ICT-317756

TRILOGY2

Trilogy2: Building the Liquid Net

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

D3.3 - Use Case Deployment

Due date of deliverable: December 30, 2015
Actual submission date: December 30, 2015

Start date of project	1 January 2013
Duration	36 months
Lead contractor for this deliverable	Universidad Carlos III de Madrid
Version	1.0, December 31st 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	WP3, Using Liquidity
Editor	Felipe Huici, NEC
Project Co-ordinator	Marcelo Bagnulo Braun, UC3M
Copyright notice	© 2015 Participants in project TRILOGY2

Executive Summary

This deliverable describes a wide range of use cases that create liquidity in three fundamental areas: mobile devices, operator infrastructure and wide area networks. We present detailed descriptions of the use cases' implementations as well as extensive evaluation results from their deployment on testbeds and on real operation networks.

The use cases in the area of *mobile devices* focus on enhancing the performance or experienced quality of mobile apps and devices. Transparent migration facilitates the migration of applications between devices in a fashion transparent to the user, with minimal performance and no functional impact. While evaluation shows some limitations to the technique, it is viable especially for apps with low to medium memory usage, when migrating over reasonably fast links (>100 MBit/s). Two further use cases show the wide applicability of Multipath TCP (MPTCP) for liquidity in mobile devices. MultiWifi allows mobile devices to associate with several WiFi access points at the some time, using MPTCP subflows to stay connected over both links. This not only allows seamless mobility, but also can increase throughput for the mobile device, using available bandwidth of several networks concurrently. This section concludes with a measurement study that analyzes the behavior of Multipath TCP on real-world mobile devices with several connections (cellular and WiFi).

The section on *liquidity in operator infrastructure* considers use cases inside data centers or edge-deployed infrastructure. ClickOS uses specialized minimalistic unikernel virtual machines (VMs) that are designed for scenarios such as Network Function Virtualisation (NFV) which, due to their small size, allow massive deployments, e.g., several thousand VMs on a single server, as well as fast (<1 second) migration times. Quagga is an open-source implementation of a control plane for an IP router. Within the scope of this project, it was extended to support Multi-Protocol Label Switching (MPLS) for enhanced BGP support and evaluated for use as a carrier-grade NAT. Another outcome of the project is a novel resilience and scaling mechanism for IP Multimedia Subsystem (IMS) networks that allows transferring users transparently among the functional elements that implement the IMS call session functions. Experiments show that this system can support much higher rates of sudden user transfers (e.g., for failover reasons) than the original system. The final contribution in this section present Minicache, a minimalistic VM-based data cache that leverages a novel filesystem as well as extensions to previous minimalistic VMs such as Mini-OS. Due to its small size, it can be deployed very fast and even on resource-constrained embedded edge devices. Its performance is shown in real-world tests, while the potential benefit of Minicache on CDNs is evaluated in simulation.

As a third area, this deliverable considers *liquidity in the wide area*. First, Irmin, a data repository for unikernels, is used for rapid deployment of updated versions of VMs. This allows version control of full (minimalistic) operating systems, facilitating easier management of patches in large cloud deployments. Second, cloud liquidity envisions a system to migrate data, workloads and even complete virtual machines between clouds with no restrictions on providers or virtualisation technology. A prototype of such a system, complete with a UI for easy control, is presented using the example of Disaster Recovery as a Service (DRaaS), which allows continuous synchronization of VMs for rapid failover in disaster cases.

List of Authors

Authors	Felipe Huici, John Thomson, B. Hesmans, C. Moraru, C. Raiciu, G. Bernini, J. Crowcroft, O. Bonaventure, Q. De Coninck, P. Aranda, T. Moncaster, J. Chesterfield, J. Garcia-Reinoso, I. Vidal, G. Carozzo, E. Kraja, P. Cruschelli.
Participants	NEC, UCL-BE, UCAM, Intel, NXW, UPB, UC3M, ONAPP, TID
Work Package	WP3, Using Liquidity
Security	Public (PU)
Nature	R
Version	1.0
Total number of pages	63

Contents

List of Figures

List of Tables

1 Introduction

In this deliverable we give detailed descriptions of implementation and evaluation results of the Trilogy2 use cases when deployed on network testbeds or operational networks. It is organized into three main areas. First, we present use cases that target liquidity in mobile devices. These include transparent migration of running applications between mobile devices; enhancements to MPTCP so that it can more efficiently make use of multiple, concurrent access points; and results from two measurement campaigns of MPTCP on mobile devices.

The second area covers liquidity in operator infrastructure. Under this heading, we present results from ClickOS and Xen optimizations that provide NFV liquidity, including massive consolidation and fast service instantiation and migration, among others. Further, we introduce a control plane for liquid MPLS VPNs, mechanisms to provide resilience to virtualized IMS functions, and the ability to build on-the-fly, virtualized CDNs with Minicache.

The final area focuses on liquidity in wide area networks. Within this, we describe the implementation of Irmin, a system for liquid, fast and continual updates of cloud operating software, including unikernels. The area further includes cloud liquidity, which facilitates continuous synchronization of VM states for load balancing as well as high availability and disaster recovery.

All of these items consist of actual implementations and deployments on testbeds or actual operational networks. In the rest of this deliverable we provide detailed descriptions of each of them and present evaluation results from them.

2 Liquidity in Mobile Devices

2.1 Transparent Migration

In this sections we present the challenges and results of deployment for the transparent migration technology. Virtual Machine Migration Manager (vM3) is the tool that takes advantage of transparent migration technology. As detailed in deliverables D3.1 and D3.2, the use-case is to make it possible for a mobile application to switch the device it is running on. This migration must preserve the running state of the application and have no impact and its functionality and minimal impact on performance.
The impact of releasing the vM3 tool is discussed not only in terms of performance evaluation, but also in terms of licensing.

2.1.1 Impact of open source release

A key aspect of every project deployment is licensing. The vM3 applications that represents the transparent migration use-case developed during Trilogy2 will be released as open-source. There are two motives for this decision: visibility and feedback. An open-source project has the best chances to be on the radar of developers and other innovation entities like universities and development centers. Because the transparent migration technology is not yet ready as a product, the main target is not the users, but the developers that can actually help with the evolution process of this technology.
With having the code open-source, there is a far better change to improve on the technology by having feedback not only on about the usage, but also on implementation decisions. This is really a key factor with every new technology that aims to impact the way things work, and this is also true for MPTCP. Also, this decision is a natural one taking into account that the development team for vM3 is part of the Intel Open Source Technology Center (OTC) division.

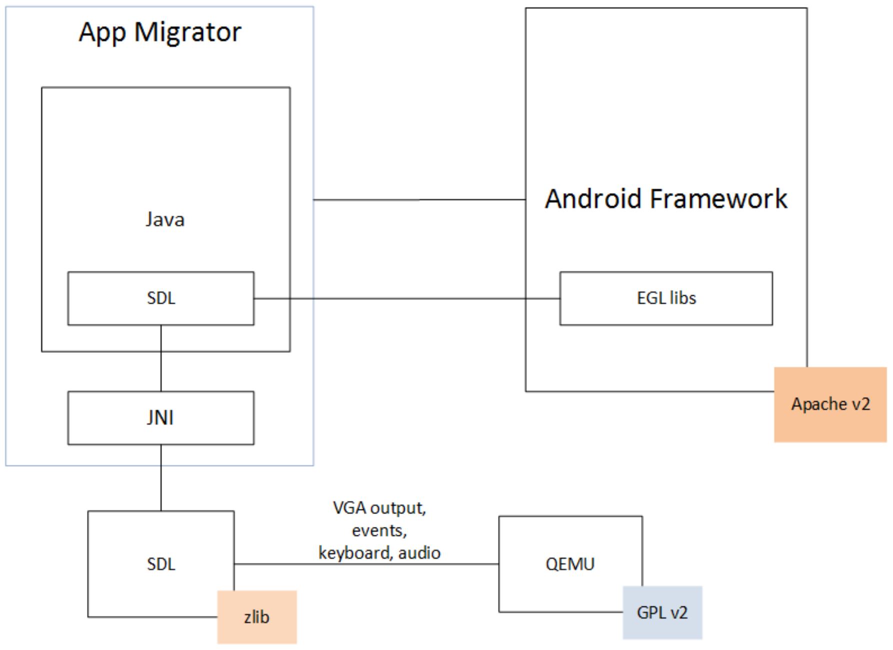


Figure 2.1: Licence tree of VM3 components

The decision to make vM3 open-source is related to the licensing issue. The first intended licence for the vM3 was BSD. This licence has the advantage of unrestricted access to the code, with the benefit of protection against improper usage of the code's provider name. Figure 2.2 presents a quick view on the components of vM3, from a licensing point of view. The main components involved have very diverse licences. There is the Qemu hypervisor responsible for the virtual machine existence, and thus a key component of the project, has the GPLv2 licence. The SDL library, which links the input/output from the hypervisor to the management component represented by the Android application, has the more liberal zlib licence. This is also a key component, but not an irreplaceable one since it is only "glue" code. The other key component is the Android

application that manages the lifetime of a virtual machine encapsulated application. The Android application takes advantage of the Android Framework to be able to use the system resources. The Android Framework is released under the Apache v2 licence.
Furthermore, it is interesting to give some details on the way the components are linked. It is worth mentioning that the Android application is statically linked with the Qemu hypervisor. The call flow starts in Android app java code and crosses over to the SDL code through JNI code, and then through callbacks into the hypervisor code. This is not the only means of communication between the Android application and the Qemu hypervisor.

2.1.2 Testing setup

The purpose of this section is to have a performance evaluation of the vM3 tool. At the time of writing the process of releasing the project as open-source is not finished. Because of this, we have no real-world experimental results. Instead, we focused on having relevant benchmarks especially for application developers. It is important to have a general view of how well things work before trying to have a specific application as a migratable application.
Because the key ingredient to this applications is transparent virtual machine migration, benchmarking has focused on specific virtual machine migration parameters. Evaluation is carried out upon the following dimensions:

- Migration time: How much does it take from the moment the migration is issues in the UI until the applications resumes on the other side.
- Downtime: The amount of time when the application is unavailable on any device. This should be as small as possible
- Type of application: Different workloads heavily influence migration times.
  - CPU intensive apps
  - Memory intensive apps
  - Network intensive apps
  - Idle app – Used for reference
- RAM size: Different applications have different memory constraints. It it interesting to see how this affects migration speed.
- VPCU: Most mobile device this days are multicore – hence the number of VPCU's allocated to the VM can vary
- Network speed: With so many different Wifi standards being active at the same time and also changing conditions in

Since we are interested in the previous mentioned parameters, we only used one type of hardware, an Acer Iconia W700 Tablet. The hardware parameters relevant to testing are: Intel Core i5-3337U CPU (4 cores with virtualization capabilities), 4096 MB RAM, USB 3.0. The last part is important also since we connected the tablet through and Ethernet-to-USB adapter witch assures network speeds up to GB level. We chose to test the solution from the networking point of view this way because we wish to test migration between devices, not how the application reacts in face of network failures or real life networks.
Figure 2.2 highlights the general setup topology for the benchmarks. The colored part of the schema reflects the parameters that will vary. There are two tablets connected to each other through a switch, representing the source and destination of the app migration. Another station is also connected to the switch and will provide a video stream service accessible via web interface. All tests involve the following basic steps: on the first tablet vM3 will be started with specific virtualization parameters and a specific applications. The virtualization parameters such as RAM size and number of VPCUs will be passed through to the hypervisor, qemu. The application type is set up by selecting different application types at vM3 startup. After full initialization, this application will be migrated to the other device tablet, where a similar vM3 instance is running. This vM3 instance starts with the same RAM and VCPU parameters, but with no application started.

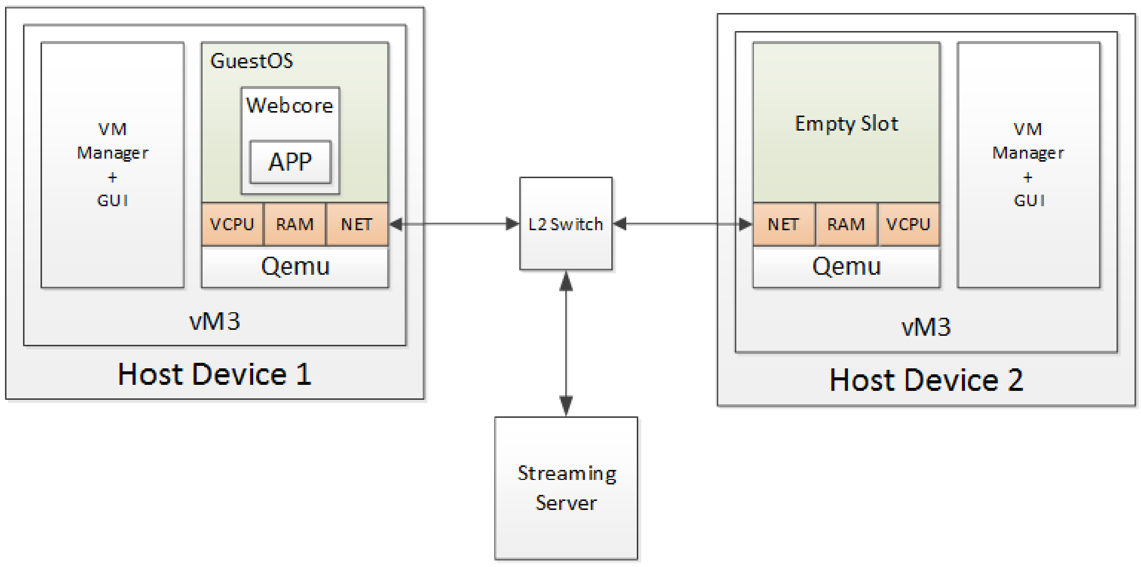


Figure 2.2: Benchmarking setup

It will instead wait for incoming migration requests. The measurements are performed around the migration, the migration time being one key element. This migration time is double checked from two places. It is first reported as a debug message by the hypervisor that signals the migration end and how much time it has lasted, in milliseconds. The second check is done by predefined time stamped messages with migration start and migration end on both tablets, after a clock synchronizations. It is notable to state that this synchronization is not needed in a non-test scenario. Another important parameter is downtime. This will only be reported by the hypervisor.The switch is the one responsible with tweaking the network speed. It is a gigabit capable switch that has the ability to limit the network speed.

2.1.3 Results

Figure 2.3 shows a comparison of migration times from the point of view of network speed and allocated RAM size. The type of application in this case is the one described in the previous deliverables D3.1 and D3.2 – the live video streaming app. It is to be noted that this application is more networking rather than CPU or memory intensive. This is because data packets with video frames only alter the network buffers and video buffers, and after visualization are discarded. The live streaming app manages to transmit part of the video before migration, and the rest after the app is migrated, to the correct destination. The results show us that there is not much video frame rerouting: little video data sent from the server gets to Host1 and then is redirected through the memory migration process to Host2.
Figure 2.4 shows a generic comparison between the types of applications to be migrated. As previously stated, the Idle app represent only the VM container overhead. The migration times are comparable with the CPU app. This is explained by the fact that both apps have a very low memory usage. The CPU intensive app works mostly with the registers and cache. Once the app is suspended, the register states are saved and sent over the network, while the cache content is discarded. So the CPU intensive app practically has no networking consumption overhead – in terms of migration – when compared with the Idle app. The only difference is due to the competition for the CPU between the migration thread and the thread that runs the VM. The networking app presents a slight performance degradation besides the other two types of applications. This is because it is hard to have a pure networking app, with minimal memory impact. The networking app must send and receive data, and must store this data in buffers throughout the networking stack. From the migration point of view, this is a process of “memory trashing”, and the same memory pages holding the networking buffers must be sent over and over again. This is still not as bad as in the case of memory intensive app. The post-copy migration algorithm has a threshold for how many times a page can be transmitted. If there is a small number of pages that have to be constantly resend, then the CPU is stopped and all remaining pages are transmitted to the destination. This does not work very well in the case of a constantly changing pattern of pages accessed, like in the case of the Memory access intensive app. Here, the app in question is more of a

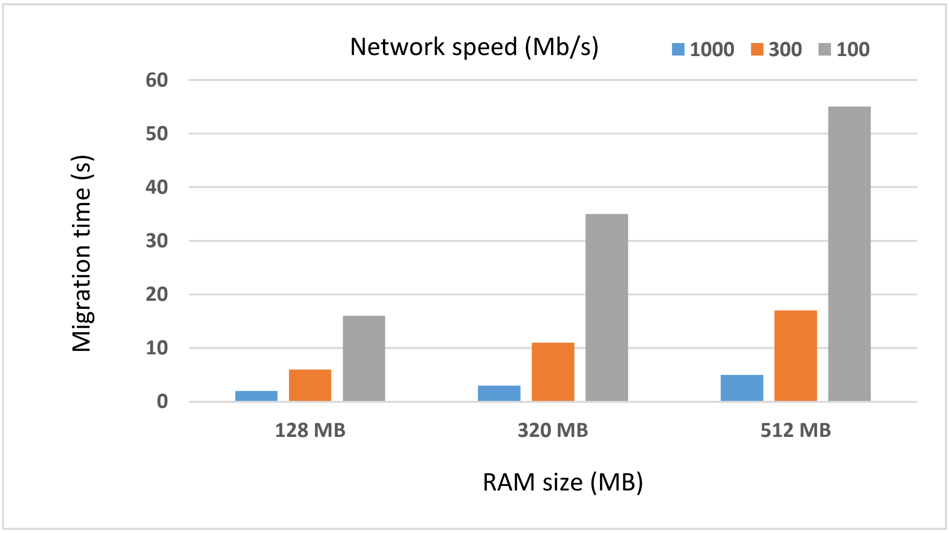


Figure 2.3: Test results comparing the migration time with different RAM sizes and network speeds

synthetic one, with measurement purposes in mind. The app is a C program that allocates memory pages and writes in them at random. The rate that pages are touched by the program is 2pages/second. If we double the rate, the migration process becomes problematic and does not finish for 100Mb/s and 300Mb/s speeds.

2.1.4 Evaluation

From the previous section we can draw a set of conclusions about transparent migration as a technology. First, not every type of application is suited to be live migrated. From the above tests, it seems that only applications that are inclined to use more CPU and Network are fitted to be live-migrated within a reasonable time frame, rather than the ones that are memory intensive. Another takeaway from this set of tests is that the live-migration app becomes viable at medium and high speeds. This means that most of the current wireless LAN technologies are not really fit for live-migration.

2.2 Multi WiFi results

In this section we present the simulation and implementation results for the methods introduced in Deliverable D1.3, Section 3.2. There, we proposed two basic methods to allow a client to connect to multiple APs: on the same channel, and on multiple channels. Staying on the same channel, a client has the chance to use its virtual interfaces to connect to all APs visible, and exploit each link separately using an MPTCP subflow. When APs are available on different channels, the client may employ channel switching to share its time on each channel, again with a virtual interface. Both cases are similar from a routing and MAC naming perspective, in that the client holds two different MAC and IP identities to connect to different APs. In the next sections (2.2.1, and 2.2.2) we present results for the same channel case, while in Section 2.2.3 the results for the channel switching procedure.

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: for one, the loss rate perceived by MPTCP on each subflow does not reflect the efficiency of the AP routing 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. Secondly, when subflows have shared bottlenecks, MPTCP assumes that

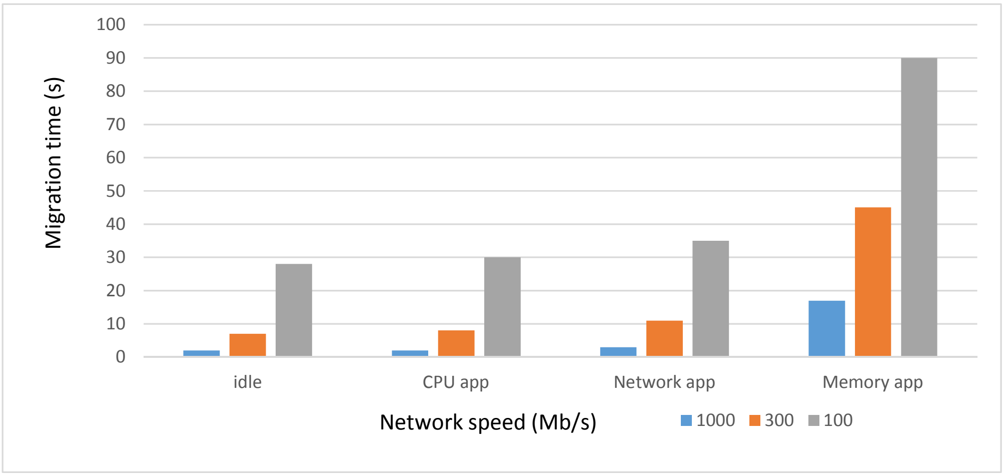


Figure 2.4: Test results between different types of app, using the same allocate RAM 320MB

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, it is simplest to stop using APs that decrease total throughput, but this comes at the expense of poorer performance when mobile. A more sensible option is **on–off**: receive data from a single AP at any point in time while all the others are shut-off, and periodically cycle through all available APs. **on–off** has already been proposed in the context of channel switching [21, 33, 16] and we use it in Section 2.2.3 to cope with APs on different channels. In our context, **on–off** can be implemented either by using the MP_BACKUP mechanism in MPTCP [12] which allows prioritizing subflows, or by relying on WiFi power save mode. It seems natural to extend the **on–off** solution for single channel APs as in [16], since there is no real cost of "switching" between APs on the same channel, beyond a couple of control messages: there is no wasted airtime. However, there are also a few fundamental drawbacks:

- Switching between APs increases packet delay and jitter, which affects interactive traffic. For instance, with a 200ms duty cycle, many packets experience RTTs that are 200ms longer that the path RTT.
- Gains from channel independence are lost.
- When multiple clients make independent decisions to switch between APs, they may end-up switching at the same time to the same AP, wasting capacity available elsewhere. Simulation results in Section 2.2.3 that show around 35% of the available capacity can be wasted in such cases.

