load of registrations.

The Control Function Selection (CFS) is our contact point with the NFV architecture. The OSS/BSS triggers the CFS when a redistribution of the load among the existing virtualised P-CSCFs, S-CSCFs, I-CSCFs or CFD ASs is required. This may happen as a consequence of a failing or overloaded virtualised network function, or due to the instantiation or removal of a CSCF or CFD AS. To support an appropriate operation, the CFS implements a Cx interface, as defined for the IMS in [1], that enables the communication with the HSS.

The CFS mainly bases its operation on the information stored by the CFD AS in the location service of the home network domain. When the CFS receives an indication from the OSS/BSS, via the reference point os-cfs, to perform a load transfer from a virtualised P-CSCF (or S-CSCF) to another, it retrieves the information about the users who are served by the former CSCF from the location service. Using this information, and according to the policies established by the home operator, the CFS makes a decision about the set of users that will be transferred to the target CSCF. Then, the CFS triggers the IMS signalling procedures that allow changing the allocation of the CSCF for each of these users, by contacting a Local Control Function (LCF) in the IMS terminals through the reference point cfs-ue. The LCF is integrated in the IMS stack of the terminal, and executes these signalling procedures transparently to any end-user applications running at the terminal, which are kept unaware of the CSCF change. Additionally, the CFS is contacted by the OSS/BSS when a virtualised network function executing an I-CSCF or a CFD AS is instantiated or deleted. This enables the CFS to ensure an appropriate load balancing among the existing virtualised I-CSCFs and CFD ASs, by properly configuring the DNS server of the home network.

Last but not least, according to the IMS specifications [2], there is the possibility that the P-CSCF allocated to a roaming user is located in a visited network. In that specific case, after the successful registration of the user, the CFD AS stores the address of the P-CSCF in the location service. Additionally, if this P-CSCF was not allocated to any other user of the home network, the CFD AS contacts the CFS via the reference point cfd-cfs. If the visited network implements the solution presented in this solution, and there is a service layer agreement between the home and the visited operators, the CFS in the home network communicates with the CFS in the visited network, through the reference point cfs-cfs, and registers its desire to receive notifications about the state of the virtualised network function holding the P-CSCF allocated to the user. This way, the CFS in the home network will be notified in case that the virtualised P-CSCF fails or needs to be removed, due to a load redistribution procedure in the visited network. If this occurs, CFS in the home network can trigger a change of P-CSCF for each affected IMS terminal.

In the following, we illustrate the IMS signalling mechanisms proposed in this solution to change the allocation of the P-CSCF or the S-CSCF for a given user of the home operator. Additionally, we describe the procedures required to support the virtualisation of the IMS contact point within the home network, i.e., the I-CSCF, as well as the virtualisation of the CFD ASs.

Change of the P-CSCF allocation. This section describes the procedures that are performed in our solution to change the P-CSCF allocated to a given user. Prior to the execution of these procedures, we assume that the user has already registered to IMS, and has established a set of multimedia sessions by means of SIP dialogs.

In our solution, the OSS/BSS in the home network contacts the CFS to perform a load transfer from a virtualised P-CSCF to another existing P-CSCFs (step 1). This may happen, for instance, due to the overload of a virtualised P-CSCF, or due to a failure of a P-CSCF, which requires its load to be transferred to a new P-CSCF instantiated by the NFV management and orchestration system. It may also occur because the load supported by the existing P-CSCFs is at a low level, and a P-CSCF needs to be removed and its load redistributed. In any case, the OSS/BSS explicitly indicates the reason of the trigger, and the assignment of load that should be transferred to each existing P-CSCF.

As a consequence of the indication received from the OSS/BSS, the CFS retrieves from the location service the information about the users that have been allocated the involved P-CSCF. Based on this information, and taking into account both the information provided by the OSS/BSS in the request and the policies established by the home operator, the CFS determines the set of users who will be transferred to each (already existing) P-CSCF.

In addition, the CFS may also be contacted by a CFS of another operator, in case that a virtualised P-CSCF in a visited network has failed or is to be removed, and the CFS in the home network had expressed its desire to receive notifications about the status of that specific virtualised network function. In this case, the CFS of the visited network will indicate the reason of the trigger, and may provide an alternative P-CSCF to be allocated to the affected users of the home operator. In this case, the CFS in the home network will determine the set of users who are assigned to affected P-CSCF, by means of the location service. These users will be transferred to the P-CSCF included in the trigger, if any was indicated.

Next, the CFS contacts the IMS terminal of each user to be transferred to a new P-CSCF (step 2). This communication is established with the LCF of each terminal via the reference point cfs-ue . As a result of this communication, each LCF can independent initiate the IMS signalling procedures that are necessary to enforce the allocation of the new P-CSCF to the user. In particular, the LCF executes a new IMS-level registration, using the address of the new P-CSCF (step 3). After the successful registration of the user, and following the regular IMS procedures, the S-CSCF allocated to the user performs service control functionalities, by sending a SIP REGISTER request to any AS specified in the user profile for the registration event. This way, a REGISTER request is sent to a CFD AS, which updates the information contained in the location service to reflect the address of the new P-CSCF allocated to the user. Finally, to complete the transfer of the user to the new P-CSCF, it is necessary to replace all the SIP dialogs established by the user via the old P-CSCF, with new SIP dialogs through the new P-CSCF. This process is done according to the procedures defined in [25].

Change of the S-CSCF allocation. This section focuses on the procedures proposed in this solution to change the S-CSCF allocated to a given user. Analogously to the previous case, we assume that the user is registered in the IMS, and has established a set of multimedia sessions using the SIP protocol.

The procedures are similar to those described in the previous section, with a main difference. As the S-CSCF is a functional entity that is always located in the home network of the user, the indication to perform the redistribution of the load corresponding to a S-CSCF can only be triggered by the OSS/BSS (step 1), and cannot be originated from a visited network. After receiving this trigger, the CFS determines the set of users that will be transferred to each existing S-CSCF, using the information contained in the request from the OSS/BSS, the information stored in the location service and the policies established by the home operator. Then, for each of these users, the CFS contacts the HSS through the reference point Cx, and changes the S-CSCF assigned to the user (step 2). This can be done using a Multimedia-Authentication-Request (MAR) Diameter command, according to [1]. Next, the CFS communicates with the IMS terminal of each of the users, and instructs the LCF to start the IMS procedures needed to enforce the allocation of the new S-CSCF to the user (step 3).

Similarly to the previous case, the LCF first performs an IMS-level registration using the new S-CSCF. In (step 4), the registration proceeds in two phases. In a first phase, the IMS terminal generates a SIP REGISTER request, which is routed via the P-CSCF allocated to the user and an I-CSCF belonging to the home network. Then, the I-CSCF contacts the HSS to discover if there is a S-CSCF allocated to the user. As the user has already been allocated the new S-CSCF in step 2, the HSS returns the address of the new S-CSCF to the I-CSCF; the new S-CSCF, consequently receives the REGISTER request and performs the authentication of the user, which requires a new REGISTER transaction between the LCF and the S-CSCF.

After the successful registration of the user, the S-CSCF performs a set of service control functionalities, which result in the update of the information stored in the location service by a CFD application server, to reflect the address of the new S-CSCF assigned to the user. Finally, the SIP dialogs established by the user are replaced via the new S-CSCF (step 5).

Adding or removing I-CSCF elements and CFD ASs. Finally, the solution presented in this work supports the virtualisation of I-CSCF elements in the IMS deployment. As it was previously commented, I-

CSCFs play the role of entry points in the home operator network. The selection of an I-CSCF during the IMS registration and session setup follows the SIP procedures specified in [35], and is based on DNS usage. Therefore, when a virtualised I-CSCF is created or removed by the NFV management and orchestration system, the OSS/BSS triggers the CFS, which in turn changes the configuration of the DNS in the home network to reflect the addition or deletion of the I-CSCF address. This will enable an appropriate (for low-medium frequency updates such as the ones of interest for many application domains [6]) load balancing among the available I-CSCFs.