Clients can monitor local PHY/MAC conditions accurately, but have a limited view of end-to-end downlink capacity available via an AP, because end-to-end loss rate and RTT are only available at the TCP sender. The sender, on the other hand, uses inaccurate metrics that are influenced by the WiFi behavior. For these reasons, our solution allows different APs to simultaneously send to the same client, while allowing the MPTCP congestion controller to direct traffic to the most efficient APs. In particular, our MPTCP client uses local WiFi information to find out which APs are preferable, and relays this information to the sender as additional loss notification. One non work-conserving way to relay this information is to have the client drop a percentage of packets. Instead, we use explicit congestion notification(ECN) to tell the server to avoid, if possible, the bad APs.

Our solution has two distinct parts discussed next: a novel client-side passive measurement technique that allows the client to accurately estimate the efficiency of each AP, and an algorithm that decides the amount of ECN marks that a client can set on a subflow to enable efficient airtime usage.

2.2.1.1 Measuring AP Efficiency

When deciding which AP it should prefer, the client needs to estimate the time T_i it takes on average for AP i to successfully send a packet, assuming the AP is alone in accessing the wireless medium. This metric purposely ignores the effect other WiFi clients may have on the transmission time 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.

We model the packet delivery time T (if the client were alone), when using the bitrate MCS at the AP and a PHY loss rate p with R retransmissions per packet, ignoring packets undelivered after R retries:

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

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 MCS (1Mbps). 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 MCS 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 N_0, the number of frames received with the retry bit set to 0. All the other frames reaching the client will have the retry bit set to 1, and are counted as N_1. We 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 successive attempts as shown by the sequence number present in the MAC header.

Implementation. To accurately estimate 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.

2.2.1.2 Helping senders make the right choice

Consider the two scenarios depicted in Figure 2.5, where AP_2's packet time is shorter than AP_1's, and the two subflows going via AP_1 and AP_2 do not interfere at the last hop. MPTCP congestion control [49] 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 using a simple model of TCP throughput, $B = \sqrt{\frac{2}{p}} \cdot \frac{MSS}{RTT}$. With this estimate, MPTCP adjusts its overall aggressiveness (total congestion window increase per RTT over all its subflows) so 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.

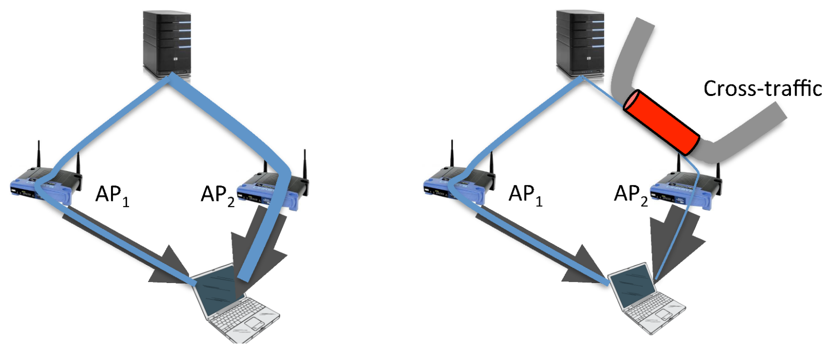


Figure 2.5: Scenario 1 (left): a client using AP_1 and AP_2 prefers AP_2 because of its more efficient use of airtime. Scenario 2 (right): moving all traffic to AP_2 with its better radio conditions is not the optimal strategy end-to-end

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 use ECN to signal the sender 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

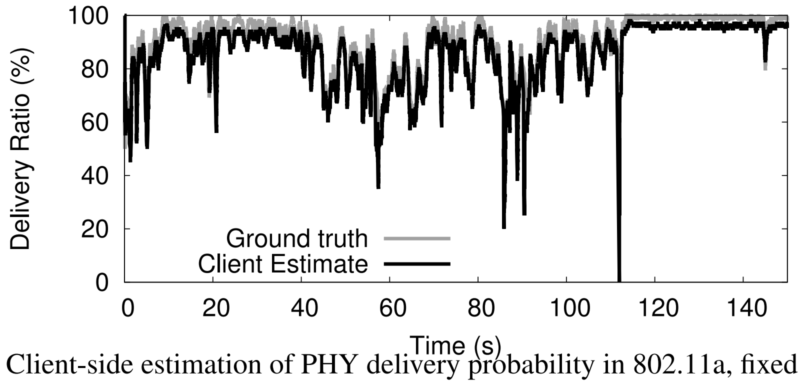


Figure 2.6: Client-side estimation of PHY delivery probability in 802.11a, fixed rate (54Mbps)

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 ECN marking happens before the packets get delivered to IP processing code at the client.

2.2.2 Evaluation

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 fairness to existing clients and performance for short downloads. 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.

2.2.2.1 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 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. MPTCP congestion control (we use the OLIA algorithm [22]) reacts fairly quickly to moderate differences in loss rates (20% higher loss rate on one path). Experiments show that it takes between 1s and 2s for traffic to shift to the better path once an abrupt change has occurred, when the RTT is 50ms.

The client downloads a large file and we plot its estimation of the delivery probability (relation (2.2)) 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 2.6 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 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 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.
- In 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.

2.2.2.2 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 Figure 2.12 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, as the whole distance was covered in around 10s. The results (not shown) are much noisier, but show that the ECN mechanism still works fairly well overall; a similar result with a mix of 11n and 11g is later discussed in Figure 2.13. All further experiments presented in this document are run with the ECN algorithm enabled, unless otherwise specified.

2.2.2.3 Simulation analysis

To understand how our proposed marking algorithm works in a wide range of scenarios, we also implemented it *htsim*, a scalable simulator that supports MPTCP and that has been used in previous works [49, 41]. *htsim* does not model 802.11; instead, we implemented a new type of queue that models the high level behavior of shared access in 802.11 DCF, namely: different senders' (AP) packets take different time for transmission on the medium, and when multiple senders have packets ready, packets are chosen according to a weighted fair-queueing algorithm, with predefined weights for the different APs.
Using this simple model, we can explore specified outcomes in the MAC contention algorithm, for example when AP1 wins contention four times more often than AP2, that are difficult to obtain in 802.11 setups simply

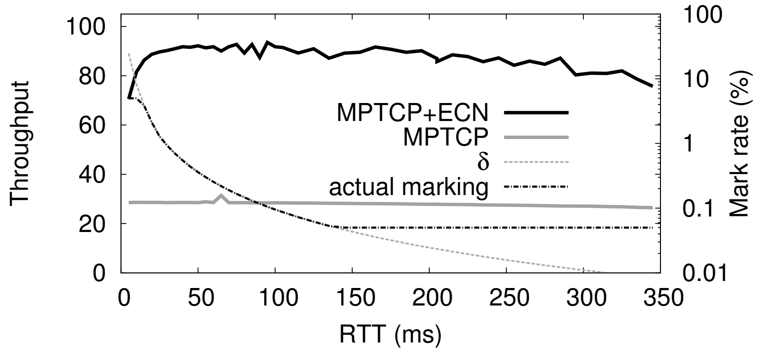


Figure 2.7: The marking algorithm provides 80% of the optimal throughput over a wide range of RTTs.

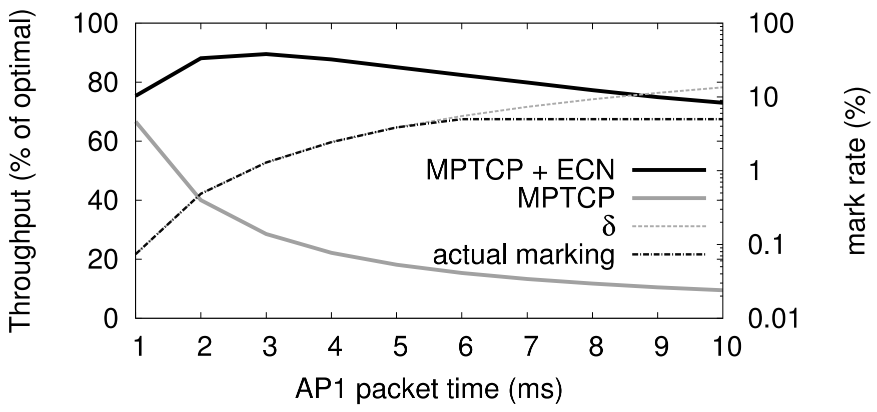


Figure 2.8: Varying the quality of the AP_1 link has little effect on the throughput experienced by the MPTCP client when marking is active.

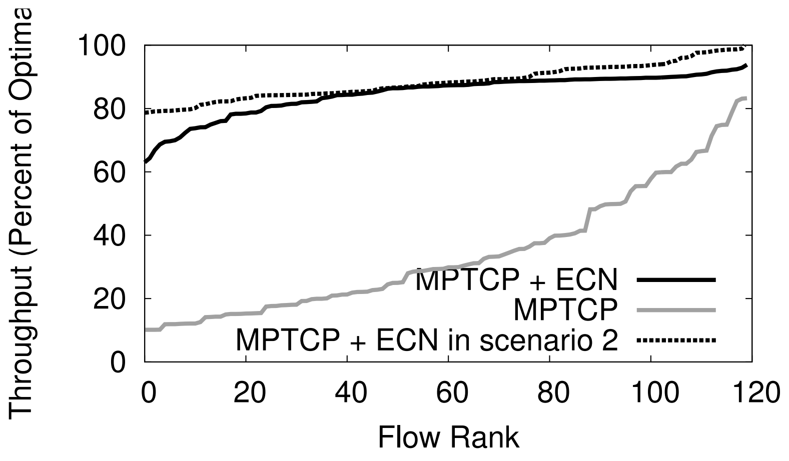


Figure 2.9: Flow throughput across a wide range of parameters. Marking achieves on average 85% of the optimal throughput.

by choosing different rate selection algorithm. Our simulated topology is shown in Fig. 2.5a, where the client is using both AP_1 and AP_2. In all our tests, AP_2 offers a perfect 802.11a/g connection (max 22Mbps), meaning that T_2, the packet time for AP_2 is set to 0.5ms.

We ran simulations testing all the combinations of parameters important in practice: RTT (10, 25, 50 and 100ms), T_2 (from 1ms to 6ms), and the weights for different APs (1, 2, 4, 8 or 16). We ran a total of 120 simulations, and we present the throughput obtained as percentage of the optimal, sorting all values ascendingly. Figure 2.9 shows that the ECN marking algorithm is very robust: its average performance is 85% of the optimal (median is 87%), and its worst case is 65%. In contrast, MPTCP alone fares poorly: 34% throughput on average (28% in the median). Finally, the throughput of MPTCP in Scenario 2 is also robustly close to the optimal: average at 84% and median at 88%.

There are parameter combinations where the ECN algorithm is not as effective: when RTTs are small, δ is fairly high so ECN does manage to reduce the congestion window over AP_1. However, even a very small window of packets sent via AP_1 is enough to eat up a lot of airtime that would be better used by AP_2, and this effect is more pronounced when AP_1 wins the contention more often, because it has fewer retries.

In all experiments, we cap the marking rate to a maximum of 5% to avoid hurting throughput in Scenario 2. This is a direct tradeoff: the higher the allowed rate, the better MPTCP behaves in scenario 1, but the worse it behaves in scenario 2. The reason for this behavior is that the traditional formula used by MPTCP to estimate throughput over its subflows is overly optimistic for higher loss rates, where retransmit timeouts begin to dominate throughput.

To analyze scenario 2, we use a setup where $T_1 = 3ms$ (approx. 4Mbps), $RTT = 25ms$ and vary the number of TCP flows contending for the uplink of AP_2, whose speed is set to $25Mbps$. Figure 2.10 shows

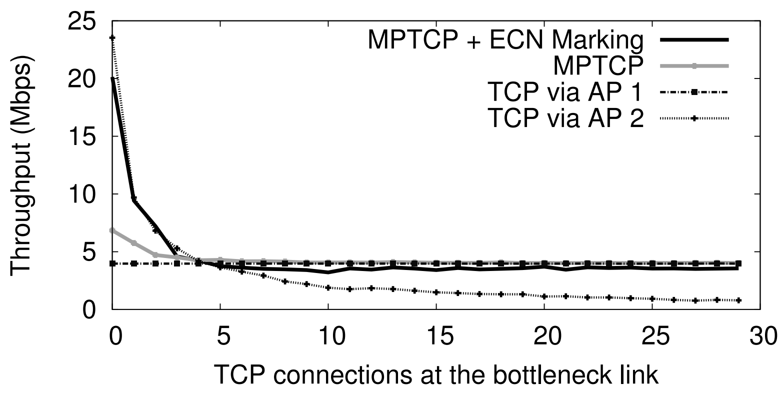


Figure 2.10: ECN Marking delivers near optimal throughput for scenario 2.

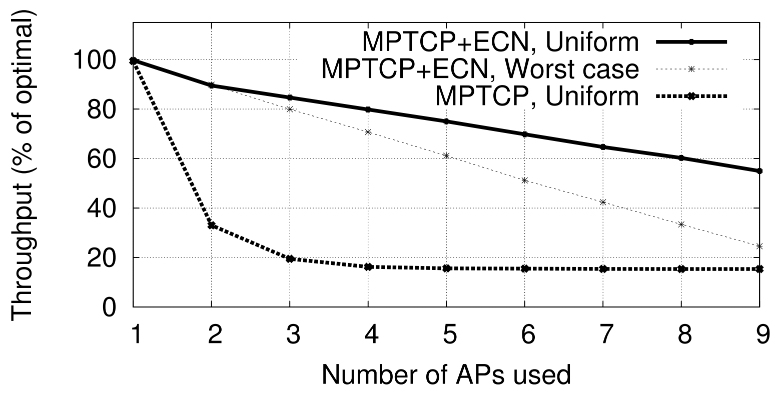


Figure 2.11: Connecting to more APs reduces the total available throughput.

that MPTCP alone fails to deliver when the AP_2 uplink is idle, but obtains the maximum possible throughput when AP_2's uplink is busy (same as TCP over AP_2). MPTCP with ECN marking gets the best of both worlds: it closely tracks the performance of a single TCP flow via AP_2 when there is little contention for AP_2's uplink, and it stabilizes to just under 4Mbps when AP_2 uplink is congested.

Increasing the number of APs. We've looked at connecting to two APs until now. What happens when there are more APs the client can connect to? We ran an experiment where the best AP offers maximum rate, and we are adding a varying number of other APs. In our first experiment, we consider a worst case where all the added APs are poor: their packet times is set to $6ms$ (2Mbps); in our second experiment we distribute the packet times of the APs uniformly between $0.5ms$ and $6ms$, mimicking a more realistic situation, and plot both results in Figure 2.11. The results show a linear drop in the throughput obtained as the number of APs increases when ECN is used, however the slope is steeper when all APs have poor performance. This graph shows that connecting to more than 3-4 APs is a bad idea: the client should choose the few best APs and connect to those alone.

2.2.2.4 A mobility experiment

We now discuss a mobility experiment run in a building with mixed WiFi coverage: the user starts from a lab on the second floor of the building, goes down a flight of stairs and then walks over to the cafeteria. En route, the mobile is locked on channel 6 and can associate with five different APs, each covering different parts of the route. We repeat the experiment several times, each taking of around one minute, during which the client is either: a) using one AP at a time, with standard TCP; b) using MPTCP and associating to all APs it sees all the time; or c) performing handover between the different APs by using MPTCP. Our client monitors the beacons received over a 2s period, and switches to a new AP when it receives Δ more beacons than the current AP. It is well known that TCP performance suffers in cases of frequent mobility [29]. The same effect happens during MPTCP handovers, when a new subflow is created and has to do slowstart after switching to a new AP. In-between APs the client may flip-flop between APs based on its observed beacon

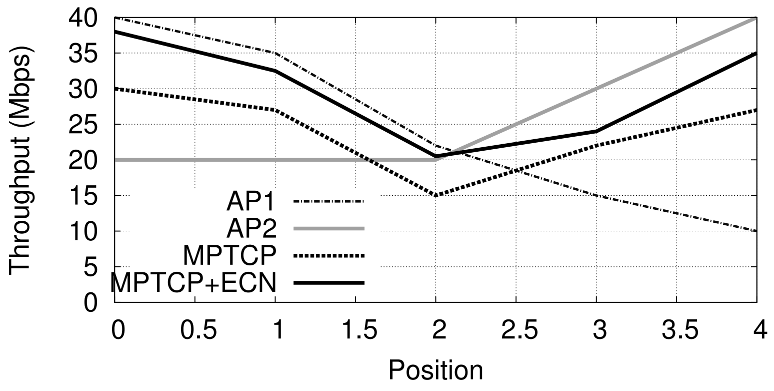


Figure 2.12: Throughput of a nomadic client at different position between AP1 and AP2 in 802.11n. MPTCP with ECN marking provides 85% of the optimal throughput.

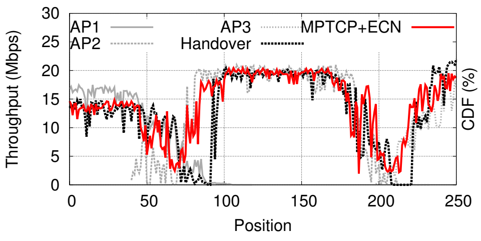


Figure 2.13: Mobility experiment: indoor client walking speed.

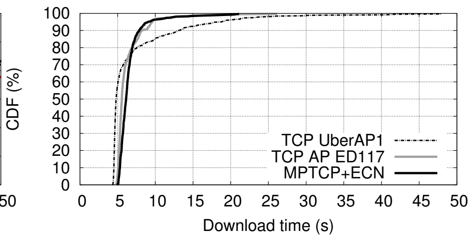


Figure 2.14: Static clients experience performance variations outside their control; MPTCP offers predictable performance

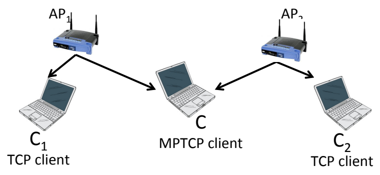


Figure 2.15: Experimental setup to test fairness

count, reducing overall performance. To avoid this effect, we experimentally chose $\Delta = 10$.

In another experiment, the client slowly moves through the building at 1km/h, and the results are shown in Figure 2.13. At the beginning of the walk, the client has access to two Linux APs running minstrel, and these are also accessible briefly on the stairs, in the middle of the trace. The departments' deployment of uncoordinated APs (Fortinet FAP 220B) are available both in the 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 uses a $break - before - make$ strategy and 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.

2.2.2.5 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 2.14. 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.

Power consumption 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 [13]. The energy cost of our solution is therefore dependent on the actual throughput achieved, which is near-optimal.

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

2.2.2.6 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 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 (WiFi is a completely distributed protocol and also inherently unfair), nor maximizing aggregate throughput (which may mean some distant clients are starved). Rather, we want our solution's impact on other clients to be *no worse than that of a TCP connection using the best available AP*.

The theory of MPTCP congestion control reassures us that two or more 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 setup shown in Figure 2.15 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. 2.15 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.

2.2.3 Channel Switching results

We first test a simple switching procedure, which spends a fixed period of 100ms on each channel, regardless of the performance obtained. One optimization implemented is that in case no packets arrive for a given time (10ms), the channel may be switched before the actual 100ms pass. For this purpose, we set up in a room two APs with reduced power at a distance of 20m, so that when a client is next to one AP, the performance towards the other AP at the end of the room is much lower. In Figure 2.16, the red and green lines show the behaviour with standard TCP when the client is connected to each AP. When close, the mobile achieves 17-20Mbps, and when far only 4-6Mbps. Intermediate positions are all within LOS of the two APs, with relatively even spacing. All measurements are taken while the user is static at the designated location. MPTCP with two flows across each AP is able to obtain an weighted average of the two feasible capacities, but in the middle region is able to exploit the channel diversity to harvest the better portions of each channel.

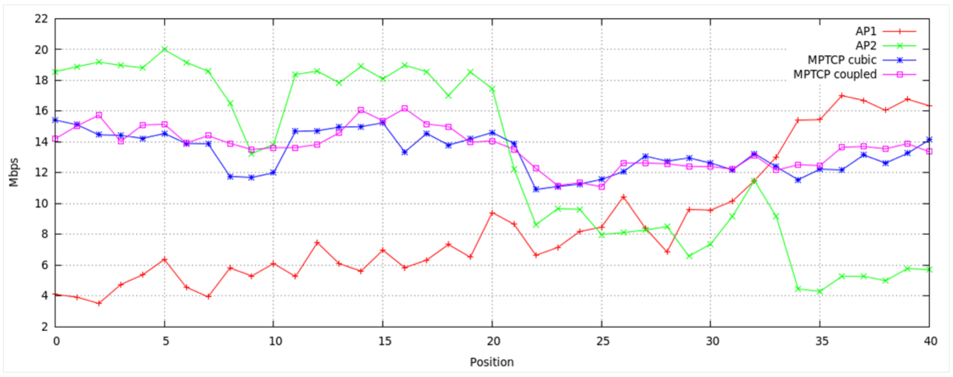


Figure 2.16: Switching for a fixed period of 100ms between channels 1 and 6. MPTCP exploits channel diversity and achieves better throughput than the best when the quality is average.

2.2.3.1 Channel switching with many users

The one key difference between the single channel and multi channel scenarios is the behavior when multiple users are connected to the same APs. When on the same channel, users tend to stick to the AP closest to them as our experiments showed in Deliverable D3.2. When switching, the clients are not coordinated and will affect each other's throughput, depending on how their slots overlap. Intuitively, when multiple clients make independent switching decisions we expect the overall channel utilization to be suboptimal. We resort to simulation to understand these effects. We model a number of mobile clients connected to three APs on three distinct channels, and all clients can download from every AP at 22Mbps (i.e., the 802.11g perfect channels). The optimum is for the clients' speeds to sum up to 66Mbps. With channel switching, however, clients' slots will overlap and some channels will be idle while others may have two clients using them simultaneously. Our simulator uses a simplified model that assumes no channel switch overheads, and that bandwidth is shared equally amongst all clients using a channel. When computing slot times, we also add a number of ms chosen uniformly at random between 1-10ms, to model for random delays experienced by channel switching code due to interactions with the AP [21]. In the table below we plot the average throughput obtained as a percentage of the optimum.

Users	1	2	3	4	5	7	10
$\alpha = 2$	100	80	64	70	76	81	87

The results show worst results when three users contend for three channels and a third of the capacity is lost; if fewer or a lot more users share the channels, the effects are less pronounced. Note that these results also hold for the single channel setting, when the AP backhaul is the bottleneck (e.g., DSL scenarios). These are inherent issues with distributed AP switching and solving them without a centralized controller or changes to APs and MAC protocols is very difficult.

2.3 Observing Multipath TCP on smartphones

Multipath TCP [12] is one of the tools that enables liquidity on mobile devices such as smartphones. Despite the huge interest for this use case, little is known about the operation of Multipath TCP on smartphones. Apple uses Multipath TCP on all their smartphones, but only to support the Siri voice recognition application. This application only exchanges data with servers operated by Apple and no measurement study has analysed how Multipath TCP behaves on smartphones. We fill this gap in this section by providing the results of two measurement campaigns to understand how smartphones interact with Multipath TCP.

We first automate a dozen of popular Android applications and develop various usage scenarios when running them over Multipath TCP. Sections 2.3.1 and 2.3.2 describe our measurement methodology and provide lab results. Then, we deploy Multipath TCP on a dozen of smartphones and collect a six weeks trace of all their Multipath TCP traffic. This dataset is described in Section 2.3.3. The collected packet trace is discussed in more details in Section 2.3.4 while Section 2.3.5 analyses the main results extracted from this dataset.

2.3.1 Automating smartphone applications

Various researchers have analyzed the performance of Multipath TCP through measurements. Raiciu et al. [40] discuss how Multipath TCP can be used to support mobile devices and provide early measurement

results. Chen et al. [4] analyze the performance of Multipath TCP in WiFi/cellular networks by using bulk transfer applications running on laptops. Ferlin et al. [11] analyze how Multipath TCP reacts to bufferbloat and propose a mitigation technique. As of this writing, this mitigation technique has not been included in the Linux Multipath TCP implementation. Livadariu et al. explore the performance of Multipath TCP on dual-stack hosts in [25] and show that performance over IPv6 and IPv4 paths differ. Ferlin et al. [10] propose a probing technique to detect low performing paths and evaluates it in wireless networks. Lim et al. propose in [24] a technique to reduce the energy consumption of smartphones that are using Multipath TCP and evaluates it experimentally. Williams et al. analyze in [48] the performance of Multipath TCP on moving vehicles. Deng et al. [6] compare the performance of single-path TCP over WiFi and LTE networks with Multipath TCP on multi-homed devices by using active measurements and replaying HTTP traffic observed on mobile applications. They show that Multipath TCP provides benefits for long flows but not for short ones, for which the selection of the interface for the initial subflow is important from a performance viewpoint. Hesmans et al. [18] analyze a one week-long server trace supporting Multipath TCP.
Several backports of the Multipath TCP kernel on Android smartphones were released in the last years. However, these ports were often based on old versions of the Multipath TCP kernel. For this work, we rely on a backport of the latest version 0.89v5 of the Multipath TCP Linux kernel[1] on a Nexus 5 running Android 4.4.4. It should be noted that the Linux kernel used on such Android devices is tweaked to use only one interface at a time. When the default interface changes, the Android 4.4.4 kernel executes the `tcp_nuke_addr` to kill all the TCP connections that use this interface. We disable this function and configure the kernel to be able to simultaneously use two interfaces. The Multipath TCP kernel controls the utilization of the available interfaces thanks to a path manager. We use the Full Mesh path manager that creates a subflow over all network interfaces for each established TCP connection. To spread packets over the available paths, we use the default RTT-based scheduler [36] which sends packets over the available path with the lowest Round-Trip-Time.
Most popular smartphone applications use TCP to interact with servers managed by the application developers. As of this writing, it has not been possible to convince them to install Multipath TCP on their servers. To overcome this issue, we configure the smartphone to use a Multipath TCP capable SOCKS proxy server for all its connections as shown in Figure 2.17. This is exactly the same setup as the one that was launched commercially in Korea in June 2015 [44]. Each (Multipath) TCP connection initiated by the smartphone is thus redirected to, and terminated at, the proxy server. The proxy server then establishes a regular TCP connection to the server. Thanks to this setup, the smartphone can use Multipath TCP over the cellular and WiFi interfaces while interacting with legacy servers via the proxy. The SOCKS server itself uses ShadowSocks and is configured to use the minimum encryption scheme to reduce the overhead. The other settings are set to the recommended values.[2] On the smartphone, we use the standard Android ShadowSocks client.

2.3.1.1 Automating measurements

In order to collect a large number of measurements, we developed a test framework that automates the interactions with these applications.[3] A high-level overview is shown in Figure 2.17. On this basis, we identify two main tasks: *controlling devices* and *mimicking user interaction*.

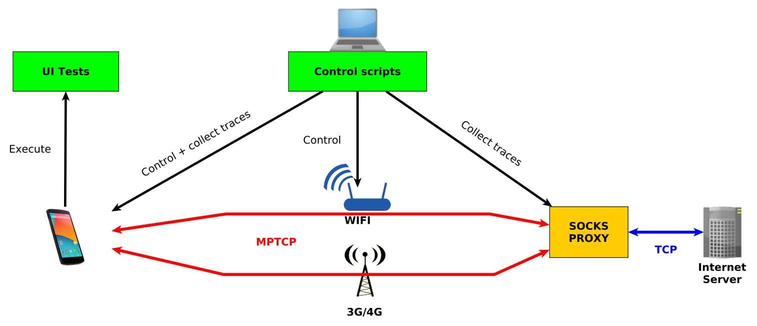


Figure 2.17: High-level view of the test framework.

[1]Available from `www.multipath-tcp.org`
[2]See `http://shadowsocks.org/en/config/advanced.html`
[3]Results are reproducible, instructions are publicly available. See `http://github.com/MPTCP-smartphone-thesis/uitests`

The devices are controlled by Python and shell scripts (3100 lines split into different modules). Our controller checks their availability of the smartphones and the wireless networks, collects packet traces and modifies settings such as the protocol (either TCP or Multipath TCP) or the interfaces (WiFi, cellular or both) used by the smartphone. It was designed to be reusable, modular using parameters and to cope with unexpected situations caused by unreliability of this kind of devices.
User interactions are simulated through application UI tests to produce each high-level scenario. Each of the eight selected applications has its own UI test. These UI tests are implemented by using the MonkeyRunner Android UI testing tool. Each unit test is implemented as a new program and all of them use a shared `Utils` class. Thanks to this class, our framework allows to build a scenario with less than a few hundred lines of code. Each test was designed to resist different unusual situations, such as the failure of the smartphone, the failure of one of the wireless networks or unexpected reaction of the application. The measurements presented in this section were performed with the versions of the applications released on November 15th, 2014. To avoid network optimization and have repeatable measures, cached files are deleted when launching our tests.
All the tests described in this document were performed during the night to reduce interferences with other users on the networks. The WiFi network was provided by a controlled router with an `802.11n` interface on the 5 GHz frequency band with a bit rate of 65 to 72 Mbps. The router was connected with a 100 Mbps link to the university network. We ensured that no other WiFi network was emitting in this frequency band in the building. The cellular network is a commercial one and we configured either 3G or 4G on the smartphone. The test scenarios were run in a random order each day to limit correlation of the results with the time at which they were launched.

2.3.1.2 Test scenarios

Now, we provide an overview of the scenarios used to generate network traffic. Our test scenarios can be split into two categories: *upload intensive* scenarios and *download intensive* scenarios. Each test takes less than 120 seconds.
We first consider two interactive applications: Facebook and Messenger. With the Facebook application, our test first updates the news feed, then writes a new status, takes and shares a new photo with a description and finally performs a new check out status. With Messenger, it sends a text message, then puts a smiley and finally sends a new photo. Then we consider two cloud storage applications: Dropbox and Google Drive. For both, we create a fresh file containing 20 MB of purely random data and upload it.
First, we use Firefox to browse the main page of the top 12 Alexa web sites with an empty cache. Our second application is Spotify. This is a music delivery application. The test plays a new music (shuffle play feature) for 75 seconds. Finally, we consider two popular video streaming applications: Dailymotion and Youtube. For both applications, we play three different videos in the same order and watch them for 25 seconds. Those videos are available in HD and we fetch the best possible quality even when using cellular networks.
We used these applications on the testbed shown in Figure 2.17. This setup allows us to capture all the packets sent by both the smartphone and the SOCKS server. We captured more than 110000 connections over about 1400 different tests conducted in February and March 2015 carrying more than 15 GBytes of data. The entire dataset is publicly available[4].

2.3.2 Measurements with automated applications

We use our test framework to analyze the interactions between smartphone applications and the network under various conditions. We first observe our applications with regular TCP, then we study how they behave over Multipath TCP. We use `tstat` [31] and `mptcptrace` [17] to extract information from packet traces.

2.3.2.1 Single-path measurements

The selected applications interact in different ways with the underlying transport protocol. An important factor that influences the performance of TCP is the lifetime of the connections and the number of bytes that are exchanged. To study this factor, we analyse the TCP connections established by our studied applications. Figure 2.18 shows that they create different types of TCP connections. Each point on this figure represents one captured TCP connection. The x-axis (in logarithmic scale) is the connection duration in seconds while the y-axis is the number of bytes exchanged on the connection. Firefox is clearly the application that uses the largest number of connections (63.9 % of all connections) which is not surprising given that our Firefox

[4]See `http://multipath-tcp.org/data/IEEEComMag16`

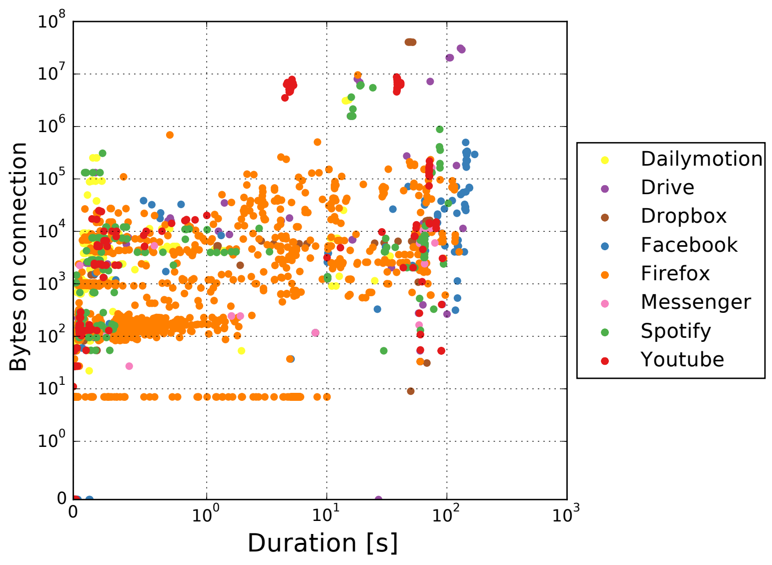


Figure 2.18: Duration and data transferred by the smartphone applications.

scenario contacts the 12 top Alexa web sites. Unsurprisingly, streaming and cloud storage scenarios with Dropbox (31.75%), Youtube (29.7%), Drive (19.9%), Dailymotion (9.6%) and Spotify (5%) are the applications that exchange the largest volume in bytes. On the other hand, our Facebook scenario generates long TCP connections that do not exchange too many bytes.

Some of the connections that we observe are caused by the utilization of a SOCKS proxy. There are hundreds connections that last up to tens of seconds but only transfer seven bytes of data. After investigation, Firefox proactively opens new TCP connections but sometimes never uses them. The seven exchanged bytes correspond to the command sent by the SOCKS client. This command contains the IPv4 address and destination port used by the SOCKS proxy to establish the regular TCP connection to the remote servers. Most of the short connections that only transfer about 100 bytes are the DNS requests that are sent over TCP by the SOCKS client.

Our connections can be categorized in 3 types : *(i)* the short connections carrying a relatively small amount of data, *(ii)* the long connections carrying most of the data, and *(iii)* long-lived connections carrying a small amount of data. In our tests, 74% of the connections last less than 1 second. Among the connections that last more than 1 second, 32% carry more than 10 KB and represent 98.6% of the overall volume. Finally, the remaining 68% of the connections that last more than 1 second exchange less than 10 KB of data. This tends to match many measurement studies which identified that most TCP connections are short and most of the traffic is carried by a small fraction of all TCP connections [15].

The Round-Trip-Time is one of the key factors that influence the performance of TCP connections. We used `tstat` to compute the average RTT for each of the captured TCP connections. Figure 2.19 provides the CDF of the RTT measures among all the TCP connections used in the upstream (data sent by the smartphone) and downstream directions. The 4G network exhibits an RTT in upstream with a median of 42.6 msec and a mean of 50 msec. In the downstream direction, the median RTT increases up to 38.1 msec. On the WiFi network, 60% of the connections have an RTT shorter than 15.4 msec. Unsurprisingly, there is some bufferbloat on the 3G network, mainly in the upstream direction, but the bufferbloat remains reasonable compared to other networks [11].

2.3.2.2 Multipath measurements

The previous section showed that our measurement scenarios cover different utilizations of TCP. We now enable Multipath TCP on our smartphone and perform the same measurements to understand how our eight applications interact with Multipath TCP. The first, but important, point to be noted is that we did not observe any incompatibility between the applications and Multipath TCP.

Multipath TCP can be used in different modes [34] on smartphones. For our measurements, we focus on a configuration where Multipath TCP tries to pool the resources of the cellular and the WiFi interfaces simultaneously since the handover and backup performance has already been studied in [34].

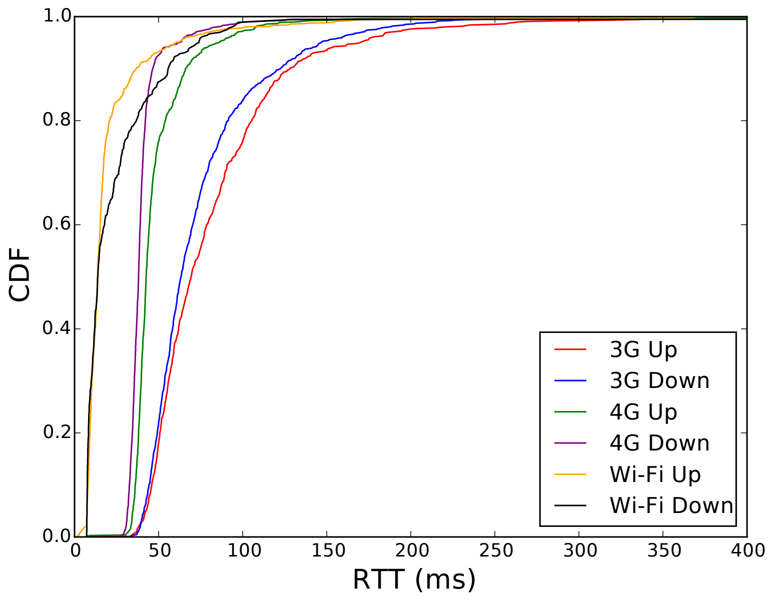


Figure 2.19: Average Round-Trip-Time of the TCP connections over the WiFi, 3G and 4G networks.

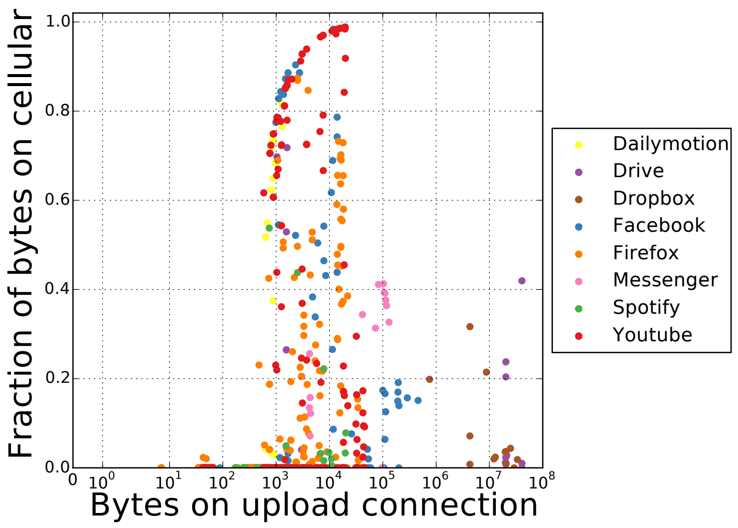


Figure 2.20: Fraction of data bytes sent on a connection depending of its data size

When a 4G and a WiFi interface are pooled together it is interesting to analyze which fraction of the traffic is sent over which interface. With the Multipath TCP implementation in the Linux kernel, this fraction depends on the interactions between the congestion control scheme, the packet scheduler, the underlying networks and the application.

We first consider Multipath TCP connections using WiFi and 4G interfaces, with WiFi set as the `default` interface. In Figure 2.20, each point corresponds to one Multipath TCP connection, and the x axis indicates the number of bytes transfered by this connection from the smartphone to servers. Although we observe connections using both WiFi and cellular interfaces, Fig. 2.21 shows that 96% of the connections only use the WiFi interface. However, Figure 2.22 indicates that those connections are small since they carry only 16.3% of all the data bytes contained in the considered connections.

Several factors explain why Multipath TCP does not use the cellular network for these short connections. The first factor is the configured `default` route. When an application initiates a connection, Multipath TCP sends the `SYN` over the interface with the `default` route, in our case the WiFi interface. This is the standard configuration of Android smartphones that prefer the WiFi interface when it is active. If the Multipath TCP connection is short and only transfers a few KiloBytes or less, then most of the data fits

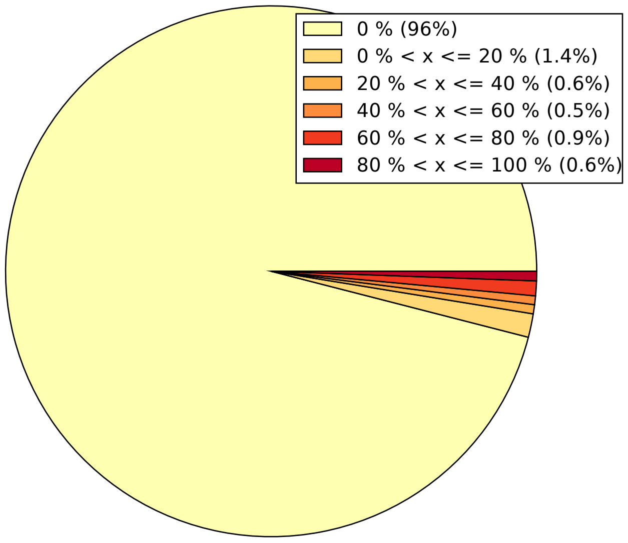


Figure 2.21: Connections classified by the percentage of data sent on cellular interface

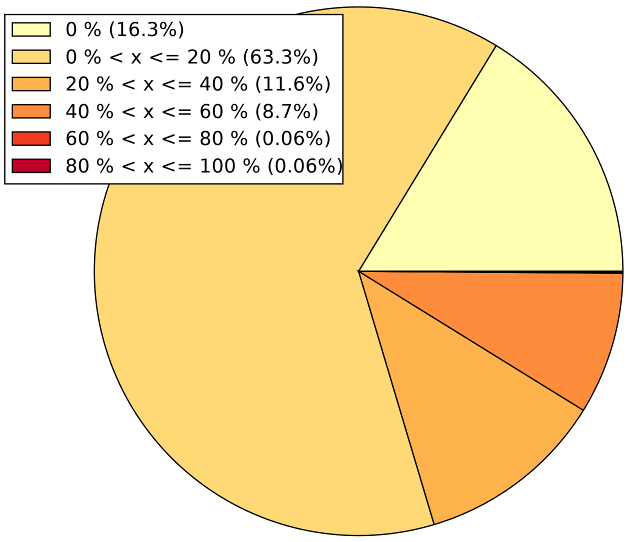


Figure 2.22: Data bytes classified by the percentage of data sent on cellular interface

inside the initial congestion window and can be sent over the WiFi interface while the second subflow is established over the cellular interface. 71% of the connections sending only on WiFi interface are in this case. Furthermore, the RTT over the WiFi interface is shorter than over the cellular interface. This implies that most of the time, as long as the congestion window is open over the WiFi interface, Multipath TCP's RTT-based scheduler [36] prefers to send packets over the WiFi interface. Indeed, 84% of the connections with both subflows established have a smaller average RTT on WiFi than on 4G.