In addition, as it was mentioned, the CFD AS can also be virtualised to adapt the processing capacity of our architecture to the registration load. This AS will be identified by a SIP URI, containing a domain name that, according to the procedures specified in [35], can be resolved by the S-CSCF in the DNS to the different addresses of the virtual machines executing the CFD service. Analogously to the case of the I-CSCF, the OSS/BSS will trigger the CFS when a virtualised CFD application server is created or removed, so that it can change the configuration of the DNS in the home network accordingly, this way enabling an appropriate load balancing among the existing ASs.

Validation of the proposed mechanisms. In order to check the correctness of our proposal, we have deployed a test-bed using the FOKUS OpenIMS core[2] and the SIPp[3] open software . The former includes the call/session control functions of the IMS (i.e., P-CSCF, S-CSCF and I-CSCF) and a home subscriber server (HSS), while the latter can be used as an IMS user equipment. The OpenIMS core is deployed over three virtual machines: a first one including the HSS together with 3 CSCFs, namely P-CSCF1, S-CSCF1 and I-CSCF1; a second one executing P-CSCF2 and S-CSCF2; and a third one including P-CSCF3 and S-CSCF3. Two additional virtual machines are used to execute the SIPp scripts corresponding to the UEs.

In a first experiment, there are 20 registered users, who have been allocated P-CSCF1. These users will establish IMS sessions with 2 registered users that utilise P-CSCF3. The aggregate session setup rate is 1 session/s, which is considered in this experiment to be above the load that a P-CSCF can process. Consequently, using the SIPp scripts, we execute the procedures to change the P-CSCF allocation of 10 users. This change is triggered approximately after 60s. In this particular setup, all the users are transferred in parallel, which imposes a transitory high load for both P-CSCFs. Or future work will target this aspect, comprising the study of algorithms to reduce this load. Figure 3.6.a) presents the result of this experiment, where we can observe that, once the users are transferred to the new P-CSCF, the load is distributed between them.

In a second experiment, we show a scenario where the load of two S-CSCFs are below a minimum threshold. So the users assigned to one of them can be transferred to the other. In this case, there are

[2]http://www.fokus.fraunhofer.de/en/fokus_testbeds/open_ims_playground/components/osims/index.html
[3]http://sipp.sourceforge.net

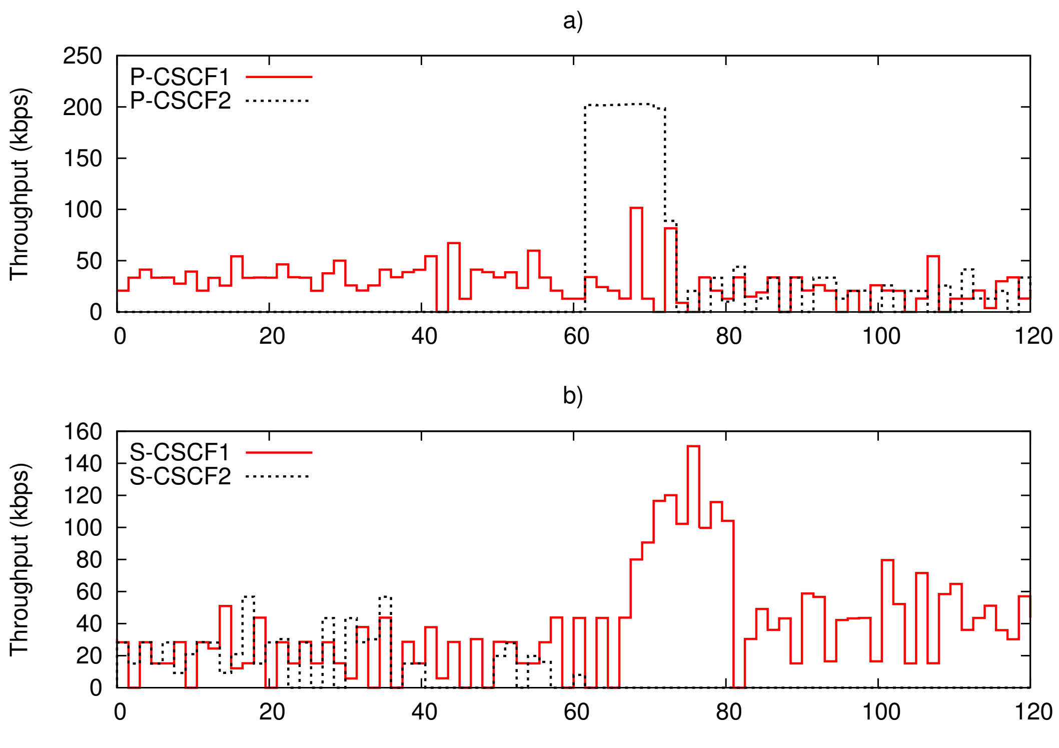


Figure 3.6: Validation of the resilient IMS solution

10 users registered in S-CSCF1, while another 10 are registered in S-CSCF2. The 20 users generate an average aggregate rate of 1 sessions/second, as in the previous experiment. After approximately 60 seconds, the users registered in S-CSCF2 are transferred to S-CSCF1. As it can be seen in Fig. 3.6.b), the users and their active sessions are transferred to the new S-CSCF between 60s and 80s, causing a higher load in S-CSCF1 during that period. After 80s, users have been successfully transferred to S-CSCF1, which receives an average rate of 1 session setup per second.

3.3.3 Use-case challenges

The solution presented here includes new functional elements and interfaces to coordinate the appropriate allocation of users to the available CSCFs in a virtualised IMS deployment. Previous to this proposal, it was impossible to resiliently recover from failures or even to transfer users' sessions from under or overloaded IMS entities. In order to validate our original proposal and implementation mechanisms, we have developed a prototype of them as a proof-of-concept, which allows us to deploy a test-bed for validation, testing, and quantitative experimental evaluation of achievable performance. The encouraging results achieved so far show the viability and correctness of the proposed solution, and are stimulating further research activities in order to extend the current solution prototype to include the possibility to manage other IMS functional elements, like ASs, as virtualised network functions. The signalling procedures to allow transferring sessions between ASs is challenging, but necessary to provide a complete solution for the IMS.

3.4 Minicache video delivery

3.4.1 Use-case Design

Consumer Internet video traffic will make up 69% of all consumer Internet traffic by 2017, up from 57% in 2012 [7]. Within this, real-time entertainment dominates, with Netflix accounting for as much as 33% of peak period downstream traffic in North America [36]. The large majority of this traffic is delivered via Content Delivery Networks (CDNs), that are expected to transport almost two-thirds of all video traffic by 2017 [7]. CDNs optimize video delivery by deploying so-called content caches in different networks, often geographical dispersed, so that user requests can be handled locally by nearby caches.

While Akamai is the largest player others exist, ranging from traditional CDNs to ISP-operated CDNs, content provider CDNs, free CDNs and even peer-to-peer CDNs [40]. Their business models and infrastructures vary, but what they all have in common is that they would benefit from having a presence, even if small, at a large number of distributed points in networks, such as the Points of Presence (PoPs) that operators deploy at aggregation points in their networks. Proof of this is the fact that Akamai has already started forming alliances with a number of large ISPs, including AT&T, Orange, Swisscom and KT [14].

Beyond such two-way agreements, more recently ISPs have started deploying small datacenters called *micro datacenters* at PoPs and to allow third parties access to these infrastructures [14], while others have suggested the creation of *nano datacenters* that would include equipment such as home gateways and set-top-boxes at user premises but controlled by ISPs [22].

The availability of such “pay-on-demand” infrastructure would allow CDNs to dynamically expand their capacity and reach, without having to go through the expensive and time-consuming process of deploying hardware and facilities. Such an arrangement would be especially beneficial to smaller players in the market who cannot afford large up-front investments. For instance, we could envision a virtual “CDN” instantiated just for the delivery of one piece of live content (e.g., a one hour documentary), grown based on the number of actual subscribers for that show, and taken down when it is over. This would reduce or outright eliminate up-front investment costs, minimize business risks and encourage new players to enter the market.