Those factors explain why data on the short connections are exchanged only over the WiFi interface. We experimentally verified this by performing the same set of measurements with the default route pointed to the 4G interface. Figure 2.23 shows that with this configuration most short connections still use the 4G network (see label 1 on figure 2.23), but this concerns only 65% of all connections. It seems that even if cellular is the default interface, many connections still mainly use WiFi, even for connections exchanging less than 1 KB. This occurs for connections that do not push data as fast as possible. If the connection lasts more than two RTTs, Multipath TCP has enough time to establish the second subflow. The packet scheduler will then select the subflow with the lowest RTT – 88% of connections using both subflows have a WiFi subflow with a lower average RTT than the cellular one.

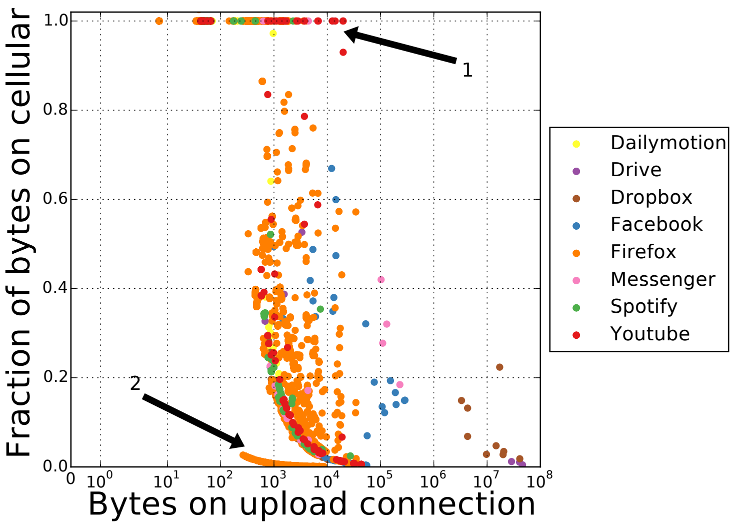


Figure 2.23: When the default route points to the cellular interface, many of the connections only use the WiFi interface.

This explains the bottom of Figure 2.23 (annotated as 2): a group of Firefox connections that transfer less than 10 KB only use the WiFi interface. A closer look at the packet trace reveals that these connections are part of the connection pool managed by Firefox. This behavior does not happen with other applications. When Firefox creates a connection in this pool, the initial handshake and the SOCKS command to our SOCKS server are sent. These packets are exchanged over the cellular interface and Firefox does not immediately send data over the established connection. This leaves enough time for Multipath TCP to create the subflow over the WiFi interface and measure its RTT. When Firefox starts transmitting data over such a connection, the RTT-based scheduler used by Multipath TCP prefers the WiFi subflow and no data (except the initial SOCKS command) is sent over the cellular subflow.

When the applications push more data over the Multipath TCP connection, the distribution of the traffic between the cellular and the WiFi interface also depends on the evolution of the congestion windows over the two subflows. If the application pushes data at a low rate, then the packet scheduler will send it over the lowest-RTT interface (WiFi in this case). However, this distribution can be fragile. If one packet is lost, then the congestion window is reduced and the next data might be sent over the other interface. If the application pushes data at a higher rate, then the congestion window over the lowest-RTT interface is not large enough and the packet scheduler will send data over the second subflow.

In some cases, data transfered by Multipath TCP on one flow may be retransmitted again on the other flow. This phenomenon is called reinjection [42] and might limit the performance of Multipath TCP in some circumstances [46]. We used `mptcptrace` to compute the reinjections over all observed Multipath TCP connections. In our experiments (WiFi and 4G), reinjections in the upstream direction were rare (less than half a percent of all connections include a reinjection) and short (no more than 5 KB are reinjected on a connection). Looking at the proxy traces in the downstream direction, reinjections are observed on only 2% of all connections, and the largest observed reinjection is 30 KB on a 5 MB connection. This overhead is thus low.

An important benefit of the resource pooling capabilities of Multipath TCP is its ability to adapt to various networking conditions. When a smartphone moves, the performance of the WiFi and cellular interfaces often vary. Previous work with bulk transfer applications has shown that Multipath TCP can adapt to heterogeneous networks having different bandwidths and delays [37]. Our measurement framework also allows exploring the performance of smartphone applications under various network conditions. As an illustration, we analyze the packet traces collected when the smartphone is uploading a file with Dropbox. We first consider a WiFi access point attached to a DSL router having 1 Mbps of upstream bandwidth and 15 Mbps of downstream bandwidth. When the smartphone is attached to both this WiFi access point and the 4G network, it sends on average 91% of the data over the 4G network. This is expected because although the WiFi has better RTT, the

congestion window of this path is quickly full and slowly empties. In that case, the Multipath TCP scheduler selects the next available subflow with lowest RTT – here the cellular interface. Since the cellular network offers a larger bandwidth, Multipath TCP can take advantage of it and thus avoids being trapped in a low performance network for big connections.
As a second test case, we consider our standard WiFi access (around 70 Mbps in both streams) and the 4G network whose bandwidth is limited down to a few hundred kilobits per second. This is the shaping enforced by our cellular network once we reach the monthly traffic volume quota. In this case, 98.8% of the bytes are sent over the WiFi interface.

2.3.3 Multipath TCP with real users

Although Multipath TCP is already used by hundreds of millions of Apple smartphones to support the Siri voice recognition application, it is difficult to collect both WiFi and cellular traces without cooperation from an ISP. Instead, a Multipath TCP capable SOCKS proxy was set up (like KT) and this analysis focuses on the Multipath TCP implementation in the Linux kernel [35]. This implementation is distributed from `http://multipath-tcp.org` and can be integrated in Android.
The dataset covers the traffic produced by a dozen Nexus 5 smartphones running Android 4.4 with a modified Linux kernel that includes Multipath TCP v0.89.5. This dataset allows to understand how real smartphone applications behave when using Multipath TCP instead of regular TCP. However, installing Multipath TCP on the smartphones is not sufficient to use it for all connections established by applications. As of this writing, there are probably only a few dozens of Multipath TCP enabled servers on the Internet and these are rarely accessed by real smartphone applications. To force these applications to use Multipath TCP, `ShadowSocks`[5] was installed on each smartphone and configured to use a SOCKS server that supports Multipath TCP for all TCP connections. Note that since `ShadowSocks` does not support IPv6, this trace only contains IPv4 packets. The smartphones thus use Multipath TCP over their WiFi and cellular interfaces to reach the SOCKS server and this server uses regular TCP to interact with the final destinations. From the server side, all the connections from the dozen smartphones appear as coming from the SOCKS server. This implies that the external (cellular or WiFi) IP address of the smartphone is not visible to the servers that it contacts. This might affect the operation of some servers that adapt their behavior (e.g. the initial congestion window) in function of the client IP address. Moreover, note that the `ShadowSocks` client sends DNS requests over TCP.
A special Android application [5] managing the utilization of the cellular and WiFi interfaces was also installed on each smartphone. Smartphones with Android 4.4 assume that only one wireless interface is active at a time. When such a smartphone switches from cellular to WiFi, it automatically resets all existing TCP connections by using Android specific functions. Our application enables the cellular and WiFi interfaces simultaneously. It also controls the routing tables and updates the policy routes that are required for Multipath TCP every time the smartphone connects to a wireless network. Thanks to this application, the modified Nexus 5 can be used by any user since it does not require any networking knowledge.
The SOCKS proxy ran `tcpdump` to collect all the packets exchanged with the smartphones. Measurements were performed in Belgium from March 8th to April 28th 2015. Over this period of 7 weeks, more than 71 millions Multipath TCP packets were collected for a total of 25.4 GBytes over 390,782 Multipath TCP connections.
To our knowledge, there is no equivalent dataset containing such a diversity of Multipath TCP packets that are publicly available for researchers [18]. The anonymized traces will be released on the CRAWDAD database. The analysis scripts are also open-sourced [5].[6]

2.3.4 Characterization of the Trace

The main characteristics of the Multipath TCP connections in the dataset are first analyzed. The destination ports of the captured packets are not sufficient to identify the application level protocol. Since the smartphone connects through a SOCKS proxy, all the packets are sent towards the destination port used by the proxy (443 to prevent middlebox interferences). We extract the real destination port from the SOCKS command sent by the `ShadowSocks` client at the beginning of each connection.
As shown in Table 2.4, most of the connections and data bytes are related to Web traffic. Since

[5]Available at `http://shadowsocks.org`.
[6]Available at `github.com/multipath-tcp/mptcp-analysis-scripts`.

Table 2.4: Statistics about destination port fetched by smartphones.

Port	# connections	% connections	Bytes	% bytes
53	106,760	27.3	17.4 MB	0.1
80	103,527	26.5	14,958 MB	58.8
443	104,142	26.7	9,260 MB	36.4
4070	570	0.1	91.7 MB	0.4
5228	10,596	2.7	27.3 MB	0.1
8009	10,762	2.8	0.97 MB	< 0.1
Others	54,425	13.9	1,079 MB	4.2

Table 2.5: The different (sub)traces analyzed in this section.

Name	Description	# connections	Bytes to proxy	Bytes from proxy
$\mathcal{T}_0$	All traces	390,782	652 MB	24,782 MB
$\mathcal{T}_1$	At least 2 established subflows	126,040	238 MB	13,491 MB
$\mathcal{T}_2$	At least 2 used subflows	32,926	152 MB	11,852 MB
$\mathcal{T}_3$	With handover	7,553	31.9 MB	4,062 MB

`ShadowSocks` sends DNS requests over TCP, it is expected to have a large fraction of the connections using port 53. Among other popular port numbers, there are ports 4070 – e.g., used by Spotify –, Google Services (5228) and Google Chromecast (8009). 65% of the observed connections last less than 10 seconds. In particular, 4.3% are failed connections, i.e., the first SYN was received and answered by the proxy, but the third ACK was lost (or a RST occurred). 20.8% of the connections last more than 100 seconds. Six of them last for more than one entire day (up to nearly two days).
Looking at the bytes carried by each connection, most (86.9%) of them carry less than 10 KBytes. In particular, 3.1% of the connections carry between 9 and 11 bytes. Actually, those are empty connections, since the SOCKS commands are 7 bytes long, two bytes are consumed by the SYNs and the use of the remaining two bytes depend on how the connections were closed (RST or FIN). The longest connection in terms of bytes transported around 450 MBytes and was spread over five subflows.

2.3.5 Analysis

In the following, the analysis will focus on relevant subsets of the trace such as connections with at least two subflows, connections using at least two subflows or connections experiencing handover (see Table 2.5). We use them to analyse how Multipath TCP subflows are created (Section 2.3.5.1), study the heterogeneity of the available networks in terms of round-trip-times (Section 2.3.5.2), estimate the packet reordering of Multipath TCP (Section 2.3.5.3), study how subflows are used (Section 2.3.5.4), quantify the reinjection overhead (Section 2.3.5.5) and identify connections experiencing handovers (Section 2.3.5.6).

2.3.5.1 Establishment of the subflows

With Multipath TCP, a smartphone can send data over various paths. The number of subflows that a smartphone creates depends on the number of active interfaces that it has and on the availability of the wireless networks.
Table 2.6 reports the number of (not necessarily concurrent) subflows that are observed in $\mathcal{T}_0$. Most of the connections only have one subflow. Conversely, 2.29% of the connections have more than two subflows.

Table 2.6: Number of subflows per Multipath TCP connection.

Number of subflows	1	2	3	4	5	>5
Percentage of connections	67.75%	29.96%	1.07%	0.48%	0.26%	0.48%

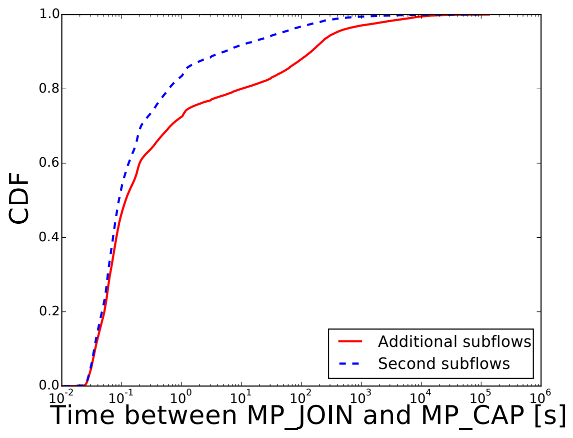


Figure 2.24: Delay between the creation of the Multipath TCP connection and the establishment of a subflow.

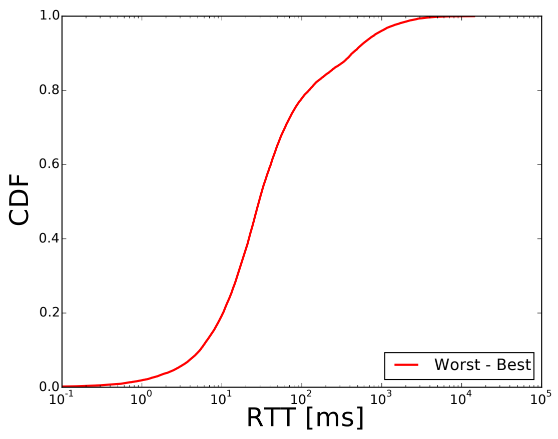


Figure 2.25: Difference of average RTT seen by the proxy between the worst and the best subflows with at least 3 RTT samples.

Having more subflows than the number of network interfaces is a sign of mobility over different WiFi and/or cellular access points since IPv6 was not used. We observed a connection that used 42 different subflows.

Another interesting point is the delay between the establishment of the connection (i.e., the first subflow) and the establishment of the other subflows. The smartphone tries to create subflows shortly after the creation of the Multipath TCP connection and as soon as a new interface gets an IP address. Late joins can be expected mainly when a smartphone moves from one network access point to another. To quantify this effect, Figure 2.24 plots the CDF of the delays between the creation of each Multipath TCP connection and all the additional subflows that are linked to it. 60% of all the additional subflows are established within 200 ms. This percentage increases to 72.5% if this limit is set to one second. If we restrict the analysis to the first additional subflow, these percentages are respectively 67.3% and 81.1%. Joins can occur much after the connection is established. Indeed, 14.5% of the additional subflows were established one minute after the establishment of the connection, and 1.5% of them were added one hour later. The maximal observed delay is 134,563 seconds (more than 37 hours) and this connection was related to the Google Services. Those late joins suggests network handovers, and late second subflow establishments can be explained by smartphones having one network interface unavailable.

2.3.5.2 Subflows round-trip-times

From now, we focus on the subtrace $\mathcal{T}_1$ that includes all the connections with at least two subflows. A subflow is established through a three-way handshake like a TCP connection. Thanks to this exchange, the communicating hosts agree on the sequence numbers, TCP options and also measure the initial value of the round-trip-time for the subflow. For the Linux implementation of Multipath TCP that we used, the round-trip-time measurement is an important performance metric because the default packet scheduler prefers the subflows having the lowest round-trip-times.

To evaluate the round-trip-time heterogeneity of the Multipath TCP connections, the analysis uses `tstat` [31] to compute the average round-trip-time over all the subflows that a connection contains. Then, it extracts for each connection the minimum and the maximum of these average round-trip-times. To have consistent values, it only takes into account the subflows having at least 3 RTT estimation samples. Figure 2.25 plots the CDF of the difference in the average RTT between the subflows having the largest and the smallest RTTs over all connections in $\mathcal{T}_1$. Only 19.4% of the connections are composed of subflows whose round-trip-times are within 10 ms or less whereas 77.9% have RTTs within 100 ms or less. 3.9 % of the connections experience subflows having 1 seconds or more of difference in their average RTT. With such network heterogeneity, if a packet is sent on a low-bandwidth and high-delay subflow s_0 and following packets are sent on another high-bandwidth low-delay one s_1, the sender may encounter head-of-line blocking.

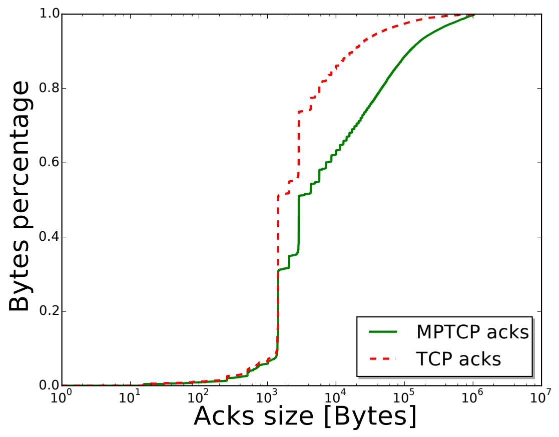


Figure 2.26: Size of the Multipath TCP and TCP ACKs received by the proxy.

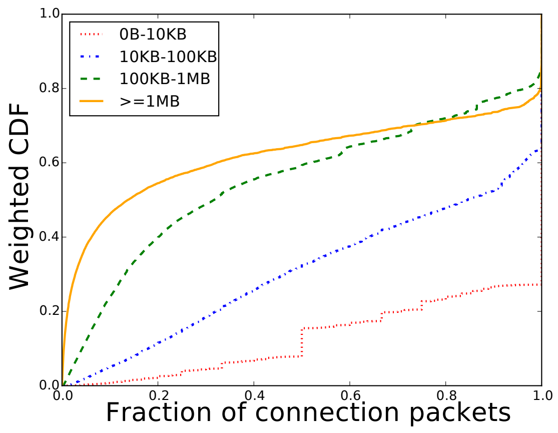


Figure 2.27: Size of the subflow blocks from proxy to smartphones on connections having at least two established subflows ($\mathcal{T}_1$).

2.3.5.3 Multipath TCP acknowledgements

Multipath TCP uses two ACK levels: the regular TCP ACKs at the subflow level and the cumulative Multipath TCP ACKs at the connection level. It is possible to have some data acknowledged at TCP level but not at Multipath TCP one, typically if previous data was sent on another subflow but not yet acknowledged. Fig. 2.26 plots in red-dotted curve the CDF of the number of bytes sent by the proxy that are acknowledged by non-duplicate TCP ACKs. This plot is a weighted CDF where the contribution of each ACK is weighted by the number of bytes that it acknowledges. In TCP, ACKs of 1428 bytes or less cover 50.7% of all acknowledged bytes and considering ACKs of 20 KB or less the percentage is 91.1%.

The same analysis is now performed by looking at the DSS option that carries the Multipath TCP Data ACKs with `mptcptrace` [17]. The green curve in Fig. 2.26 shows the weighted cumulative distribution of the number of bytes acked per Data ACK. Compared with the regular TCP ACKs, the Multipath TCP ACKs cover more bytes. Indeed, 51% of all bytes acknowledged by Multipath TCP are covered with Data ACKs of 2856 bytes or less, and this percentage increases to 70.6% considering Data ACKs of 20 KB or less.

The difference between the regular TCP ACKs and the Data ACKs is caused by the reordering that occurs when data is sent over different subflows. Since the Data ACKs are cumulative they can only be updated once all the previous data have been received on all subflows. If subflows with very different round-trip-times are used, it will cause reordering and data will possibly filling the receiver's window during a long period. This can also change the way applications read data which would be more by large bursts instead of small frequent reads. On mobile devices, such memory footprints should be minimised.

2.3.5.4 Utilization of the subflows

The next question is how data is spread among the subflows. Does Multipath TCP alternate packets between the different subflows or does it send bursts of packets? Again, to be relevant, the subtrace $\mathcal{T}_1$ is considered.

To quantify the spread of data, we introduce the notion of *subflow block*. Intuitively, a *subflow block* is a sequence of packets from a connection, all of which are sent over a given subflow but the next packet is transmitted over a different subflow. Consider a connection where a host sends N data packets. Number them as $0, ..., N-1$ with 0 the first data packet sent and $N-1$ the last one. Let f_i denote the subflow on which packet i was sent. The n^{th} subflow blocks b_n is defined as $b_n = \{\max(b_{n-1}) + 1\} \cup \{i \mid i-1 \in b_n \text{ and } f_i = f_{i-1}\}$, with $b_0 = \{-1\}$ and $f_{-1} = \perp$. As an example, if the proxy sends two data packets on s_0, then three on s_1, retransmits the second packet on s_0 and sends the last two packets on s_1, we will have $b_1 = \{0, 1\}$, $b_2 = \{2, 3, 4\}$, $b_3 = \{5\}$ and $b_4 = \{6, 7\}$. A connection balancing the traffic with several subflows will produce a lot of small subflow blocks whereas a connection sending all its data over one single subflow will have only one large subflow block containing all the connection's packets. Figure 2.27 shows the repartition of the fraction of data packets that each subflow block carries. This CDF is weighted over connections such that each connection has the same impact on the graph, independently of their number of subflow blocks. Each curve contains connections carrying their labeled amount of bytes. For most of the large connections, Multipath TCP balances well the packets over different subflows. On average connections carrying more than 1 MB have 54.5% of all their data packets sent on subflow blocks carrying 20% or less of the connection packets. As expected, the shorter the connection is, more the subflow blocks tend to contain most of the connection traffic. For short connections carrying less than 10 KBytes, 72.8% of them contain only one subflow block, and therefore they only use one subflow. This number raises concerns about unused subflows. If connections having at least two subflows are considered, over their 276,133 subflows, 41.2% of them are unused in both directions. It is worth noting that nearly all of these unused subflows are actually additional subflows, leading to 75.6% of unused additional subflows. This is clearly an overhead, since creating subflows that are not used consumes bytes and energy [38] on smartphones since the interface over which these subflows are established is kept active.

There are three reasons that explain those unused subflows. Firstly, a subflow can become active after all the data has been exchanged. This happens frequently since 62.9% of the connections carry less than 2000 bytes of data. In practice, for 21% of the unused additional subflows the proxy received their third ACK after that it had finished sending data. Secondly, as suggested in Section 2.3.5.2, the difference in round-trip-times between the two available subflows can be so large that the subflow with the highest RTT is never selected by the packet scheduler. If the server does not transmit too much data, the congestion window on the lowest-RTT subflow remains open and the second subflow is not used. Though, 36.2% of the unused additional subflows have a better RTT for the newly-established subflow than the other available one. However, 59.9% of these subflows belong to connections carrying less than 1000 bytes (90.1% less than 10 KBytes). Thirdly, a subflow can be established as a backup subflow [12]. Indeed, a user can set the cellular subflow as a backup one, e.g., for cost purpose. 2.1% of the unused additional subflows were backup subflows.

2.3.5.5 Reinjections and retransmissions

In addition to unused subflows, another Multipath TCP specific overhead is reinjections. A *reinjection* [42] is the transmission of the same data over two or more subflows. Since by definition, reinjections can only occur on connections that use at least two subflows, this analysis considers the subtrace $\mathcal{T}_2$. A reinjection can be detected by looking at the Multipath TCP Data Sequence Number (DSN). If a packet A with DSN x is sent first on the subflow 1 and after another packet B with the same DSN x is sent on the subflow 2, then B is a reinjection of A. A reinjection can occur for several reasons: *(i)* handover, *(ii)* excessive losses over one subflow or *(iii)* the utilisation of the Opportunistic Retransmission and Penalization (ORP) [42, 37] algorithm. This phenomenon has been shown to limit the performance of Multipath TCP in some wireless networks [46].

Typically, Multipath TCP reinjections are closely coupled with regular TCP retransmissions. Figure 2.28 shows the CDF of the reinjections and retransmissions sent by the proxy. The number of retransmitted and reinjected bytes are normalized with the number of unique bytes sent by the proxy over each connection. 52.7% of the connections using at least two subflows experience retransmissions on one of their subflows whereas reinjections occur on 29.3% of them. For retransmissions, this percentage tends to match previous analysis of TCP on smartphones [9, 19]. 68.7% of $\mathcal{T}_2$ connections have less than 1% of their unique bytes retransmitted, and 85% less than 10%. 79.7% of the connections have less than 1% of their unique bytes reinjected, and 89.8% less than 10%. Observing more retransmissions than reinjections is expected since retransmissions can trigger reinjections. In the studied trace, the impact of reinjections remains limited since over more than 11.8 GBytes of unique data sent by proxy, there are only 86.8 MB of retransmissions and 65 MB of reinjections. On some small connections, we observe more retransmitted and reinjected bytes than the unique bytes. This is because all the data sent over the connection was retransmitted several times. On Figure 2.28 the thousand of connections having a fraction of retransmitted bytes over unique bytes greater or equal to 1 carried fewer than 10 KB of unique data, and 83.3% of them fewer than 1 KB. Concerning the reinjections, the few hundred of connections in such case carried less than 14 KB, 63.4% of them carried less than 1 KB and 76.1% of them less than 1428 bytes.

2.3.5.6 Handovers

One of the main benefits of Multipath TCP is that it supports seamless handovers which enables mobility scenarios [12, 34]. A handover is here defined as a recovery of a failed subflow by another one. A naive solution is to rely on `REMOVE_ADDR`s to detect handover. However, 17.9% of the connections experiencing

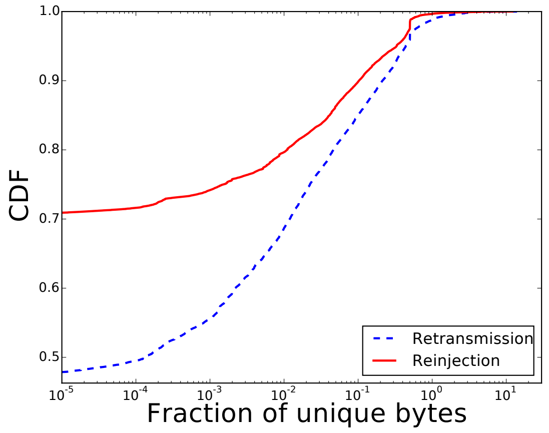


Figure 2.28: Fraction of bytes that are reinjected/retransmitted by the proxy on connections having at least two used subflows ($\mathcal{T}_2$).

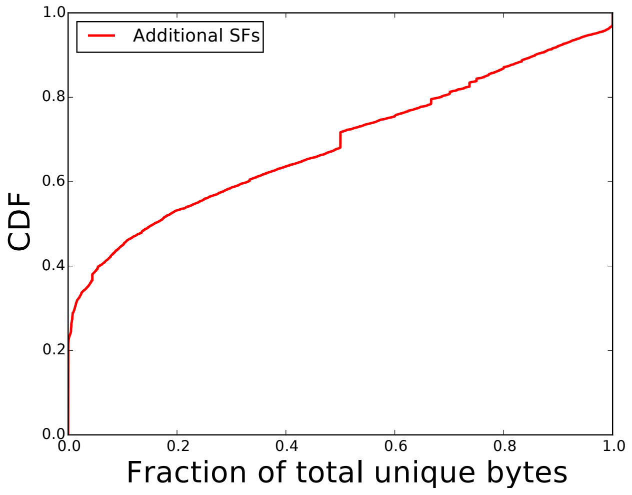


Figure 2.29: Fraction of total data bytes on non-initial subflows sent by the proxy on connections with handover ($\mathcal{T}_3$).

handover have no REMOVE_ADDR.
This document proposes an alternative methodology that relies on the TCP segments exchanged. Let LA_i the time of the last (non-RST) ACK sent by the smartphone seen on subflow i (that was used to send data) and LP_j the time of the last (non-retransmitted) segment containing data on subflow j. If $\exists\ k, l \mid k \neq l$, no FIN seen from the smartphone on subflow k, $LA_l > LA_k$ and $LP_l > LA_k$, then the connection experiences handover. Notice that only handovers on the subflows carrying data are detected. Among the connections that use at least two subflows, 22.9% experience handover. It has also the advantage to be implementation independent since it does not use the ADD_ADDRs or REMOVE_ADDRs options that are not supported by all implementations [7].
Based on the subtrace $\mathcal{T}_3$, Figure 2.29 shows the fraction of unique bytes that were sent by the proxy on the additional subflows on connections experiencing handover. This illustrates the connections that could not be possible if regular TCP was used on these mobile devices. Indeed, a handover is typically related to the mobility of the user who can go out of the reachability of a network. Notice that this methodology can also detect handovers in the smartphone to proxy flow. Indeed, 21.3% of connections experience handovers with all data sent by the proxy on the initial subflow because the smartphone sent data on another subflow after having lost the initial one.

3 Liquidity in Operator Infrastructure

3.1 ClickOS: NFV Liquidity

3.1.1 Introduction and Implementation

As described in previous deliverables, ClickOS consists of a unikernel specifically targeted at network processing and is thus an obvious candidate for efficient NFV deployments in operator networks. With respect to previous work, we have now carried an important number of optimizations to the Xen subsystem that allow us to create liquidity: fast migration to allow ClickOS instances to seamlessly be moved from server to server, massive consolidation to allow up to 10K VMs to *concurrently* run on a single server, and support for single-board PCs, allowing us to reach even edge network deployments, among others. The rest of this section describes our prototype's implementation as well as extensive evaluation results.

As mentioned, we base our implementation on the Xen hypervisor, and the guests (i.e., the virtual machines) on MiniOS, a minimalistic, para-virtualized operating system distributed with the Xen sources. As future work, we are looking to expand this to include KVM and other operating systems for the guests such as OSv [23] and stripped-down versions of Linux. We will further look at carrying out optimizations for containers.

Our work continues along the trend of using specialized, minimalistic VMs (e.g, Mirage[27], ClickOS [30], Erlang on Xen [8], OSv [23], etc.), but takes it further, seeking to optimize the number of VMs that can be concurrently run, how fast they can be instantiated and torn down, how fast they can be migrated, their memory footprint, which platforms they can run on, and the throughput that can be handled. We are further working towards comparing these against containers, and provide some early results of this in this section.

3.1.1.1 Massive Consolidation

Our first and biggest contribution is towards *massive consolidation*: the ability to concurrently run large numbers of VMs (potentially 10,000 or more) on a single, inexpensive commodity server. Out of the box, when we started this work with Xen 4.2, we were limited to only about 300 guests at most due to Linux not being configured to provide enough file descriptors in order for Xen to provide console access to the VMs. While this issue is easily fixed, a number of others existed, so we carried out three major changes and a number of minor modifications in order to push the number of concurrent guests.

The first major change was to the XenStore, a `proc`-like back-end used to keep information about running VMs (e.g., their names and IDs, their virtual mac addresses, etc.). Basic operations such as creation, destruction and migration of VMs need to read and write entries to it, and so their performance is tightly linked to that of the XenStore. To improve it, we have written a streamlined version from scratch we call `lixs` (LIghtweight XenStore); `lixs` is written in c++, consists of about 2500 LoC, and has a pluggable system allowing us to use different storage back-ends (e.g., from a full database to a simple map).

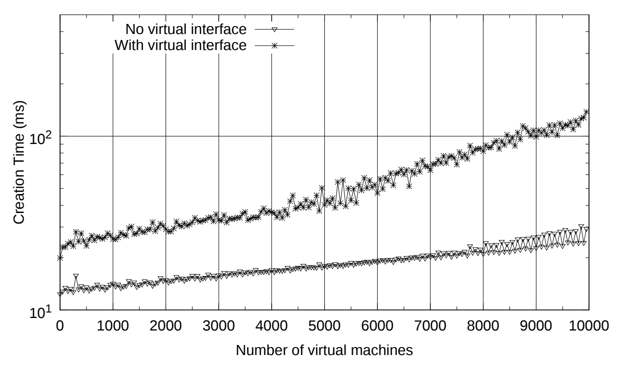
The second major change is to `xl`, the main Xen management command and toolstack used to carry out operations such as VM creation and console access. We partially replace `xl` by `xcl` (XenCtrl Light), a simplified toolstack comprising 600 LoC and tailored to our purposes (e.g., it only supports para-virtualized and PVH modes, and VIF virtual interfaces). Among other things, it reduces the number of required per-guest XenStore entries from 37 to just 17.

The third change is to Xenconsoled, the daemon in charge of providing users with console access to the VMs. Out of the box, the daemon has high CPU overhead during high rates of VM creation (practically 100% for the core assigned to it). To address this, we modify it to use the `epoll` mechanism which scales better to large numbers of watched file descriptors, and we optimize the VM creation process by, for instance, preventing Xenconsoled from listing all existing domains every time a domain is created or destroyed.

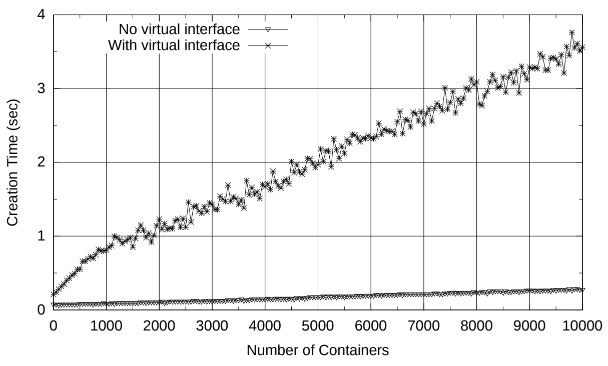
Finally, we carried out a set of minor changes such as (1) increasing Linux's maximum number of PTYs, file descriptors and IRQs; (2) upgrading to Xen 4.4 to take advantage of higher numbers of available event channels (virtual interrupts); (3) using a ramfs for dom0's and the guests' root filesystems; and (4) replacing the hotplug script in charge of setting up virtual interfaces and adding them to the back-end switch with a faster, compiled, purpose-built, udevd-like daemon.

3.1.1.2 Fast Service Instantiation

Most of the modifications we implemented for massive consolidation apply to this category, and have the effect of reducing VM creation and destruction times.



(a) Minimalistic Xen VMs (log scale, in milliseconds).



(b) LXC containers (linear scale, in seconds).

Figure 3.1: Comparison of boot storm times for Xen-based minimalistic virtual machines versus LXC containers. Each point denotes the time it took for the nth VM or container to start up.

3.1.1.3 Fast Migration

In Xen, performing migration consists of the `xl` toolstack connecting a file descriptor to the stdin/sdtout of an ssh process in order to pipe the VM image out to the receiving host. Benchmarking reveals that the overall migration time for a minimalistic VM is about 400 milliseconds, most of which is spent on copying data over the ssh pipe and only a small portion of which on write system calls.

We optimized this process by implementing a daemon that is deployed on the receiver. The daemon allocates a socket which the toolstack uses to write domain pages to; this is then used in the low-level Xen APIs so that the hypervisor can directly receive the pages. After all pages have been received, the receive daemon gets a final notification to bring up the (restored) domain, completing the process.

As future work, we are looking into optimizing live migration in order to reduce actual downtime, as well as investigating potential bottlenecks in other virtualization technologies such as KVM. We are further planning on evaluating these mechanisms when VMs are carrying out memory-intensive processing.

3.1.1.4 Microserver Support

In order to see the feasibility of using microservers as platforms for the superfluid cloud, we attempt to run Xen, KVM and/or containers (`lxc`) on a number of them, ranging from ARM-based CubieTrucks and Raspberry Pis to x86-based Intel NUCs, Intel Edisons and AMD Gizmos. This is still ongoing work, and we are planning on conducting extensive testing to see, among other things, what sort of processing and how many concurrent VMs these microservers can handle.

3.1.1.5 High Performance and Throughput

We rely on the performance numbers obtained through the optimized network back-end introduced in [30]. Beyond this, we are in the process of implementing *persistent grants*, a mechanism that provides significant speed-ups without needing changes to the VMs nor major changes to the network back-end. In addition, the work in [30] had at most 100 concurrent VMs running; scaling the back-end, and in particular the software switch, to larger numbers is ongoing research.

3.1.2 Evaluation

Unless otherwise stated, all experiments were performed on a system with four AMD Opteron 6376@2.3GHz CPUs (64 cores total) and 128GB of memory, costing about $4,000 and running Xen 4.4 and Linux 3.14 for dom0. The virtual machines are based on MiniOS and include basic functionality that allows them to respond to pings; this is one of the ways we use to verify that they are correctly instantiated. In future work we will carry tests with more realistic workloads.

As a first test, we conduct a *boot storm*: we attempt to create up to 10,000 virtual machines on our server as quickly as possible, and for each VM we measure the time it took for it to be created. For comparison purposes, we carry out the same test using LXC containers (kernel version 3.16.3 and 256KB of memory assigned to each container) instead of Xen/VMs.

The results for our minimalistic Xen VMs when using all of the optimizations described in the previous

section are shown in Figure 3.1(a). We are able to run as many as 10K concurrent virtual machines, with creation times ranging from about 20 milliseconds with a virtual interface and 12 msecs without for the first VM, up to a still rather low 135 msecs with a virtual interface and 30 msecs without one for the 10,000th VM. While these are still rather early results, they are, to the best of our knowledge, the first time this large number of VMs have been concurrently run on commodity hardware.
Figure 3.1(b) depicts results for LXC containers. We measure a container creation time of about 210 milliseconds with an interface and 70 msecs without one for the first container; and creation times of 3.5 seconds with an interface and 270 msecs without one for the 10,000th container. This shows that minimalistic VMs are a viable alternative to containers whenever strong isolation is a requirement.

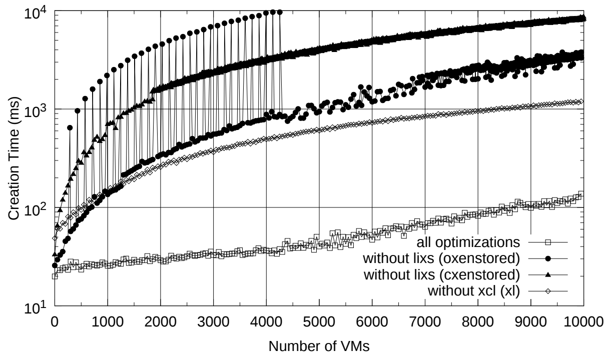


Figure 3.2: Breakdown of Xenstore and toolstack optimizations (log scale).

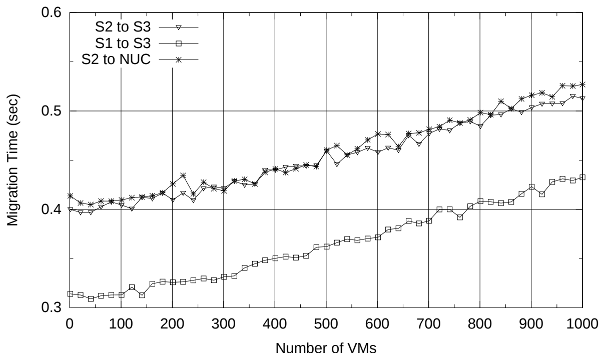


Figure 3.3: Migration time between servers and between a server and a microserver.

Figure 3.2 shows the effect that using our improved XenStore (`lixs`) and toolstack (`xcl`) have over using the standard `oxenstored` [14] and toolstack (`xl`) when carrying out a boot storm. The bottom line in the graph is there as reference and shows the same (best) results as Figure 3.1, that is, when all optimizations are enabled. The curve above it reports figures when we switch from `xcl` to the standard `xl`; the next curve are results when using `oxenstored` instead of `lixs`; and the final curve reports figures when relying on the older, C-based `cxenstored`.
Overall, these optimizations bring tangible results. For instance, for the 10,000th VM, using `oxenstore` yields boot times in the range of 2.9 seconds and 1.9 seconds with `xl`, compared to about 135 milliseconds with all optimizations turned on. It is worth noting that the spikes on the `oxenstored` curve might be due to the OCaml garbage collector kicking in; we are currently looking into the issue to confirm this.
To test our migration optimizations, we conducted a test whereby we create an increasing number of VMs on a server and migrate each of them to another server in turn. For each measurement, the receiving server has no VMs running[1] so that we measure the performance of the sending host.
Figure 3.3 shows migration times between servers and between a server and a microserver. The server labeled S1 has 2x Intel Xeon E5-2697 v2 2.70GHz CPUs (24 cores total), S2 has 4x AMD Opteron 6376@2.3GHz CPUs (64 cores total), S3 an Intel Xeon E5-1630 v3 3.7 GHz CPU and the microserver is an Intel NUC with an i5-4250U 2.6 GHz processor. Tests were done using a 1Gb/s link.
For the S2 to S3 case, it takes 400 msecs for the first VM to be migrated and 512.7 msecs for the 1000th VM. When replaced with a faster sender (S1 to S3) results improve considerably to 314.1 msecs for the first one and 432.6 msecs for the last one; for the server to microserver case (S2 to NUC), it takes 413.7 msecs for the first VM and 526.9 msecs secs for the 1,000th one.
As a further test, we connected two S3 servers over a direct, 10Gb/s link. With this setup, we were able to measure a migration time of only 86.4 msecs (versus 143 msecs on a 1Gb/s link). We further obtained a migration time of 480 msecs when using the standard toolstack as opposed to ours, showing that our optimizations provide a significant improvement.
The reader may have noticed approximately a factor of 5 difference between the 86.4 msecs when migrating from S3 versus 413.7 for S2. We note that the difference in memory throughput between these two machines is approximately a factor 5; we are conducting further tests to confirm this as the cause.