In order for this vision to work, virtualization would be required to enable the safe sharing of common infrastructure resources (in the form of content caches) across different virtual CDN operators. Towards this, we introduce Minicache, a virtual cache node prototype. Minicache is a tiny single-purpose Xen virtual machine built on top of MiniOS, a minimalistic, para-virtualized OS available with the Xen sources. Our contributions consist of (1) a prototype implementation of Minicache that packetizes data retrieved from block devices and sends the resulting packets out to a NIC at rates of 10Gb/s, and (2) a preliminary implementation and evaluation of an HTTP server on top of MiniOS that we will eventually merge with Minicache.

Because it is built on top of a minimalistic OS, Minicache’s virtual machines can be instantiated in 30 milliseconds and have a memory footprint of only 5 MB [4], allowing it to quickly scale out to make use of

[4]Please refer to [26] for details on how these figures are achieved.

additional resources on a server such as CPU cores, SSD drives and NIC ports without need for content replication; for instance, 64-core AMD servers are now inexpensive, in the order of $3,500, and other companies such as Tilera are pushing towards hundreds of cores in a single chip [41].

Such small boot times matter: a study of flash crowds and their effects on CDNs shows that even having boot times in the order of seconds can result in over-subscription, and that services such as Amazon's EC2 can only spin up VMs in one second at best [39]. Being to able to react this quickly to load also means that content caches can be run at higher utilization levels than usual.

In short, the aim with Minicache is to make storage and content distribution more fluid by virtualizing content caches and allowing them to be instantiated and migrated in millisecond timeframes.

3.4.2 Use-case Implementation

Our current Minicache prototype is built on top of MiniOS and encapsulates block data read via the SSD and RAM disks into UDP packets (figure 3.7). Briefly, Minicache's main loop submits arbitrary block I/O read requests from the *blkfront* driver and splits the block read into multiple packet buffers. Each buffer is encapsulated in Ethernet/IP/UDP headers before being sent out to the optimized, Netmap-based `netfront` driver introduced in [26].

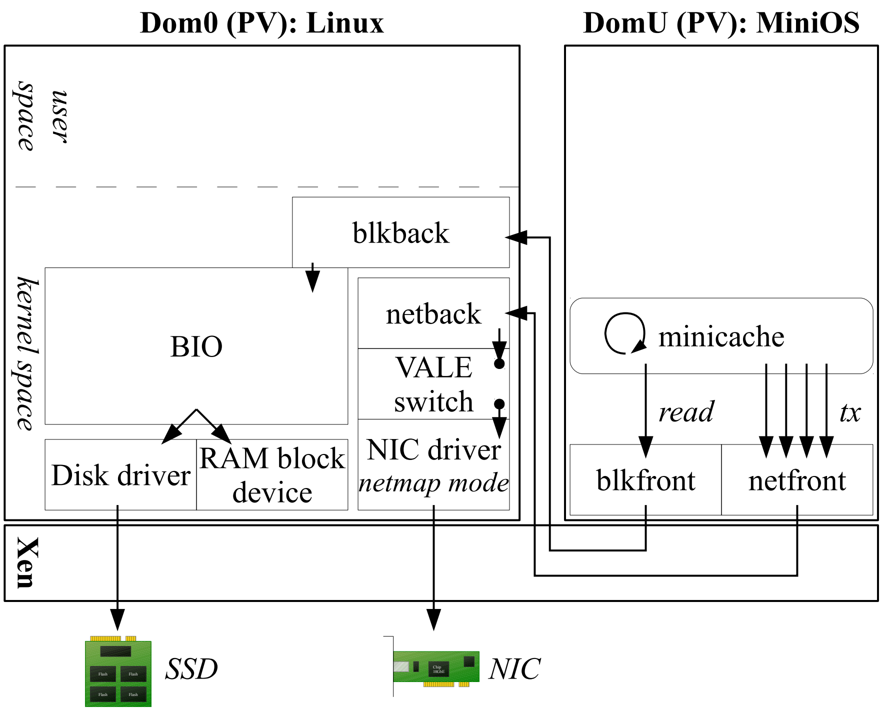


Figure 3.7: Minicache architecture.

In addition, Minicache has a pool memory allocator used for temporarily storing block data as well as packet buffers about to be sent out. Pool memory is allocated before any processing is needed in order to avoid allocation costs, including memory alignment operations, when processing requests. In addition, we use a ring structure to keep track of references to unused memory objects. If objects are *picked* from the pool, the object's reference gets dequeued and returned to the caller; the opposite operation (*put*) enqueues a reference back onto the ring.

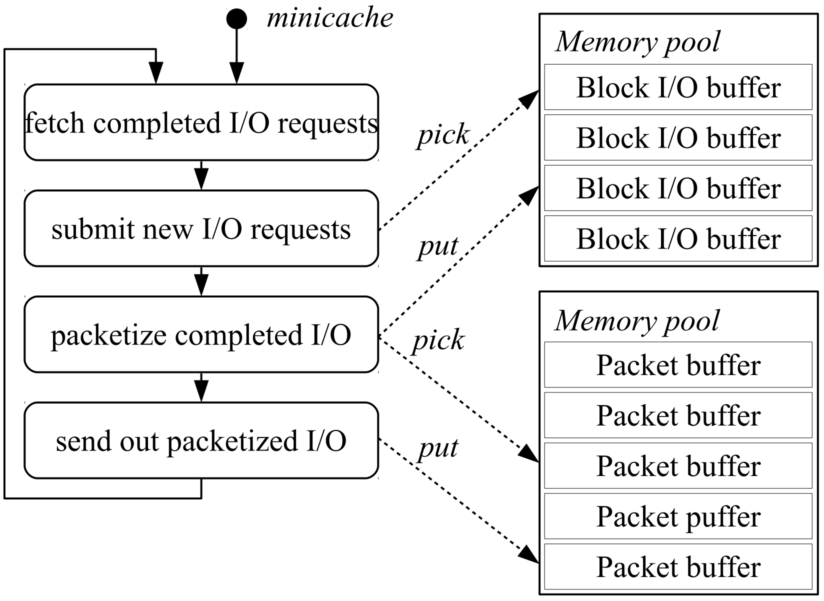


Figure 3.8: Minicache's main loop and memory allocation scheme.

Figure 3.8 illustrates the main steps involved in Minicache's main processing loop:

- First, Minicache polls the `blkfront` driver to see if there are any completed I/O requests. If so, we mark the request as completed (see third step) and carry out a few clean-up tasks.
- After that, we pick new receive buffers from the memory pool and submit new block I/O requests to MiniOS' `blkfront`.
- Once submitted, we begin packetizing the block data received in the first step. More specifically, Minicache splits one data block into multiple 1070-byte packets (1024 bytes for payload and 42 bytes for the Ethernet, IPv4 and UDP headers)[5]. Each packet is created from scratch using a packet buffer picked from the packet buffer memory pool. After the block data copy and packet header writing is done, Minicache puts the block buffer back in its pool.
- Finally, we pass the packet buffers to the optimized `netfront` driver. Once sent, we return the buffers to their memory pool.

We evaluated Minicache's performance by measuring the packet rate generated when reading blocks of different sizes from either the SSD or the RAM drive and packetizing the data. We performed the experiments on an inexpensive (about $1,500), single-socket Intel Xeon E3-1220 system (4 cores at 3.1 GHz) with 16 GB of DDR3-ECC RAM, a Crucial m4 128 GB SATA SSD[8], Linux 3.9.6, and Xen 4.2.0. We connected this server to another one with similar specs running netmap [33] to receive packets and measure rates. Both servers have Intel X520-T2 cards linked via a direct cable. In addition, the Minicache server's `netback`

[5]We chose 1024 bytes as payload because this is a common divisor of the block sizes we use in our system.

driver is connected to a modified, netmap-enabled `ixgbe` driver via the VALE software switch [34] instead of Xen's standard Open vSwitch.

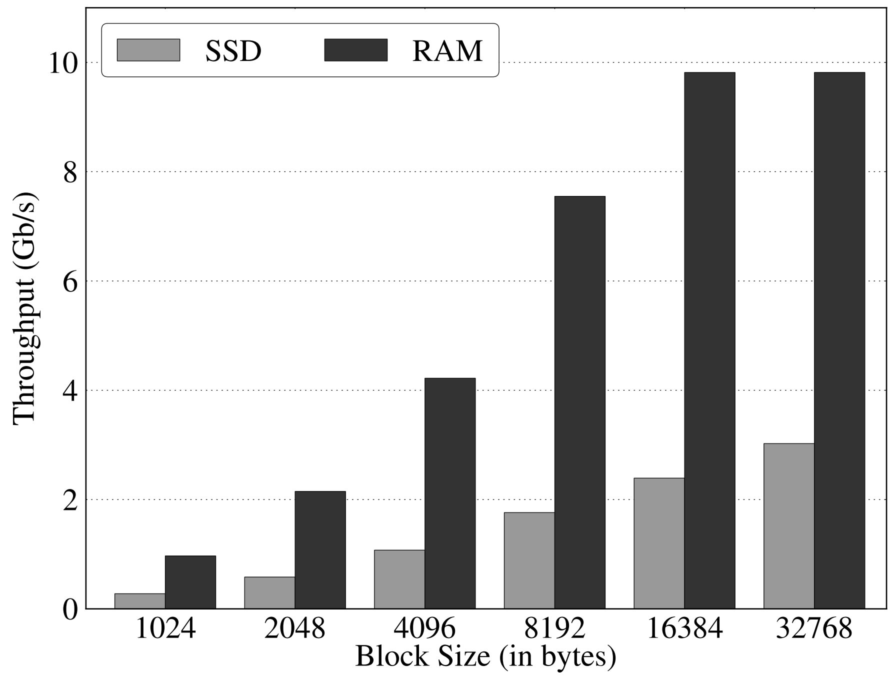


Figure 3.9: Minicache's performance.

The results are shown in figure 3.9. For 16KB blocks and larger and using the RAM drive Minicache is able to fill up an entire 10Gb pipe. For the SSD the rate is expectedly lower, limited by the SSD's maximum rate. For smaller block sizes we can start to see the impact of MiniOS' unoptimized `blkfront` driver, especially in the case of the RAM drive.

HTTP Server: Towards a more realistic content cache, we started developing an HTTP server on top of MiniOS which we will eventually integrate with Minicache. To measure its performance we relied on the `wrk` benchmark tool [16]. For now, the server can handle a rate of 2,000 requests/s on a single CPU core. For comparison, we ran a simple test using a node.js [30] HTTP server on dom0 and obtained rates of 4K requests/s; we are currently in the process of improving our server's performance.

3.4.3 Use-case Challenges

The results are promising, though more work is needed. First, we are improving the throughput performance of Minicache and looking into evaluating the system when several instances of it are running concurrently. We also recently completed a port of Minicache to the ARM architecture (more specifically, a Cubietruck), which would allow for easier deployment all the way to the edge of the network (e.g., next to base stations or even in customer-premises equipment where space and energy are constrained); we will report on this in upcoming deliverables. Finally, NEC's open source website, http://cnp.neclab.eu, is currently being served by Minicache as a live test of the system.

4 Liquidity in the Wide Area

4.1 Hot-fix propagation with Irminsule

4.1.1 Use-case Design

Hotfix propagation is a key use case for Trilogy 2. The ability to quickly propagate fixes to code will be key to providing a robust, secure liquid net. Here we describe how we solve the problems of supporting low-latency deployment of code requiring strong isolation to resource-constrained embedded platforms. The text below focuses on Mirage unikernels, but the technique is applicable more widely.

One reason that unikernels are so efficient is that they compile a single image from application logic, libraries and configuration files. Re-configuring the VM therefore often requires recompiling it, which is much more laborious than just editing a file and restarting a service in Unix. We solved this by taking advantage of the fact that unikernels are small enough (usually a megabyte or so for typical applications) to track using the same techniques that we already use for distributed version control of the constituent source libraries that make up a unikernel application.

4.1.2 Use-case Implementation

A conventional Unix binary requires tracking not only the source code that went into compiling it, but also all the host libraries that may dynamically link against it, as well as the associated configuration files and other data associated with them in a shared filesystem. In Mirage, we rely on the fact that a unikernel image is a standalone Xen VM, and all of its constituent data are represented as (OCaml) code, and typically managed using git. When building a project, Mirage parses its configuration and installs the required OCaml libraries using the OPAM14 source-based package manager[1]. OPAM uses the same answer set programming constraint solver as Debian to select a co-installable set of libraries that satisfy the requested configuration. It then records a source manifest that, when supplied with the same host compiler, is sufficient to rebuild the same unikernel.

With this git-based workflow it actually becomes easier to track versions of deployed services than with full-blown VMs. The source repository for a unikernel on GitHub is configured so that any commit triggers the Travis CI (continuous integration) service to build the repository. If this build succeeds, the Travis script in the repository commits the resulting unikernel binary back to a corresponding deployment repository. We then simply have to checkout the deployment from HEAD in this repository. This process ensures that

(i) every deployed image is trackable back to its constituent libraries via its git commit ref;

(ii) the latest committed version is always the one to be deployed;

(iii) potentially, via a small extension, that any past version can be deployed at will for testing and bisection purposes by prepending the desired deployment commit ref to the service DNS, e.g., $< git-commit-$

[1]ttps://opam.ocaml.org

$ref >$ *.service.name.*

The use of git in this fashion has been deployed for the live openmirage.org website since Oct 2013, and is only practical due to the small size of the resulting unikernel images (around 1–2MB for a typical service).

Modularity

We both exploit and enable heavy use of modularity, a valuable feature when building reliable distributed systems. The first form of modularity is found the way Mirage is implemented – a set of lightweight OCaml libraries fulfilling module type signatures, making it easy to pick and choose the features to be included in a particular unikernel. The result is that developing features such as Conduit was far more straightforward than would have been the case in a traditional OS: even during development, it never crashed the Mini-OS kernel, with almost every error being caught and turned into an explicit condition or a high-level OCaml exception. Similarly, a proxy uses the same OCaml TCP/IP library as found in the unikernels, but with very different runtime policies.

4.1.3 Use-case Challenges

The new Conduit capability also directly addresses one of the key criticisms of the unikernel approach: lack of multilingual support. With this work—specifically the combination of a proxy and Conduit's low latency high throughput inter-VM communication—it is entirely feasible to launch a TCP/IP Mirage unikernel that will proxy incoming traffic to another unikernel (e.g. in Ruby or PHP) that only needs to implement the Conduit protocol and has no need for a full TCP/IP implementation.

Backwards Compatibility

Despite the focus on unikernels, we have maintained backwards compatibility with the existing Xen guest ABI. This contrasts with systems such as ClickOS where the focus is on achieving dense, highly parallel deployments of 10,000s of VMs in an x86 64 datacenter. We anticipate that both of these approaches will converge in up-stream Xen in the future through a revision of the Xen-Store protocol. Nonetheless, we currently do support the deployment and management of conventional VMs (e.g. Linux, FreeBSD) on both ARM and in traditional x86 datacenter environments, albeit without giving any latency guarantees due to the inherent boot overheads of such VMs. Our unikernel tests on x86 point to the intriguing possibility of very fast 20–30ms response times in datacenter environments as well, which we intend to explore in future work.

An area where our work is well-suited to Xen/ARM is its use of explicit state transfer rather than depending on hypervisor-level features such as live relocation or VM forking. VM migration is more resource intensive than our approach of protocol state transfer, and not yet fully supported by Xen on ARM. Connection handover is also easily extensible, and we are currently applying it to a full 7-way SSL/TLS hand- shake to support encrypted connections.