[1]To achieve this, after each iteration we destroy the VM at the receiver and re-create it at the sender.

3.1.3 Conclusions

We have presented the notion of the superfluid cloud, an architecture that allows parties other than infrastructure owners to quickly deploy and migrate virtualized services throughout the network: in the core, at aggregation points and at the edge. We have made inroads towards implementing some of the mechanisms required by it, including the ability to concurrently run 10,000 VMs on a single commodity server, instantiate services in 10 milliseconds, and migrate them in under 100 milliseconds.

3.2 Control Plane for Liquid MPLS VPN

Quagga [39] is an open-source implementation of the control plane for an IP router. It is supported by Linux, *BSD and other operating systems and, when building a simple router, will communicate with the routing table of the machine's kernel in order to install the routes it learns through the different routing protocols it supports.

Among the supported routing protocols, BGP-4 provides support for large scale IPv4 and IPv6 routing. The original Quagga source has support to the multi-protocol extensions to support IPv6 and multicast. However, it does not include support for other protocols like IPv4 MPLS VPNs as defined in RFC2858 [2]. Moreover, Quagga includes partial support for the mechanism defined in the aforementioned RFC to advertise label information. As described in previous WP3 deliverables, the original Quagga BGP source has been improved and debugged to include this feature and implement a version that allows to advertise networks for MPLS VPN clients. This enhanced Quagga BGP version is also used as routing engine in the VNF pool enabled virtual CPE in the datacenter to implement liquid MPLS VPN services, as detailed in Deliverable D1.4.

3.2.1 Proof-of-concept in-lab deployment

In order to test the control plane, we integrated the enhanced Quagga routing daemon as a Carrier-Grade NAT (NAT) control plane in a virtual CPE in-lab proof-of-concept shown in Figure 3.4.

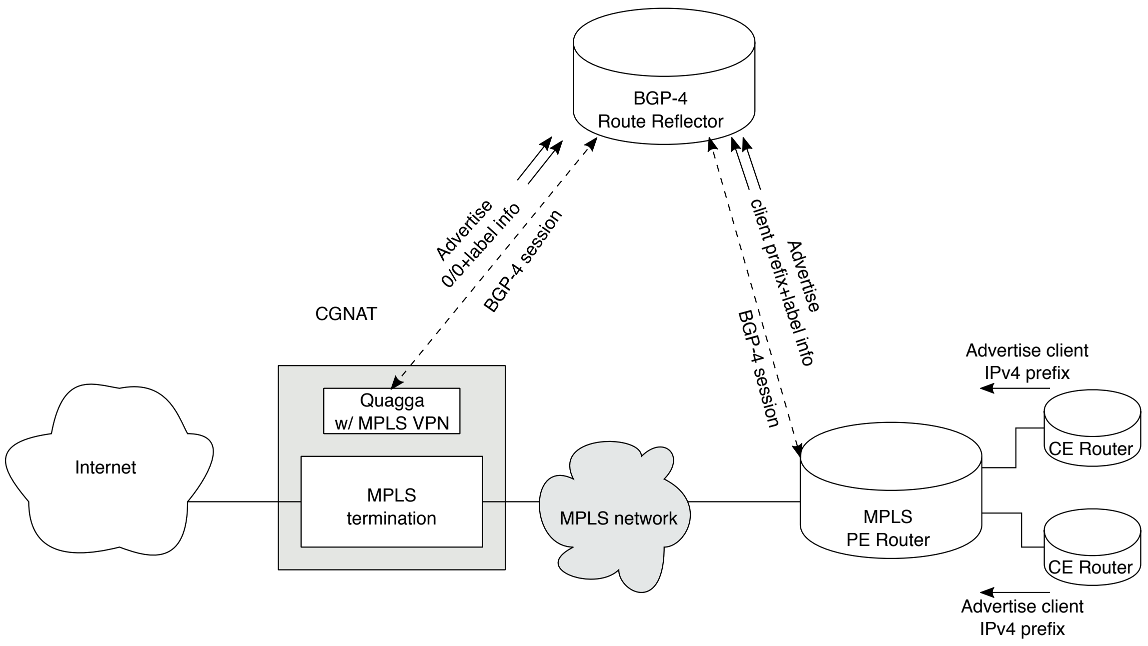


Figure 3.4: Virtual CPE architecture in-lab proof-of-concept: functionality of a CGNAT using the liquid control plane

The proof-of-concept was designed under following assumptions:

- The MPLS termination component terminates MPLS traffic and performs the network address translation functionality automatically.
- Public IP addresses are assigned randomly to users depending on the incoming label.
- Public IP addresses assignments were kept during the experiment execution

- The CGNAT will be the default termination point for all user IP traffic and thus advertises the default route (0/0) for each client.
- The PE advertises the prefixes of the CPEs
- Label assignment is done manually in the Quagga MPLS/VPN control plane and the PE router
- Fully dynamic assignment: no static port mappings in this experiment

3.2.2 Evaluation

To evaluate the scenario, we connected several computers to the CPEs and simulated user Web traffic with scripts, a program to download Youtube videos and the WGET program. As a the experiment was a functional proof-of-concept, we limited ourselves to check that the different programmed downloads were done correctly (files existed in the download PC and were the expected ones).

3.2.2.1 Lessons learnt

The development of the control plane for liquidity has been a valuable instrument. It has helped us creating clear guidelines for the use of open-source implementations in our virtual network functions, it showed us that BPG-4 will continue to play a key role in the liquid network and gave us some space to experiment with the mobility of control plane functions.
The implementation of the MPLS/VPN functionality distributed in the stock Quagga development branch was incomplete and did not interwork with the equipment in the lab. To make it interwork was not, however, the most difficult task. This component is part of our roadmap towards a set of Open-Source virtual network functions (VNFs) for our NVF reference laboratory. However, Quagga is GPL licensed. This implies that any modification done to it needs to be GPL licensed too. Therefore, we had to setup a github repository for our branch and keep it until our improvements are merged into the main branch. This has still not happened and we will continue to maintain the repository. However, this effort is worthwhile because the Quagga daemon provides the FPM interface, which is well documented as a means to interact with other external (software defined) control functions and isolate potential licensing issues between components.
The use of MPLS as a control plane for liquidity is taking up in another direction of that proposed by Trilogy2. However, when we started implementing the control plane, we were among the first efforts to do so.
Isolating the control plane in a virtual machine has helped us to implement a first VNF, fully portable and and capable to interact with other (physical) control plane entities by means of MPLS protocols, in particular BGP-4. As a further proof of this flexibility, the Quagga virtual machine is a fundamental component of the VNF Pool enable virtual CPE prototype described in Deliverable D1.4. On the other hand, the isolation of the control plane in a virtual machine has also helped us understanding the different impacts on the infrastructure that moving VNFs can have.

3.2.3 Application/exploitation

This proof-of-concept was used to demonstrate the capabilities of an MPLS control plane controlling liquidity and as a first virtual network function component (VNFC) in the Telefónica I+D internal portfolio. We can combine it with different MPLS termination data-plane components using the FPM interface. This positions us favourably with regards to vendors to come to Telefnica's NFV reference laboratory.

3.3 Providing Resilience to IMS virtualised network functions

Nowadays, telco operators own a huge and complex infrastructure to provide their services to their end-users. To maximize the usage of such infrastructure, and to accommodate the instantaneous requirements of their customers, these network operators usually deploy load balancers to select the proper network device to serve incoming requests. On the other hand, the traffic load generated by the end-users is variable during the day, where the high differences between the peak and the valley zones impose different network requirements. In such scenarios, the telco operator would like to add or remove active machines, in order to reduce costs. This reduction of costs is even clearer when operators use virtual network functions of third-party providers. By using virtualization, telco operators could dynamically request the instantiation of virtual machines in order to adapt their network resources to the traffic load. However, even using load balancers to allocate the incoming end-users traffic to (possibly virtual) network element functions, the change in number of available

active elements generates different loads on such devices. In that case, it would be desirable to transfer load between network elements, to maximize their utilization while minimizing the number of active network devices.
The IP Multimedia Subsystem (IMS) framework defined by the 3GPP is a next generation network architecture to provide multimedia services over IP networks. This architecture is designed to be scalable throughout the redundant instantiation of these entities and the usage of load balancing mechanisms, such as those supported with the DNS. In this respect, the considerations described before about the transfer of the load between functional elements would be beneficial for the IMS core elements too. Additionally, with such mechanisms, it would be straightforward to add resilience to both virtual and physical IMS deployments, as they would enable to transfer the load of failing core elements to other existing elements, without disrupting the service provided to the end users. IMS is playing a relevant role in 3G/4G telco support infrastructures, even more with the introduction of Voice over LTE (VoLTE), with many general concepts that are going to influence and to persist in future 5G networking. In particular, the elastic provisioning of virtualized IMS-based functions is still considered a challenging issue.
In this use case, already explained in Deliverable 3.2, we propose a new architecture to transparently transfer users among the functional elements that implement the IMS call session control functions. With our approach, a network operator can better adapt its active control resources to the instantaneous load generated by the users, also considering roaming users from other operators, with evident advantages in terms of efficiency and economic costs. It would also be possible to use this solution to add resilience to IMS after failures, transferring all the state of the failed control element to a new machine.
In this subsection we present the main improvements compared with the proposal already presented in D3.2. For the sake of completeness, we also include the proposal in a schematic way. Finally, we include the main results obtained with a proof-of-concept implementation.

3.3.1 Transparent reallocation of IMS control functions

This section describes the functional entities that are proposed in this use case to enable an appropriate allocation of users to the available Call-Session Control Functions (CSCFs) elements in IMS deployments. These entities, interoperating with the IMS, allow a home operator to dynamically change the allocation of the Proxy CSCF (P-CSCF) or Serving CSCF (S-CSCF) of any of its users, transparently to the end-user applications running at the User Equipment (UE). This way, our architecture enables the home operator to maintain an appropriate distribution of the load among the existing CSCFs. Figure 3.5 shows the new functional elements of our solution (highlighted in gray color), as well as their relationship with the elements of the IMS architecture. It is important to remark that our solution does not add new interfaces or functionality to standard IMS elements, so it does not require modifications to the IMS specifications.
The *Control Function Discovery* (CFD) is an application server (AS) 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 or re-registration of the user in the IMS, and following the regular IMS procedures, the CFD AS will receive the information corresponding to this particular registration, which will include, among other things, the value of the registration expiration interval and 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 new database in the home network domain that maintains all the information related with the IMS control functions serving the users of the home operator. For the sake of scalability, there may be several CFD AS entities in the home network, to appropriately process the load of notifications about the registration status of the users. However, we want to highlight that our solution does only require a reduced set of these entities with respect to the number of CSCFs, as CFD ASs are only contacted after the successful registration and re-registration of the user, and are not involved in any other IMS signaling procedures (e.g., session setup, event subscription and notification, etc.).
The *Control Function Selection* (CFS) is the key component of our solution that enables the change of the P-CSCFs and S-CSCFs allocated to the users of the home operator. This functional entity is the contact point of our solution with the Operations Support Systems/Business Support Systems (OSS/BSS) of the home network domain. The OSS/BSS can activate or deactivate control functions as needed, namely P-CSCF, S-CSCF and I-CSCF entities, to scale the IMS control resources to the instantaneous load generated by the users, and triggers the CFS when a redistribution of the load among the existing CSCFs is required. This may happen, for instance, to enforce a transfer of the load of an underutilized CSCF to other existing CSCFs, prior

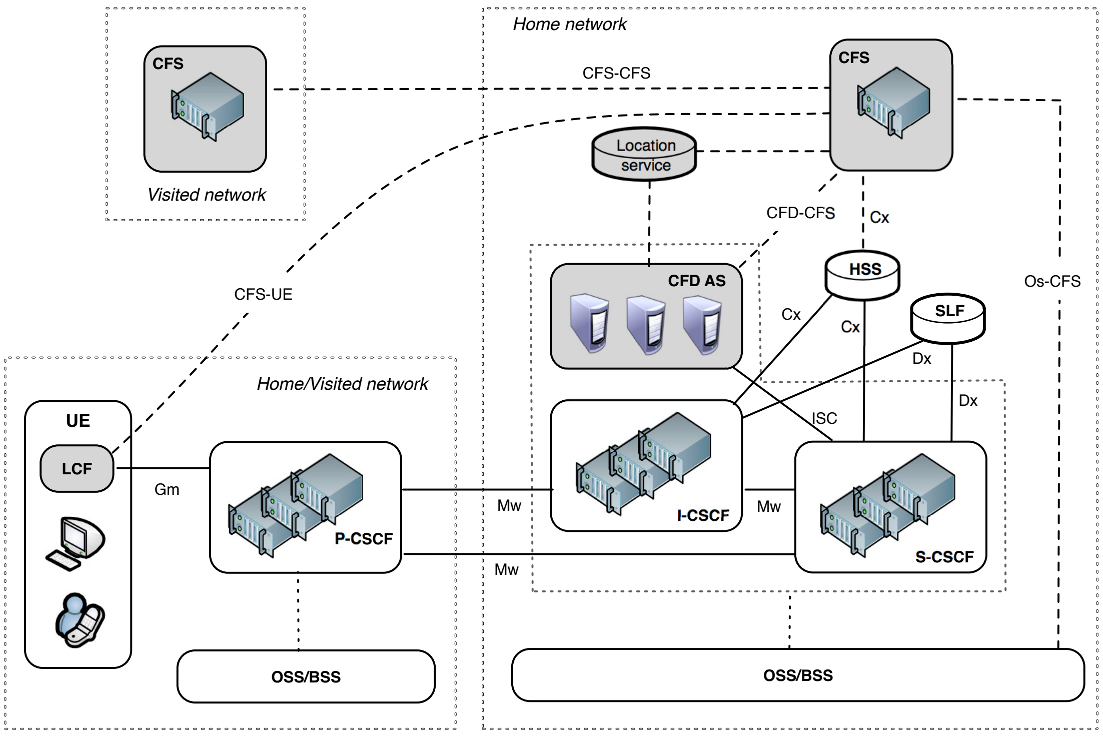


Figure 3.5: Overview of the proposed architecture.

to its deactivation, or to achieve the resilient operation of an IMS deployment under a failing or overloaded control node. To support an appropriate operation, the CFS implements a Cx interface, as defined for the IMS in [1], which enables the communication with the HSS. This interface will be used in S-CSCF reallocation procedures, as it will be explained later.

The CFS has access to the information stored by the CFD AS in the location service of the home network domain. Thus, upon receiving an indication from the OSS/BSS, via the reference point os-cfs, to perform a load transfer from a P-CSCF (or S-CSCF) to other existing CSCF entities, it may retrieve the information about the users served by the former CSCF from this location service. With this information, the CFS can initiate a reallocation procedure to satisfy the request received from the OSS/BSS, transferring a subset of the users from the initial CSCF to the target CSCFs.

Our architecture includes a Local Control Function (LCF) in the UE. This is integrated as an extension to the IMS stack of the terminal[2], and maintains status information about the SIP dialogs established by the user. The LCF supports a new reference point cfs-ue, which can be used by the CFS to trigger the transfer of the user to a new CSCF. The LCF is in charge of executing the signaling procedures that are necessary to perform this transfer. Moreover, it carries out these procedures transparently to any end-user application running at the UE, which are kept unaware of the CSCF change.

In the following, we illustrate the procedures used in our solution to change the allocation of the P-CSCF or the S-CSCF for a given user of the home operator, and also to enable an appropriate load balancing among I-CSCF and CFD AS entities.

3.3.1.1 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 set of users. Prior to the execution of these procedures, we assume that the users have already registered to the IMS, and may have established a set of multimedia sessions by means of SIP dialogs. The procedures are outlined in Figure 3.6.

In the scenario shown in this figure, the OSS/BSS in the home network contacts the CFS to perform a load transfer from a P-CSCF to other existing P-CSCFs (Step 1). Next, the CFS retrieves from the location service

[2]The required functionality can be implemented as software in the UE, and it can be delivered integrated with the required software to access the IMS services.

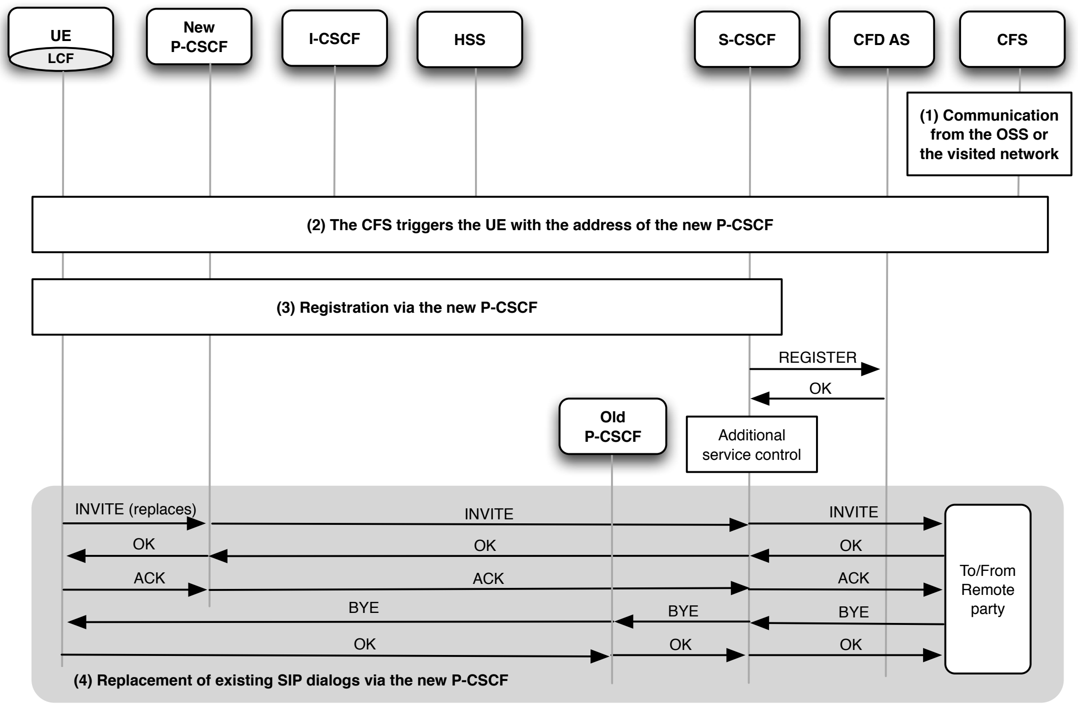

UE
LCF
New P-CSCF
I-CSCF
HSS
S-CSCF
CFD AS
CFS
(1) Communication from the OSS or the visited network
(2) The CFS triggers the UE with the address of the new P-CSCF
(3) Registration via the new P-CSCF
REGISTER
OK
Old P-CSCF
Additional service control
INVITE (replaces)
INVITE
INVITE
OK
OK
OK
ACK
ACK
ACK
BYE
BYE
BYE
OK
OK
OK
To/From Remote party
(4) Replacement of existing SIP dialogs via the new P-CSCF

Figure 3.6: IMS procedures to change a P-CSCF allocation.

the information about the users served by the affected P-CSCF, and determines the set of candidate users that will be transferred to each target P-CSCF. After defining the set of candidate users for each target P-CSCF, the CFS starts a reallocation procedure, to transfer each identified user to its corresponding target P-CSCF. For this purpose, it communicates with the UE of each user to be transferred (Step 2), using the reference point cfs-ue.[3] As a result of this communication, the LCF at each UE can independently initiate the IMS signaling 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 received by 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 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 by the LCF, according to the procedures defined in [28], using the status information corresponding to the SIP dialogs of the user. To clarify this concept, Step 3 in Figure 3.6 shows an example, where the LCF replaces a single SIP dialog established by the user.
Finally, our solution does not impose or recommend any specific mechanism to select which specific users will be reallocated to each target P-CSCF. Instead, the final decision is uniquely subject to the policy rules established by the home operator. Moreover, the proposed solution does not define the precise instant of time when each candidate user is to be contacted to carry out the reallocation. On the contrary, the time schedule of the reallocation procedures is dictated by the operator, who might contact all the candidate users simultaneously, or spread over time the requests sent to UEs. Additionally, the operator might provide the LCF (through the cfs-ue interface) with diverse policies to govern the user transfer. As an example, the operator may instruct the LCF to delay the replacement of each SIP dialog established by the terminal to a given deadline. This would allow distributing over time the signaling load corresponding to dialog replacement procedures, and even reducing this load by allowing the dialog to be terminated by the end user, via the old P-CSCF, before the deadline.