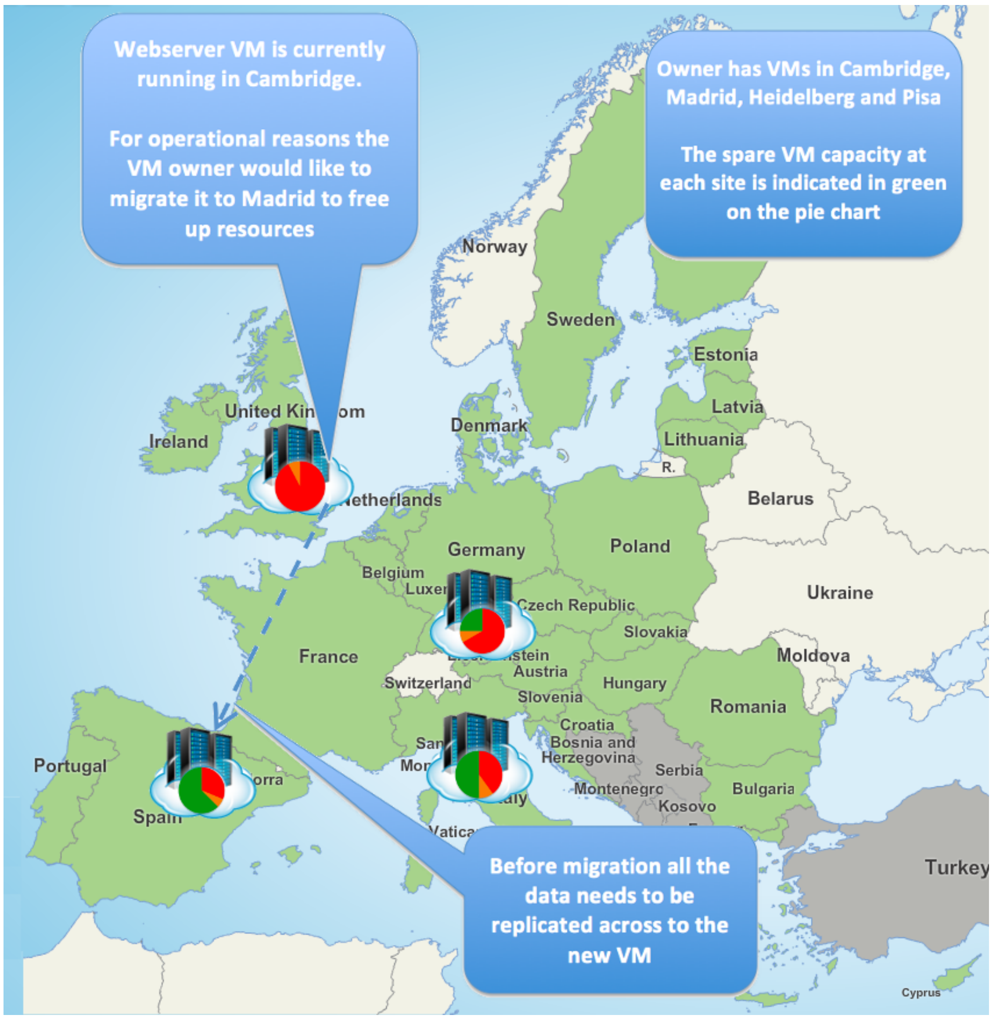

Webserver VM is currently running in Cambridge.
For operational reasons the VM owner would like to migrate it to Madrid to free up resources
Owner has VMs in Cambridge, Madrid, Heidelberg and Pisa
The spare VM capacity at each site is indicated in green on the pie chart
Before migration all the data needs to be replicated across to the new VM

Figure 4.1: Overview of the Virtual Machine Migration use case

4.2 Cloud Liquidity

4.2.1 Use-case design

Migrating Virtual Machines (VMs) between clouds is the ultimate form of Cloud Liquidity. It allows customers to avoid provider lock-in and to have more flexibility and choice when determining where their paid-for services are located. As identified in Deliverable D3.1, VM migration also offers the ability to respond to changing demands, to cope with planned maintenance windows, to load balance between datacenters and to make potential cost and energy savings. More recently live operation of the Federated Market (described in D1.3) has led us to identify three additional motivations: Disaster Recovery as a Service (DRaaS), high availability for business continuity and follow the sun/moon policies. These are based on customer feature requests, customer feedback and work from related projects. These gives additional weight to this use-case as a potentially valuable exploitation route for Trilogy 2. Migrating VM resources within a Datacenter has also been explored as a mechanism to achieve energy efficiency through increasing resource utilisation and powering down under-utilised hardware.

4.2.2 Use-case implementation

Virtual Machine Migration integrates a number of components that have been developed in the course of WP1 (Creating Liquidity) and WP2 (Managing Liquidity). These components rely on the existence and modification of an IaaS (Infrastructure as a Service) platform but the use-case is equally valid for other platforms. The components involved are:

- Inter-Cloud communication (Information Model and connector tools)
- Federated Market platform and tools (to allow resources to be traded)
- Wide-area block replication mechanism (tools in WP1 to enable storage migration)

The current state of each of these components is summarised below:

Inter-cloud Communication: Early in the project we developed the hooks needed to attach OnApp-managed resources to IaaS Cloud platform end-points. The relevant connectors and components that can be used to control OnApp resources from the CompatibleOne[2] ACCORDS[3] platform were open-sourced in year 1 of the project. Customer demand has subsequently led to integration with AWS, however this work is being pursued outside Trilogy 2 for commercial reasons.

For the revised Architecture and the Information Model we have been in discussion with the InterCloud Project[4] and the P2302 group[5] and have investigated alternatives to using separate specialised API connector pieces for each Cloud type, given the expense to support each additional platform. Effort in Y3 is expected to progress a purpose built OpenStack connector to allow provisioning of resources on OpenStack platforms. This will allow comparison between the existing connectors and mechanisms built using using the Information Model and the associated tools that will be developed in Y3. Establishing connectors to other IaaS Cloud platform types allows for the management of resources across a larger and more diverse network of end-points.

Federated Market: We have built a Federated Market platform to aid the creation and management of resources across different Cloud end-points. This uses the connector APIs and communication tools mentioned above. The Market allows a buyer to view resources from a set of sellers, choose a subset based on certain criteria and then specify resources accordingly. There is also a separate platform called Cloud.net which allows VM resources to be purchased from sellers without the need for additional specific infrastructure. More details about how the Federated Market works can be found in D1.3 and D2.4.

Wide Area Block Replication: The final tool required to enable Cloud Liquidity is our storage replication and migration tool. This tool allows for active replication across the wide-area between Cloud provider sites. The replication can be carried out by a single owner or, when authorised, can be performed between two

[2]http://compatibleone.org/
[3]https://github.com/compatibleone/accords-platform
[4]http://www.intercloudtestbed.org/
[5]http://standards.ieee.org/develop/project/2302.html

sites that are owned by different providers. More detail about this tool can be found in D1.3. The storage resources can be replicated and/or migrated between specific end-point. Once storage has been replicated, the process of starting a VM on the other site requires copying the configuration and starting a VM with the same properties on the remote system. Current VM state can also be transferred if migrating a live VM is required, although this is not currently implemented.

4.2.2.1 Migration Example

Currently the wide are block storage replication (described in D1.3) can be used to enable Virtual Machine Migration. The following workflow describes a migration operation on our platform prototype.

(i) The London Cloud manager selects which VM to migrate, in this case 'website VM' - see Figure 4.2.

(ii) Once selected, the manager chooses which migration target to use, in this case Madrid - see Figure 4.3. This target may be in a remote zone that has been set up using the Federated Market.

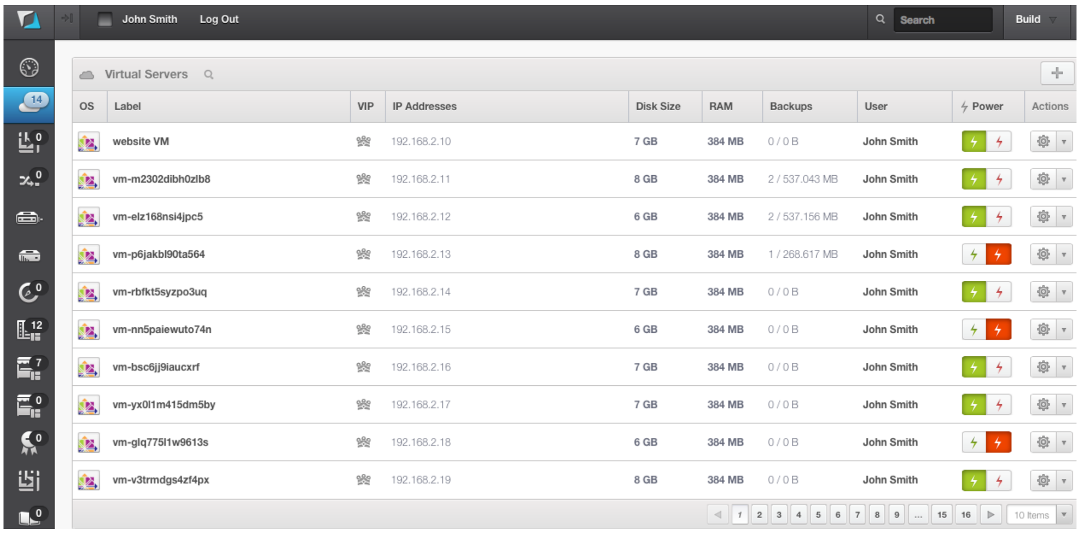
(iii) The Madrid Cloud owner will see a list of VMs associated with the local cloud before the migration - see Figure 4.4. The London Cloud manager will also see this list of resources.

(iv) Before the migration, data is replicated between the two VMs using wide area block replication. Just prior to the actual migration any blocks that have changed will be synchronised.

(v) Once the new remote VM is ready and all data has been copied it can be turned on.

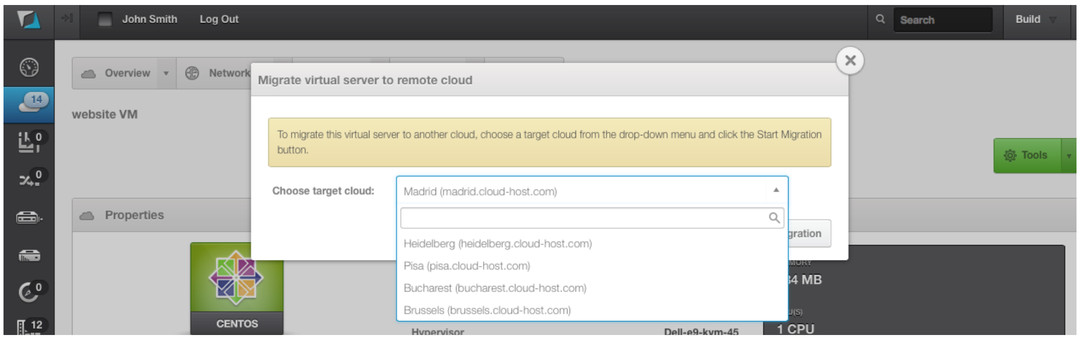
(vi) After the migration, the 'website VM' is now running on the Madrid Cloud - see Figure 4.5.

After the migration the old VM may be removed, re-purposed or may continue to run if the migration was triggered by the need to increase capacity (for instance as a result of a flash crowd hitting a web server).



OS	Label	VIP	IP Addresses	Disk Size	RAM	Backups	User	Power	Actions
	website VM		192.168.2.10	7 GB	384 MB	0 / 0 B	John Smith	🟩	
	vm-m2302dibh0zlb8		192.168.2.11	8 GB	384 MB	2 / 537.043 MB	John Smith	🟩	
	vm-elz168nsi4jpc5		192.168.2.12	6 GB	384 MB	2 / 537.156 MB	John Smith	🟩	
	vm-p6jakbl90ta564		192.168.2.13	8 GB	384 MB	1 / 268.617 MB	John Smith	🟥	
	vm-rbfkt5syzpo3uq		192.168.2.14	7 GB	384 MB	0 / 0 B	John Smith	🟩	
	vm-nn5paiewuto74n		192.168.2.15	6 GB	384 MB	0 / 0 B	John Smith	🟥	
	vm-bsc6jj9iaucxrf		192.168.2.16	7 GB	384 MB	0 / 0 B	John Smith	🟩	
	vm-yx0l1m415dm5by		192.168.2.17	7 GB	384 MB	0 / 0 B	John Smith	🟩	
	vm-glq775l1w9613s		192.168.2.18	6 GB	384 MB	0 / 0 B	John Smith	🟥	
	vm-v3trmdgs4zf4px		192.168.2.19	8 GB	384 MB	0 / 0 B	John Smith	🟩	

Figure 4.2: The list of VM clouds available in London before migrating



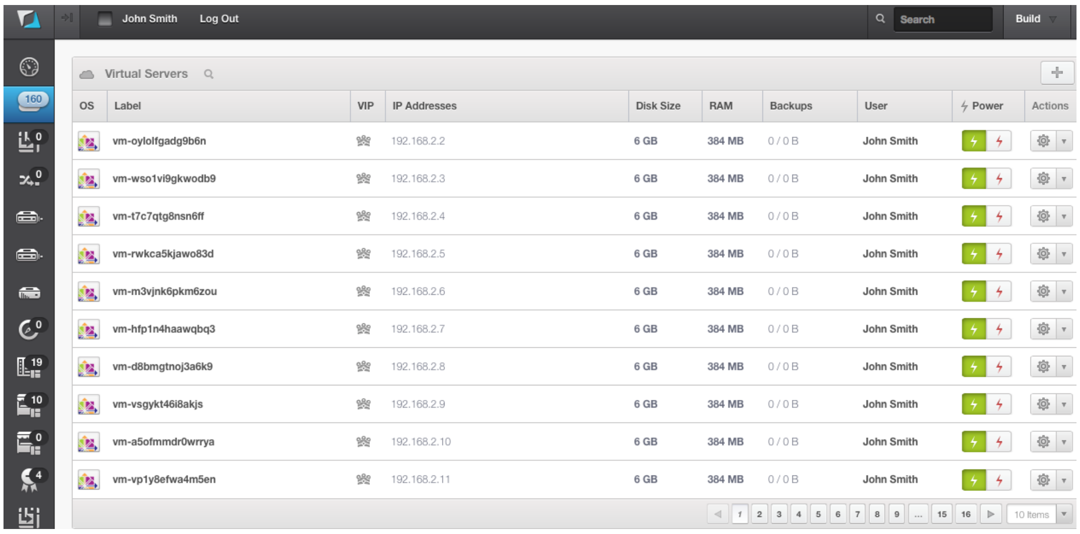
Migrate virtual server to remote cloud

To migrate this virtual server to another cloud, choose a target cloud from the drop-down menu and click the Start Migration button.