[3]The communication between the CFS and the LCF should be done out of the SIP signaling path, as this communication can be the result of a failed or overloaded P-CSCF, and it could be implemented, for example, using HTTP.

3.3.1.2 Change of the S-CSCF allocation

The procedures to change the S-CSCFs allocated to the users are similar to those described in the previous section, and are illustrated in Figure 3.7. After receiving a trigger from the OSS/BSS to perform a redistribution of the load of a given S-CSCF to other existing S-CSCFs (Step 1), the CFS retrieves the set of users served by the specified S-CSCF. Following the policy rules defined by the home operator, the CFS determines the set of candidate users that will be transferred to each target S-CSCF. Then, for each candidate user, 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 UE of each of the users, using the reference point cfs-ue, instructing the LCF to start the IMS procedures needed to enforce the allocation of the new S-CSCF to the user (Step 3). Analogously to the previous case, the LCF first initiates an IMS-level registration using the new S-CSCF. Step 4 in Figure 3.7 illustrates an example of the registration process, where the S-CSCF performs user authentication procedures. In the example, the registration proceeds in two phases. In a first phase, the UE 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 an 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. Consequently, the new S-CSCF receives the REGISTER request and contacts the HSS to download the data that is necessary to authenticate the user. Then the S-CSCF answers back the REGISTER request with a SIP Unauthorized response, containing a challenge to be satisfied by the UE. The response to this challenge requires a second REGISTER transaction between the UE and the S-CSCF, as shown in Figure 3.7.
After the successful registration of the user, the new S-CSCF performs the regular IMS 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 LCF initiates the signaling procedures to replace the SIP dialogs established by the user to use the new S-CSCF (Step 5).
Analogously to the P-CSCF reallocation procedure, note that the actual decisions on which candidate users are to be transferred from a given S-CSCF to another, and the precise time schedule of the reallocation process, are controlled by the policy rules established by the operator.

3.3.1.3 Consideration about roaming users

According to the IMS specifications, there is the possibility that the P-CSCF allocated to a roaming user is located in the visited network. In that case, there may be users served by a P-CSCF that are not registered in the network of the operator owning the P-CSCF. Our solution covers this specific use case, by allowing communications between CFS entities in the home and visited networks through the inter-domain reference point cfs-cfs.
After the successful registration of a user who is roaming in a visited network, a CFD AS in the home network of the user stores the address of the visited P-CSCF in its location service. Additionally, when this CFD AS detects a new address of a visited P-CSCF, it contacts the CFS in the home network (hereafter referred to as home CFS) via the reference point cfd-cfs. If the visited network implements the solution presented in this document, the home CFS communicates with the CFS in the visited network (hereafter referred to as visited CFS), through the reference point cfs-cfs, and registers its desire to receive notifications about the state of the P-CSCF allocated to the user.
Now, if a P-CSCF is overloaded, fails or needs to be removed, the operator owning the P-CSCF may contact any operator that has expressed its interest in receiving notifications on the P-CSCF status. To this end, the visited CFS (i.e., the CFS in the domain of the affected P-CSCF) communicates with the home CFS via the cfs-cfs interface. The visited CFS may explicitly indicate the reason of the communication, and request from the home CFS information about the visiting users that are served by the P-CSCF (e.g., the number of these users). Taking into account the information from the different CFS entities that have been contacted, and according to the policy rules established by the visited operator, the visited CFS can then request from any home CFS entity to perform a reallocation process of its visiting users. The trigger from the visited CFS may include a set of alternative P-CSCFs and the assignment of load that should be transferred to each of these P-CSCFs. With this information, and attending to the policy rules defined by the home operator, the home CFS then executes the reallocation process following the procedures that have been previously described.
Finally, we want to note that there is always the possibility of having roaming users assigned to a P-CSCF

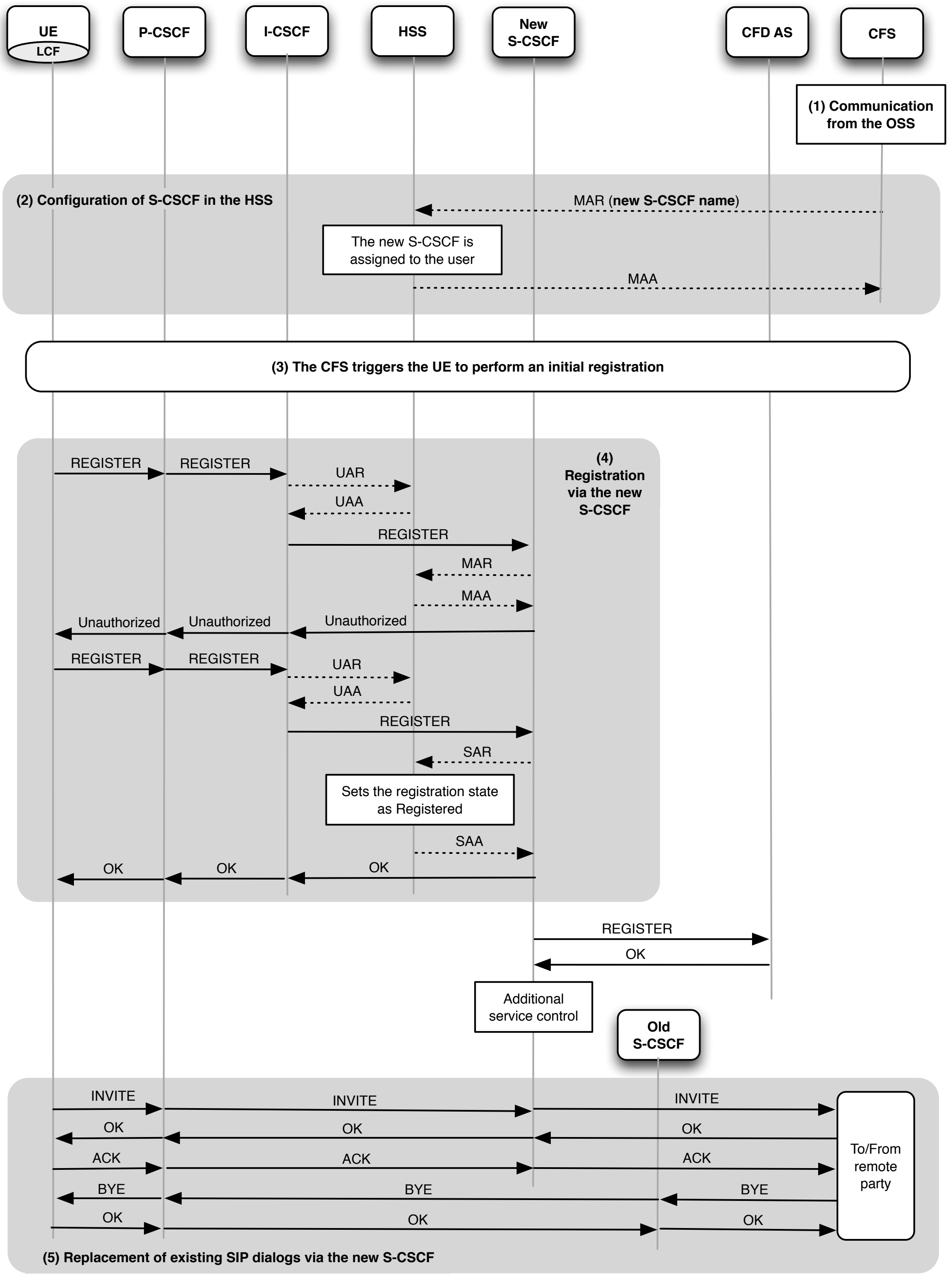

UE
LCF
P-CSCF
I-CSCF
HSS
New S-CSCF
CFD AS
CFS
(1) Communication from the OSS
(2) Configuration of S-CSCF in the HSS
MAR (new S-CSCF name)
The new S-CSCF is assigned to the user
MAA
(3) The CFS triggers the UE to perform an initial registration
(4) Registration via the new S-CSCF
REGISTER
UAR
UAA
MAR
MAA
Unauthorized
SAR
Sets the registration state as Registered
SAA
OK
Additional service control
Old S-CSCF
INVITE
ACK
BYE
To/From remote party
(5) Replacement of existing SIP dialogs via the new S-CSCF

Figure 3.7: IMS procedures to change a S-CSCF allocation.

Table 3.1: Average delay of the session setup for different values of aggregate session setup rates.

Aggregate session setup rate (sessions/s)	Average delay of the session setup (seconds)
0.5	0.068
1	0.073
2.5	0.7
5	3.1
10	10.97

belonging to a network domain that does not implement the solution described in this document. In this case, the visited CFS would not have access to the information about the visiting users from their home network domain, and would not be able to request the execution of a reallocation process for these users. However, this issue can be easily prevented if the service-level agreement, established with other operators, allow using the set of P-CSCFs of the operator only to those operators that implement the control function discovery and selection procedures described in this document (users can still be provided service with a P-CSCF in their home networks).

3.3.1.4 Experimental results

To assess our proposed solution, we have deployed a testbed using the FOKUS OpenIMS core[4] and the SIPp[5] open software. The former includes the call session control functions of the IMS (i.e., P-CSCF, S-CSCF and I-CSCF) and an HSS, while the latter is used to emulate an IMS UE. 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 run the SIPp scripts corresponding to the UEs.

To illustrate the benefits that can be achieved reallocating users to CSCF entities, we designed a first experiment with 20 users. All these users are served by P/S-CSCF1 and are configured to generate a certain aggregate session setup rate. In the experiment, we measured the average session setup delay for different session setup rates. The results are shown in Table 3.1. As it can be observed, in our particular testbed, when the session initiation rate is over 5 requests/s, the session setup delay is above the grade-of-service parameter for call setup delays recommended by the ITU-T of 3 seconds [20]. This shows the benefits of distributing the rate of sessions established by the users among CSCFs, as supported by our reallocation procedures.

In the second experiment, 20 registered users allocated to P-CSCF1 establish IMS sessions with 2 registered users using P-CSCF3. The aggregate session setup rate is 1 session/s. Using the SIPp scripts, we execute the procedures to change the allocation of 10 users from P-CSCF1 to P-CSCF2, as illustrated in Figure 3.6. This change is triggered manually approximately after 60s. In this particular setup, all the users are transferred in parallel, which imposes a transitory high load for both P-CSCFs. Figure 3.8.a) presents the result of this experiment in terms of total throughput at the two P-CSCFs, where we can observe that, once the users are transferred to the new P-CSCF, the load is distributed between them. Note that the throughput shown corresponds to SIP signaling and it is also a measurement of the load at the CSCFs to process those SIP messages. Although the specific policy to reallocate users between functional elements is under the control of the home operator, for the sake of completeness we also consider in this validation a simple algorithm to schedule the transfer of users between P-CSCFs. In such an algorithm, the users are contacted in sequence with a configurable transfer rate. In the third experiment, where we configure a transfer rate of 2 users/s, we are able to significantly reduce the peak throughput during the transfer stage, as shown in Figure 3.8.b).

In the fourth experiment, we show a scenario where the load of two S-CSCFs are below the minimum configured threshold, so all the users assigned to one of them can be transferred to the other. In this case, there are 10 users registered in S-CSCF1, while another 10 are registered in S-CSCF2. The 20 users generate an aggregate rate of 1 session/s, as in the previous experiment. After approximately 60 seconds, the users registered in S-CSCF2 are transferred to S-CSCF1 at a transfer rate of 2 users/s, using the mechanisms illustrated in Figure 3.7. As it can be seen in Figure 3.8.c), the users and their active sessions are transferred to the new

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

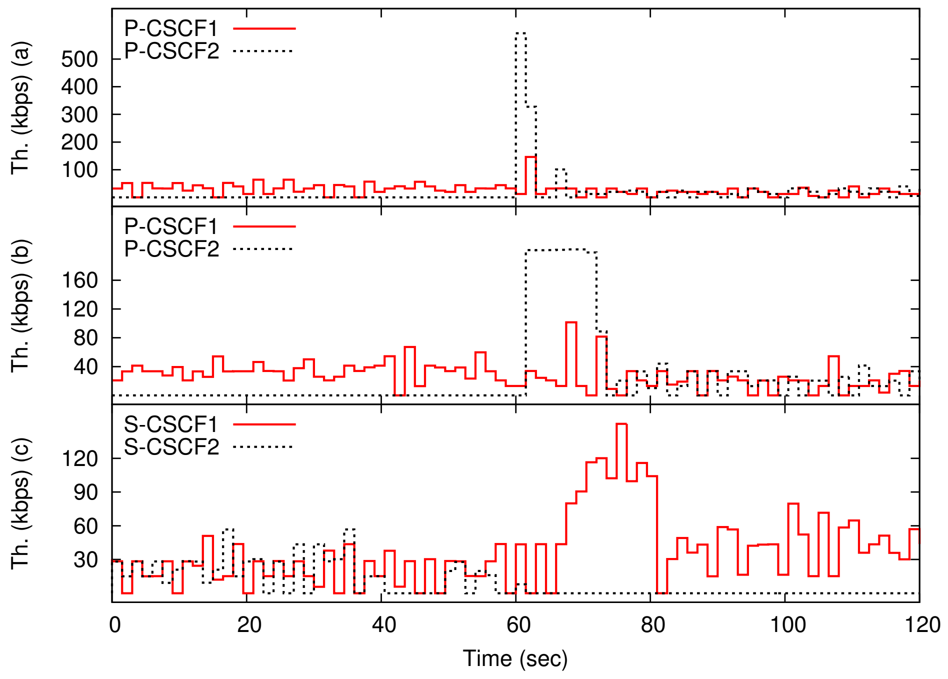


Figure 3.8: Results of the experiments re-allocating users between P/S-CSCFs.

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 a rate of 1 session setup per second.

3.4 Minicache Video Delivery

In deliverable D1.4 we provided a full description of MiniCache and its implementation. We now proceed to present evaluation results from our implementation of MiniCache as well as a few results from large-scale CDN simulations that were used to show the effects and advantages that on-the-fly, virtual CDNs have.[6]

3.4.1 Simulator Results

In this section we evaluate the potential wide-area effects that deploying ephemeral CDNs might have, and compare these to traditional CDN deployments.

3.4.1.1 Setup

We implement a custom, flow-level CDN simulator from scratch consisting of about 2,000 lines of Python code. At the highest level, the simulator uses the Internet's AS-level topology obtained from the Internet Research Lab (IRL)[7] to build its network graph (48,991 ASes in total). We further use the IRL's data to assign IP address prefixes to ASes, and use longest prefix matching for forwarding packets. In addition, we take advantage of CAIDA's AS ranking[8] to distinguish between content, transit or access ASes [26].

In terms of link speed, we assign a capacity of 40Gb/s to inter-AS transit links, 10Gb/s to leaf-AS links, and 25Mb/s for end user connections (25Mb/s is the amount recommended by Netflix for 4K video [32]). Video streams are consumed at the rate defined by their video quality: 360p – 1 Mbps, 480p – 2.5 Mbps, 720p – 5 Mbps, 1080p – 8 Mbps, 2K – 10 Mbps and 4K – 20 Mbps [45].

We simulate 200 channels (e.g., the work in [3] cites 150 for a large IPTV system) and apply video quality rates from the range above using a Poisson distribution. For the requests, we use a Zipf distribution, a typical distribution for content popularity, to decide which content/channel they should retrieve. Regarding viewing duration, the authors of [3] describe it as ranging from 1 minute to an hour. Based on this, we use a triangular distribution within that range peaking at 30 minutes (e.g., the duration of a sitcom series). Requests arriving

[6]Note that most of the development work done for the simulator was carried out under the EU H2020 SSICLOPS project; we present a few of the results here for reference.

[7]http://irl.cs.ucla.edu/topology/

[8]http://as-rank.caida.org/

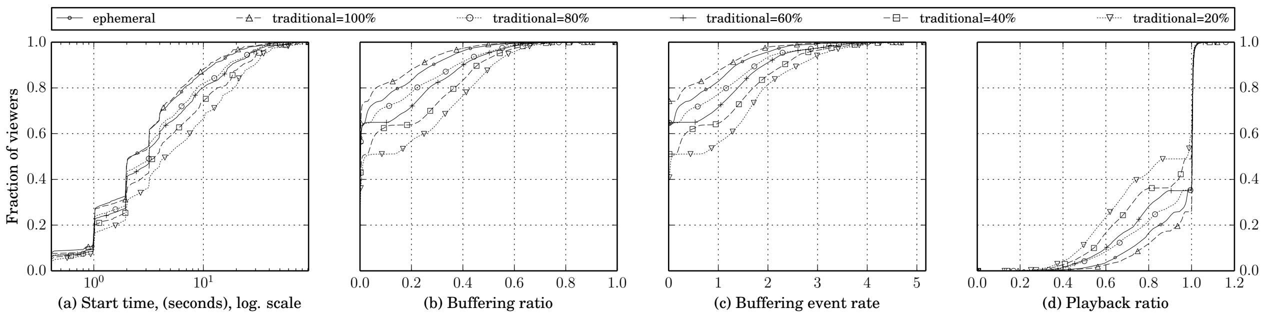


Figure 3.9: CDFs of QoE metrics for traditional vs ephemeral CDNs. The percentages for the curves marked "traditional" denote the level of cache deployment at the AS-level for the entire network.

at an access AS that does not have a virtualized content cache are directed to an origin server; in parallel, we instantiate a cache to handle any subsequent requests arriving at that AS.
For each stream request, the simulator keeps track of a number of standard video QoE metrics: (1) **start time**, the time between the video player initiating a request and the moment its buffer is filled up so that it can start playback; (2) **buffering ratio**, the fraction of total session time spent buffering; (3) **buffering event rate**, essentially the number of interruptions that the user experiences throughout the lifetime of the stream; and (4) **playback ratio**, the ratio of download rate to the video code's rate (1 being the optimal).
In order to keep simulation time reasonable and still produce meaningful results, we restrict our simulations to a subset of the overall AS topology containing 1,187 ASes, 2,880 links between them; this is comparable to a large country. Of these, 239 are content provider ASes, meaning that origin servers can be placed in them, and 871 are access provider or "eyeball" ASes – potential sites for content cache deployment.
To show the effects of ephemeral CDNs at large scale, we simulate one million users with at least 100,000 concurrent connections at any one point in time and a peak of 160,000 connections. Streams are generated randomly, with an average rate of 6,670 requests per second. As a point of reference, Rutube, one of the largest video-hosting services in Russia claims to serve up to 65,000 concurrent live streams [43]. We thus believe the simulations should be able to capture large-scale effects.
Note that for simplicity, we use a non-cooperative caching approach (i.e., on a miss, caches fetch content from origin servers as opposed to from other caches) as does Akamai [47]. We do not claim this choice to be optimal; the aim is not to provide a reference design for others to use, but rather to make the case for ephemeral CDNs and show that these can be made to work in practice by leveraging the mechanisms developed in this paper. Please refer to [47] for an analysis of different caching strategies (e.g., cooperative versus non-cooperative).

3.4.1.2 Results

For the first set of simulations, we measure QoE metrics when running ephemeral CDNs. We assume that we can bring up a cache instance in each of the access provider ASes (as labeled by CAIDA as previously mentioned); this is of course an ideal case, we expect it to become increasingly common to be able to deploy VMs on third-party infrastructure. We compare this to traditional CDNs with different content cache deployment levels ranging from 20%-100% of the access provider ASes selected randomly.
Regarding start-up times, the CDF in Figure 3.9a shows that the ephemeral CDN results in a rather reasonable 3s start time for most of the viewers, whereas a traditional CDN with 20% or 40% deployment would cause users to wait a considerably longer 8 and 5 seconds respectively.[9] For the buffering ratio (Figure 3.9b), we see 80% of streams experiencing a ratio of about 14% for the ephemeral CDN, compared to about 22%, 27% and 37% for a traditional CDN with 80%, 60% and 40% deployment, respectively. The buffering event rate (Figure 3.9c) and playback ratio (Figure 3.9d) are likewise improved.
Next, we evaluate how the virtualized cache boot times affect the overall performance of the ephemeral CDN.

[9]Note that 100% deployment, also plotted in the graphs, is unrealistic but it is interesting in that it shows that it performs equally or slightly better than the ephemeral CDN; this is due to the small but non-negligible VM start-up times (set to 100 msecs in this set of simulations).