Choose target cloud: Madrid (madrid.cloud-host.com)

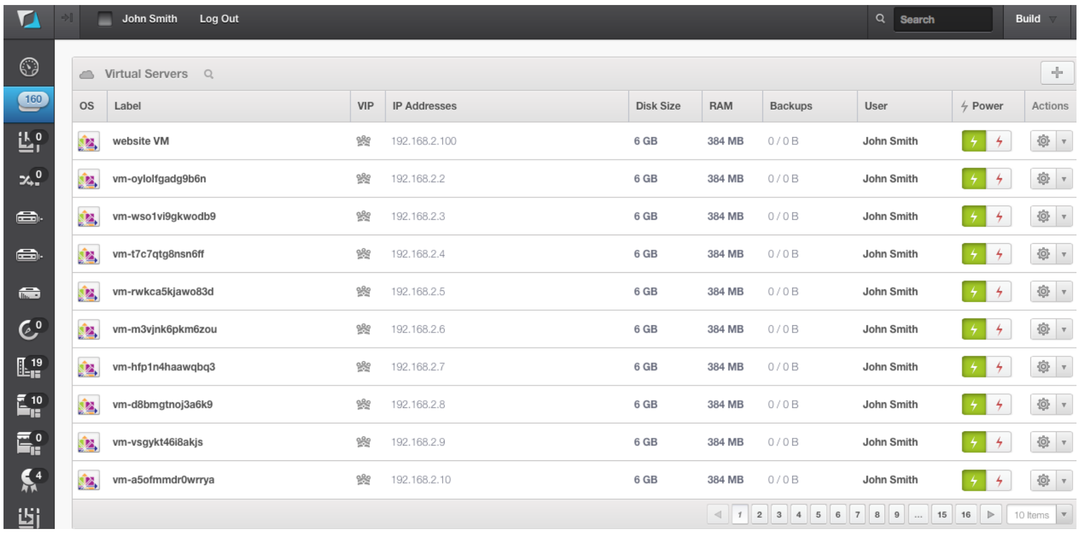
- Heidelberg (heidelberg.cloud-host.com)
- Pisa (pisa.cloud-host.com)
- Bucharest (bucharest.cloud-host.com)
- Brussels (brussels.cloud-host.com)

Figure 4.3: The drop-down box shows the possible targets for migration



OS	Label	VIP	IP Addresses	Disk Size	RAM	Backups	User	Power	Actions
	vm-oylolfgadg9b6n		192.168.2.2	6 GB	384 MB	0 / 0 B	John Smith		
	vm-wso1vi9gkwodb9		192.168.2.3	6 GB	384 MB	0 / 0 B	John Smith		
	vm-t7c7qtg8nsn6ff		192.168.2.4	6 GB	384 MB	0 / 0 B	John Smith		
	vm-rwkca5kjawo83d		192.168.2.5	6 GB	384 MB	0 / 0 B	John Smith		
	vm-m3vjnk6pkm6zou		192.168.2.6	6 GB	384 MB	0 / 0 B	John Smith		
	vm-hfp1n4haawqbq3		192.168.2.7	6 GB	384 MB	0 / 0 B	John Smith		
	vm-d8bmgtnoj3a6k9		192.168.2.8	6 GB	384 MB	0 / 0 B	John Smith		
	vm-vsgykt46i8akjs		192.168.2.9	6 GB	384 MB	0 / 0 B	John Smith		
	vm-a5ofmmdr0wrrya		192.168.2.10	6 GB	384 MB	0 / 0 B	John Smith		
	vm-vp1y8efwa4m5en		192.168.2.11	6 GB	384 MB	0 / 0 B	John Smith		

Figure 4.4: The list of VM clouds available in Madrid before migrating



OS	Label	VIP	IP Addresses	Disk Size	RAM	Backups	User	Power	Actions
	website VM		192.168.2.100	6 GB	384 MB	0 / 0 B	John Smith		
	vm-oylolfgadg9b6n		192.168.2.2	6 GB	384 MB	0 / 0 B	John Smith		
	vm-wso1vi9gkwodb9		192.168.2.3	6 GB	384 MB	0 / 0 B	John Smith		
	vm-t7c7qtg8nsn6ff		192.168.2.4	6 GB	384 MB	0 / 0 B	John Smith		
	vm-rwkca5kjawo83d		192.168.2.5	6 GB	384 MB	0 / 0 B	John Smith		
	vm-m3vjnk6pkm6zou		192.168.2.6	6 GB	384 MB	0 / 0 B	John Smith		
	vm-hfp1n4haawqbq3		192.168.2.7	6 GB	384 MB	0 / 0 B	John Smith		
	vm-d8bmgtnoj3a6k9		192.168.2.8	6 GB	384 MB	0 / 0 B	John Smith		
	vm-vsgykt46i8akjs		192.168.2.9	6 GB	384 MB	0 / 0 B	John Smith		
	vm-a5ofmmdr0wrrya		192.168.2.10	6 GB	384 MB	0 / 0 B	John Smith		

Figure 4.5: The list of VM clouds available in Madrid after migration

4.2.3 Use-case challenges

Establishing connections between remote clouds is a new challenge for the cloud platform. On most cloud IaaS platforms, the user interface allows the management of multiple VMs at a single site. In Figure 4.3, the different cloud targets that are available are configured by establshing a connection between the sites. This is done by creating an API key that allows registration when combined with the IP address of the remote cloud. Additional security checks and user permissions are described in D2.4 (Advanced tools for managing liquidity). Maintaining an accurate resource list requires regular refreshes from each side and poses a potential engineering challenge. Determining a suitable rate of refreshes will be investigated in Y3.

Once an end-target is chosen, the security of the replication tunnel must be ensured across the public Internet. For initial prototypes this is performed by establishing an SSH connection between sites. Longer term it is envisaged that this will use a modern approach such as TCPCrypt. MPTCP could be used to provide a persistent connection where there may be failures in the network. However this is likely have a performance impact that will need to be assessed.

Another issue is how to maintain perceived performance for the bursty workloads that are common in user interactions with the system. Utilising local reads can improve the latency of the system while a remote replication path is open. Synchronous writes carry a performance penalty as they require a round trip time across the slow remote path. Local caching allows writes to be performed more quickly. If the utilisation factor is low, the buffer large and the bandwidth sufficient to allow the buffer to be drained in time, then the perceived performance can be improved by using local writes during loaded periods.

From a usability perspective it is important to indicate to users how liquid resources are being treated by the system and so for Cloud Liquidity we are investigating possible ways of representing VM migration in a manner that is clear to the user and shows the current status of the resources - see Figure 4.1.

One of the other potential challenges is the wide variation in demand for different resources. Figures 4.6, 4.7 and 4.8 show the relative CPU and RAM sizes requested by customers on the Cloud.net platform. As can be seen the majority of customers want 512MB of RAM, but some request as much as 5GB. CPU resources also vary from 1 to 4 vCPUs and Disk sizes vary between 0 and 60GB. This adds significantly to the potential variability of VM migration.

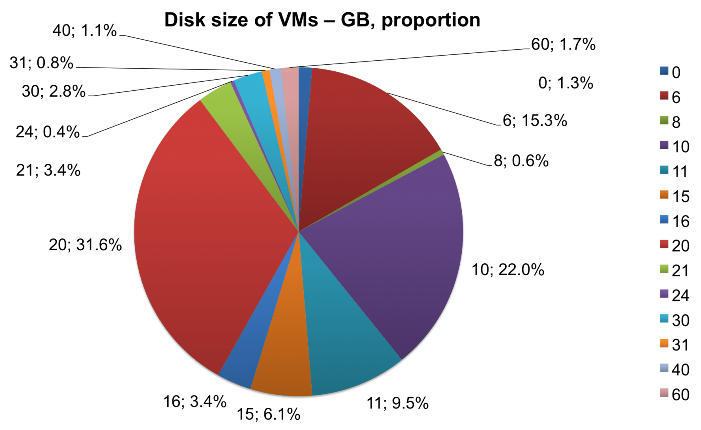


Figure 4.6: VM disk sizes requested on Cloud.net

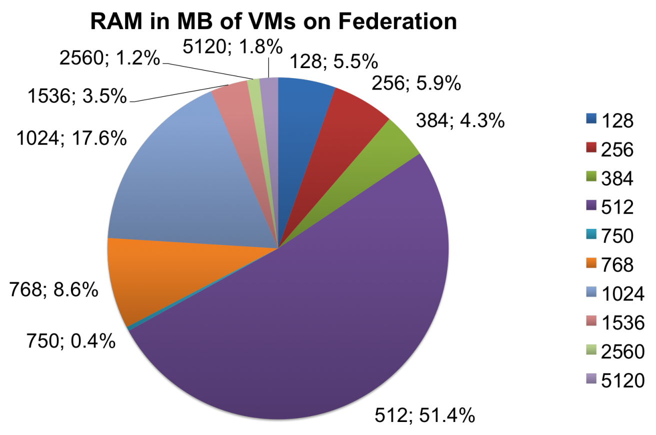


Figure 4.7: VM RAM sizes requested on Cloud.net

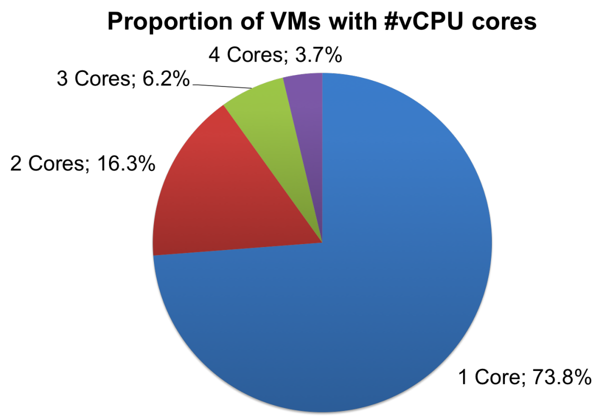


Figure 4.8: Number of vCPUs per VM requested on Cloud.net

5 Conclusion

To conclude this deliverable, we provide the a summary of the selected use-cases and the tools they require.

Use case	Tools
Migrating virtual machines	MPTCP (D1.1) Virtual Machine Migration for Mobile Devices (D1.1 and D1.3) Interactions between processing and bandwidth liquidity mechaisms (D1.2)
WiFi Channel Switching	MPTCP (D1.1 and D1.3) MPTCP and channel switching interaction (D1.2)
Trusted resource pooling with Multipath TCP	MPTCP (D1.1) MPTCP Scheduler infrastructure (D1.3)
ClickOS migration	ClickOS MPTCP (D1.1) Waypoint migration: Moving the “middle” of a connection (D2.2)
Control Plane for Liquid MPLS VPNs	Virtual CPE (D1.1 and D1.3) Network Function Virtualisation (D2.1)
Providing resilience to IMS virtualised network functions	Network Function Virtualisation (D2.1)
Minicache video delivery	Minicache
Hot-fix propagation with Irminsule	Irminsule (D1.1 and D1.3)
Cloud Liquidity	Federated Market (D1.1, D1.3 and D2.4) Block replication (D1.3)

Bibliography

[1] 3GPP. IP Multimedia (IM) Subsystem Cx and Dx Interfaces; Signalling flows and message contents. TS 29.228, 3rd Generation Partnership Project (3GPP), September 2008.

[2] 3GPP. IP Multimedia Subsystem (IMS); Stage 2. TS 23.228, 3rd Generation Partnership Project (3GPP), September 2008.

[3] Amazon. Amazon Elastic Compute Cloud (EC2). `https://aws.amazon.com/ec2/`, May 2014.

[4] Pedro Andrés Aranda Gutiérrez. Scapy community branch: MP-BGP fork. `https://github.com/paaguti/scapy-com`, dec 2014. note.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. ACM SOSP, 2003*, New York, NY, USA, 2003. ACM.

[6] Paolo Bellavista, Antonio Corradi, and Luca Foschini. Enhancing intradomain scalability of ims-based services. *Parallel and Distributed Systems, IEEE Transactions on*, 24(12):2386–2395, 2013.

[7] Cisco Systems. Cisco Visual Networking Index: Forecast and Methodology, 2012–2017. `http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf`, 2012.

[8] Crucial. *Crucial m4 SSD 2.5" Technical Specification*, 2011.

[9] Erlang on Xen. Erlang on Xen. `http://erlangonxen.org/`, May 2014.

[10] J. Erman and K. Ramakrishnan. Understanding the Super-sized traffic of the Super Bowl. In *ACM IMC*, 2013.

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

[12] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. A First Look at Traffic on Smartphones. In *ACM IMC*, 2010.

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

[14] Benjamin Frank, Ingmar Poese, Yin Lin, Georgios Smaragdakis, Anja Feldmann, Bruce Maggs, Jannis Rake, Steve Uhlig, and Rick Weber. Pushing cdn-isp collaboration to the limit. *SIGCOMM Comput. Commun. Rev.*, 43(3):34–44, July 2013.

[15] Andres Garcia-Saavedra, Pablo Serrano, Albert Banchs, and Giuseppe Bianchi. Energy consumption anatomy of 802.11 devices and its implication on modeling and design. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 169–180, New York, NY, USA, 2012. ACM.

[16] GitHub. Modern HTTP benchmarking tool. `https://github.com/wg/wrk`, July 2013.

[17] D. Goodin. Guerilla researcher created epic botnet to scan billions of ip addresses. `http://goo.gl/G86ew`, March 2013.

[18] HalVM. The Haskell Lightweight Virtual Machine Source Archive. `https://github.com/GaloisInc/HaLVM`, May 2014.

[19] J. Huang, F. Qian, Y. Guo, Y. Zhou, and Q. Xu. An in-depth Study of LTE: Effect of Network Protocol and Application Behavior on Performance. In *ACM SIGCOMM*, 2013.

[20] J. Iyengar, P. Amer, and R. Stewart. Concurrent Multipath Transfer using SCTP Multihoming over Independent End-to-End Paths. *IEEE/ACM Transactions on Networking*, 14(5):951–964, 2006.

[21] C. Jackson and A. Barth. Forcehttps: Protecting High-Security Web Sites from Network Attacks. In *WWW*, 2008.

[22] Nikolaos Laoutaris, Pablo Rodriguez, and Laurent Massoulie. Echos: edge capacity hosting overlays of nano data centers. *SIGCOMM Comput. Commun. Rev.*, 38(1):51–54, January 2008.

[23] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong. Mobile Data Offloading: How Much Can WiFi Deliver? *IEEE/ACM Transactions on Networking*, 21(2):536–550, 2013.

[24] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, March 2013.

[25] R. Mahy, B. Biggs, and R. Dean. The Session Initiation Protocol (SIP) "Replaces" Header. RFC 3891 (Proposed Standard), September 2004.

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

[27] Microsoft Corporation. Azure: Microsoft's Cloud Platform. `azure.microsoft.com/`, May 2014.

[28] J. Milliken, V. Selis, and A. Marshall. Detection and analysis of the Chameleon WiFi access point virus. *EURASIP Journal on Information Security*, 2013(1):1–14, 2013.

[29] Kenneth G. Paterson Nadhem J. AlFardan. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. *Royal Holloway on Information Security*, 2013.

[30] Nodejs.org. node.js. `http://nodejs.org`, August 2013.

[31] OSv. OSv - The Operating System Designed for the Cloud. `http://osv.io`, May 2014.

[32] Rackspace. Public Cloud Hosting, Computing, Storage and Networking. `http://www.rackspace.com`, May 2014.

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

[34] L. Rizzo and G. Lettieri. Vale: a switched ethernet for virtual machines. In *Proc. ACM CoNEXT*, December 2012.

[35] J. Rosenberg and H. Schulzrinne. Session Initiation Protocol (SIP): Locating SIP Servers. RFC 3263 (Proposed Standard), June 2002.

[36] Sandvine Inc. Sandvine global internet phenomena report. `http://www.sandvine.com/downloads/documents/Phenomena_2H_2012/Sandvine_Global_Internet_Phenomena_Report_2H_2012.pdf`, 2012.

[37] J. Vehent. SSL/TLS analysis of the Internet's top 1,000,000 websites, 2014. `https://jve.linuxwall.info/blog/index.php?post/TLS_Survey`.

[38] C.A. Visaggio and L.C. Blasio. Session Management Vulnerabilities in Today's Web. *IEEE Security & Privacy*, 8(5):48–56, 2010.

[39] Patrick Wendell and Michael J. Freedman. Going viral: flash crowds in an open cdn. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measument conference*, IMC '11, pages 549–558, New York, NY, USA, 2011. ACM.

[40] Wikipedia. Content Delivery Network. `http://en.wikipedia.org/wiki/Content_delivery_network`, 2013.

[41] Wikipedia. Tilera. `http://en.wikipedia.org/wiki/Tilera`, 2013.

[42] D. Wischik, M. Handley, and M. Bagnulo. The resource pooling principle. *SIGCOMM Comput. Commun. Rev.*, 38(5), September 2008.

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