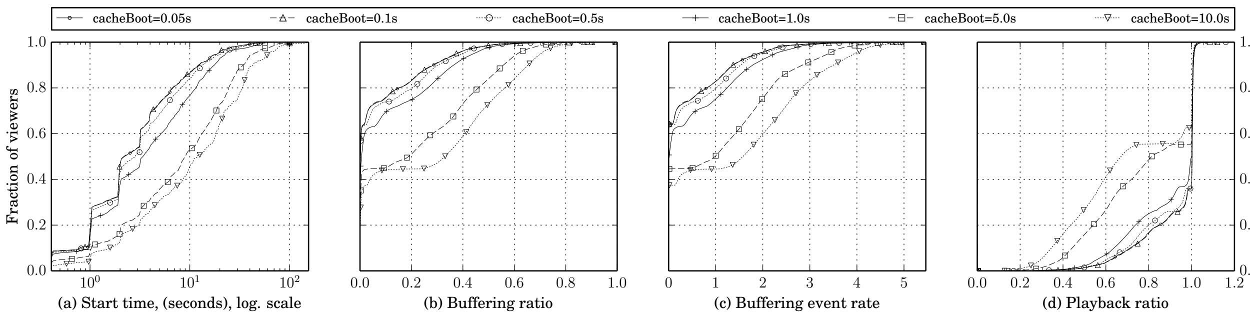


Figure 3.10: Ephemeral CDN virtualized cache boot times and how they affect QoE.

Longer times mean that a larger fraction of users would not find a cache in their access provider AS and would have to fetch content from the origin server. We vary the boot times between 50 msecs and 10 seconds and plot the effects of this on the QoE metrics in the CDFs in Figure 3.10.
We get the best results with boot times in the 50-100 msecs range, with a big degradation in QoE for boot times of one second and higher (as reference, in the next section we will show that MiniCache can boot in as little as 42 msecs on x86 and 139 msecs on ARM32). For instance, for 60% of users and boot times up to 100 msecs the start time can be kept to a low 3 seconds, with this value jumping to 13-18 seconds for boot times of 5 and 10 seconds, respectively. This significant increase is due to the early viewers being redirected to the origin server via the longer path while a content cache is not ready. The load they introduce affects the lowest throughput segments in the path, reducing the bandwidth available for the content cache to operate. Regarding buffering ratio, taking again 60% of users we see no or very little rebuffering for boot times <100 msecs and a rather high 28% and 40% ratio for boot times of 5 and 10 seconds respectively. We see similar effects for the buffering event rate and playback ratio.
Note that some of the QoE values exhibit rather poor performance for some of the users. For example, the buffering ratio for 100% of viewers in the CDFs can be as high as 40%-60%. This is due to link congestion between edge ASes and upstream ones. We could of course tweak link capacities to obtain even better results, but the main point of the evaluation is to show the relative effects of ephemeral CDNs versus traditional ones. Overall, we see that ephemeral CDNs can yield important improvements to end users' QoE, and that these further depend on how quickly the virtualized caches can be instantiated. Next, we provide a full description and evaluation of MiniCache, our implementation of a virtual cache node intended for use in ephemeral CDNs.

3.4.2 Minicache Evaluation

In this section we provide a thorough evaluation of MiniCache to check whether it meets the requirements outlined in Deliverable D1.4. In particular, we look at (1) boot and destroy times, (2) crash detection and restart time, (3) SHFS and block I/O performance, (4) HTTP performance (throughput and requests/sec) and (5) VM image size and memory consumption.
For x86, we conducted all evaluations on a server with a Supermicro X10SRi-F server board, a four-core Intel Xeon E5 1630v3 processor running at 3.7GHz, 32GB of DDR4 RAM split across four modules, an Intel 530 Series 240GB SSD and a Mellanox ConnectX-3 Ethernet card to carry out experiments up to 40Gb/s. For ARM we rely on a Cubietruck with an Allwinner A20 SoC containing a dual-core ARM Cortex A7 running at 1GHz, 2 GB DDR3 RAM, a Crucial M4 128 GB SSD and a 1Gb/s Ethernet interface.

3.4.2.1 Boot and Destroy Times

As shown in the simulation results, one of the critical factors for providing good QoE for users is for MiniCache VMs to be able to be quickly instantiated. The results presented here include all of the optimizations described in deliverable D1.4, including the hotplug binaries and the minimalistic Xen store and toolstack.
Figure 3.11 shows boot times for several setups that we investigated. "Creation" denotes the time spent on instantiating the virtual machine; "booting" is the time from instantiation to availability of connectivity; and "HTTP" the time spent until the HTTP cache is available. Using the same virtualization technology and architecture (Xen on x86), MiniCache is faster by one to two orders of magnitude. In fact, even running on

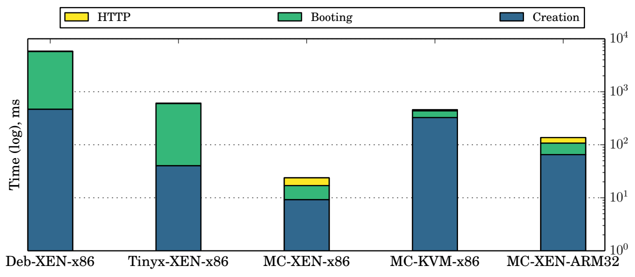


Figure 3.11: Comparison of MiniCache boot times for Debian, Tinyx and MiniCache on x86 and ARM.

resource-constrained systems (ARM32), or running a not-yet-optimized version of MiniCache on KVM, it still outperforms the other two comparison systems if they run on Xen-x86.
MiniCache virtual machine destroy times are quite small: for Xen we measure 5 msecs (x86) and 26 msecs (ARM), and for KVM on x86 a destroy time of about 7 microseconds (using the `kill` command).

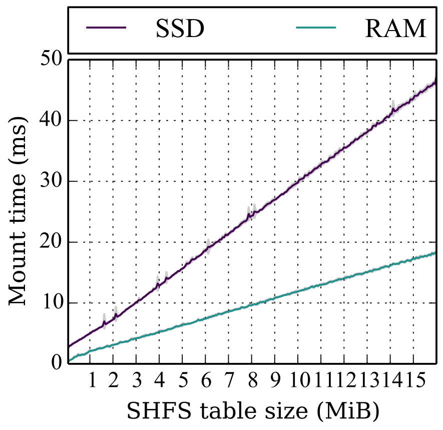


Figure 3.12: SHFS mount times for different table sizes when using an SSD or RAM device.

Because SHFS reads in the hash table to memory during mount time we also evaluated the costs that it adds to the MiniCache VM boot time (Figure 3.12). We found a linear correlation between the SHFS hash table size and the mount time, a result of the underlying block device speed.

3.4.2.2 Crash Detection and Restart Time

To make sure that a MiniCache crash does not overly affect ongoing streams, we evaluate the MiniCache mechanism that detects the crash of a VM and automatically relaunches the instance. To do so, we measure the time between receiving the last TCP packet of the crashed VM and the first one from the restored instance. We further run this same test while using the standard Xen toolstack and Xenstore for comparison. On x86 we obtain 39 msecs with all the optimizations and 2068 msecs with standard Xen; ARM results in 173 msecs and 2467 msecs respectively.

3.4.2.3 SHFS and Block I/O Performance

To evaluate SHFS, our purpose-built filesystem, we sequentially read a 256 MB file, meaning that we request the next chunk from SHFS whenever the data for the current one has been copied out. To saturate the block I/O pipe we attach a RAM block device to Xen's `blkback` driver. As further optimizations we implement persistent grants (refer back to deliverable D1.4) in MiniOS' `blkfront` driver and on the `blkback` one, and make use of AVX/SSE instructions for memory copies. As a final optimization we parallelize read requests by reading ahead up to N=8 chunks. The results in Figure 3.13 show the read performance in GB/s. Persistent grants gives a bump of about 25-50% with respect to the baseline depending on block size. Reading chunks ahead further boosts performance, up to as much as 5.8 times for 8 chunks read ahead for a maximum total of about 4.4 GiB/s.
Next, we evaluate how quickly SHFS can perform open and close operations (Figure 3.14) and compare that to `tmpfs` and `ext4` on bare-metal Linux (i.e., no virtualization). As shown, SHFS can carry out open/close

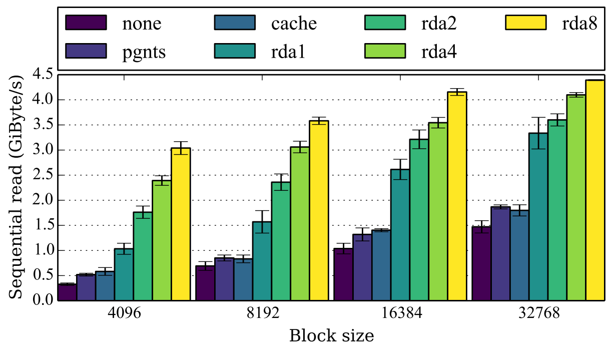


Figure 3.13: Breakdown of the different optimizations for SHFS and how they affect the sequential read performance of a MiniCache VM. Base is without any optimizations, pgnts stands for persistent grants and rda$n$ stands for n blocks read ahead.

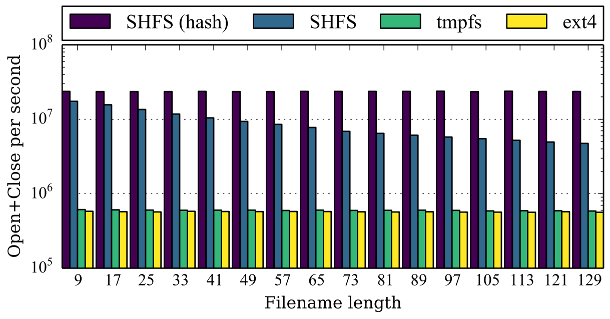


Figure 3.14: Open and close operations per second for SHFS on a MiniCache VM compared to ext4 and tmpfs on bare-metal Linux. Filename length denotes the number of characters in the file name. SHFS (hash) stands for doing the open operation with the hash digest directly instead of with its string representation.

Component	Binary Size (Bytes)
MiniOS	502,155
lwIP	128,116
SHFS	39,437
HTTP	53,233
μSh	12,058
Main routine	6,811
Other	14,458
Padding	3,331,236
Total	**756.268**

Table 3.2: Breakdown of the different components of a MiniCache VM image on Xen.

System	Image size	Min. memory
MiniCache on Xen	3.9 MiB	8 MiB
MiniCache on OSv v0.23	5.9 MiB	31 MiB
lighttpd on OSv v0.23	6.1 MiB	34 MiB
Tinyx+nginx	7.5 MiB	51 MiB
Tinyx+lighthttpd	1.8 MiB	23 MiB
Debian 8.0+lighttpd	489 MiB	82 MiB
Debian 8.0+nginx	467 MiB	82 MiB

Table 3.3: System image sizes and minimum required memory for successful bootup for MiniCache and other HTTP servers.

operations at a rate of 23.6 million/sec independent of the hash digest length which identifies the object. This is largely because (1) open and close are direct function calls to SHFS, (2) opening a file is implemented as a hash table lookup, and (3) the complete hash table is read into memory at mount time.
In case the hash digest needs to be parsed from a string representation, additional costs depending on the file name length have to be added. With a filename with 129 characters (128 characters representing a 512 bit hash digest in hexadecimal and 1 prefix character), SHFS can still perform 4.7 million operations. On the same machine and using Linux, we measured only up to 612K open and close operations per second on `tmpfs`.

3.4.2.4 Image Size and Memory Consumption

Finally, Figure 3.15 shows the MiniCache memory consumption arising from SHFS. All entries, regardless of whether they are used or not, require the same amount of space on the volume (256 bytes) but also when loaded to memory. This means that the amount of required memory is defined by the SHFS table size and not by the amount of added objects.

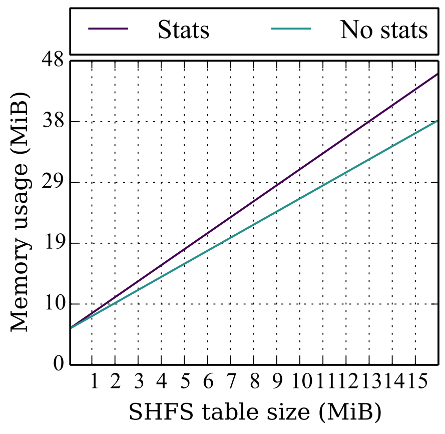


Figure 3.15: SHFS memory consumption for different table sizes with and without traffic statistics.

4 Liquidity in the Wide Area

4.1 Hot Patch Propagation for the Cloud with Irmin

One use case from the UCAM work revolves around the continual update for cloud operating software. The storage system in the Mirage release offers a Git-type service which provides the repository for source, systems and system logging. In the traditional container or classic guest OS world, the scale of a new binary really mitigates against distribution via Git. However, Unikernels are small enough to be tracked in GitHub. For example, the Mirage website is run this way, both for code and data (content).

The life cycle is as follows: Source code updates are merged to mirage/mirage-www. The repository is continuously rebuilt by Travis CI, and if a build is successful, the new Unikernel is pushed to mirage/mirage-www-deployment. The Cloud toolstack spawns VMs based on pushes there, so nothing more needs be done, as it is on automagic. For example, the DNS server interprets git-commit-ref.service.name, checks out appropriate unikernel revision, the x86 boots in 2030ms while Synjitsu proxies TCP handshake, conduit channels created, allowing (per-client) unikernel to protect legacy VMs. More details on this are in Deliverable 1.4.

Our entire cloud-facing deployment is version-controlled from the source code up to the executing unikernels, and back down to the logs which may be the input to performance enhancements or bug fixes. The implications of historical tracking of source code and built binaries in Git(hub) are several: we use git tag to link code and binary across repositories; git log is available to view deployment changelog; its as simple as using git pull to deploy new version; obviously, git checkout to go back in time to any point, and git bisect to pin down deployment failures.

What makes this part of the Liquid paradigm is that we now offer very low latency deployment of security (and other) updates. There is decentralisation of control, as there is no need for Linux distro to pick up and build the new version. There are less likely to be sins of omission, as updated binaries are automatically built and pushed, and subsequently the latest binary is picked up directly from repository.

Because Mirage uses OCaml, a statically type-checked language, a wide range of security vulnerabilities are removed, so the pace of updates does not increase the rate of risk. The combination of the library OS approach, and the repo-driven update, we have a unified development for cloud and embedded environments. The developers can write application code once, recompile to swap in different versions of system libraries, and use compiler optimisations for exotic environments. The same platform can offer a path to highly responsive deployment to support legacy services, since a unikernel can provide a gatekeeper function, as described further in D1.4.

4.2 Cloud Liquidity

Cloud liquidity revolves round the ability to migrate data, workloads and even complete virtual machines (VMs) between clouds in a dynamic fashion, free from provider lock-in and with no restrictions based on the underlying virtualisation technology. By enabling this we empower end users, allow dynamic responses to changing workloads and provide robustness in the face of disasters.

In Deliverable 3.2 we described the cloud liquidity use case in detail. We identified four primary motivations for cloud liquidity: avoiding provider lock in, in particular;

- allowing VM migration across different cloud infrastructure running different underlying virtualisation platforms
- allowing load balancing to reflect changing demands, be that in terms of following the sun/moon, responding to environmental changes or simply responding to changing customer demand
- providing high availability for mission critical systems
- providing wide area Disaster Recovery as a Service (DRaaS)

4.2.1 Use case overview

As explained in past deliverables, VM migration leverages three core components developed as part of the Trilogy2 project. These are:

- Inter-Cloud communication

- The Federated Market
- Wide Area block replication

These components combine to allow you to specify a VM and its workload, locate a new provider, create a new VM from a suitable template, transfer the current VM's data and then hand over control to the new VM. In D3.2 we described a simple version of this use case where an end-user uses VM migration across the Federated Market to transfer their VM from a cloud in London to another in Madrid. This involved a number of steps starting with finding a suitable new location, setting up the VM on that location, transferring the data using wide area block replication and then switching on the new VM. In the section below we describe the real-life implementation of this uses case as applied to Disaster Recovery as a Service.

4.2.2 Disaster Recovery as a Service

Disaster Recovery as a Service is a special case of VM migration. In a normal VM migration as soon as any data has been synchronised between the original VM and the new cloud then control is handed over and the original VM is powered down. In DRaaS data continues to be copied across asynchronously via a buffer on the origin VM. This means that the two VMs are permanently synchronised and in turn this means that if anything happens to the original VM then the end user can immediately fail over to the new VM without losing any data.
OnApp's DRaaS service was launched as a beta in May 2015 and the full version of the service will be launched in the first quarter of 2016. Since running two full VMs would be wasteful in terms of resource use, OnApp's DRaaS system uses a 'shadow' VM on the destination side. This simply handles the data being written across the network. As soon as the user fails over to this VM it is temporarily powered down, reconfigured as a full VM and powered on again.

4.2.2.1 The DRaaS Dashboard

Central to OnApp's DRaaS system is the Dashboard (see Figure 4.1). This provides a similar role to the Federated Market, providing a single point of trust and control for all DRaaS related operations. The dashboard has three classes of user: administrators; cloud owners (who may have provider clouds, source clouds or both); VM owners. Only administrators are allowed to register new cloud owners. Having been registered the cloud owner is then able to log in to the dashboard and register any clouds they want. This is done using the cloud's API key. If the cloud is a source cloud then part of the registration process is to choose which DRaaS provider cloud to replicate to. Once the registration is complete then an "Enable Disaster Recovery" option becomes available to the VM owner.

4.2.2.2 Enabling DRaaS

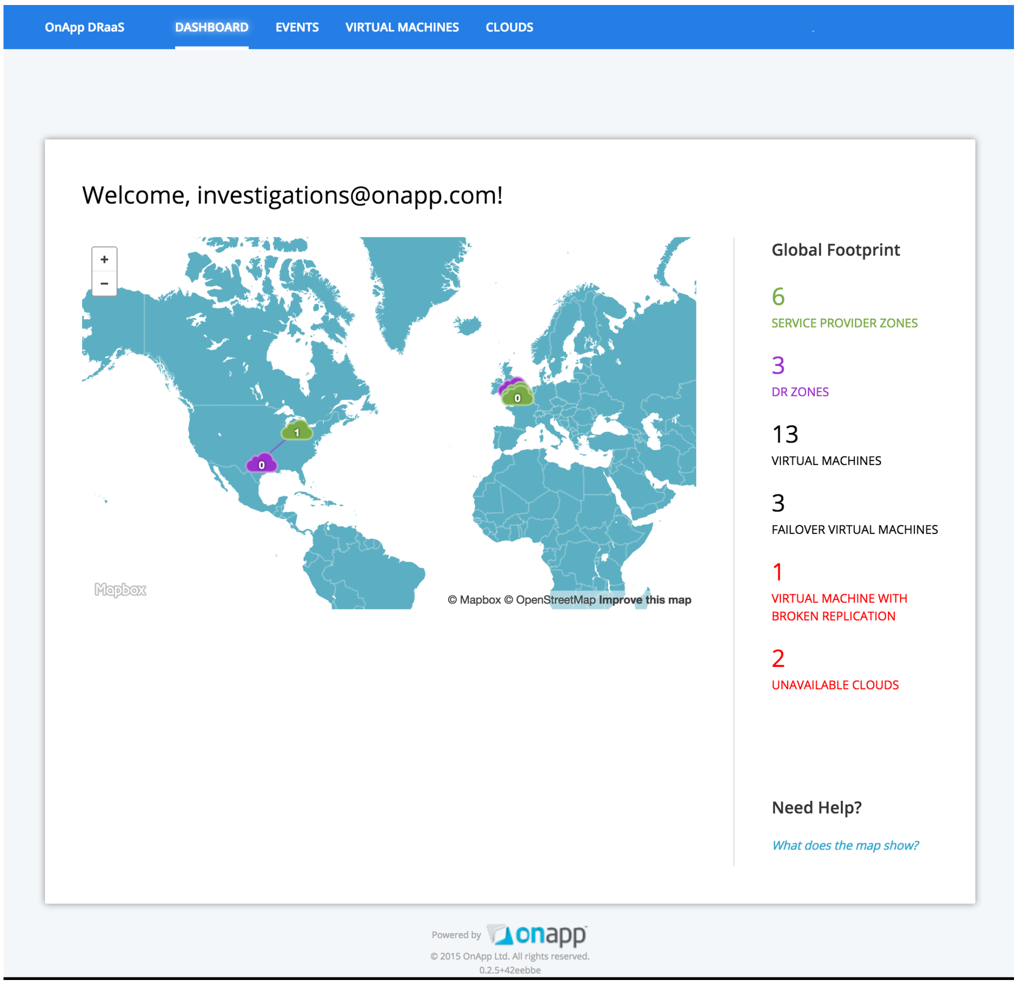
Enabling DRaaS on a specific VM triggers a number of operations. Firstly, a dashboard account is created for the VM owner. Then the VM is registered on the Dashboard and gets its own page. Then the dashboard logs into the provider cloud and creates the shadow VM for that cloud. Once the Shadow VM is running the source cloud opens a secured tunnel between the source VM's vDisk and the Shadow VM. It then transfers over all the current data. Once all data is synchronised then the dashboard will allow the fail over option. Meanwhile all new writes are copied across to the shadow VM.

4.2.2.3 Failing Over a VM

If a fail over is needed (e.g., if a disaster has made the original VM unresponsive, or if the cloud operator needs to perform major maintenance) then the VM owner is able to log into the VM's home page on the DRaaS Dashboard and click the "Failover" button. This triggers several things. First the Dashboard sees if it can cleanly shut down the original VM (as this will ensure any data in the buffer is flushed). Then the replication tunnel is removed and the Shadow VM is temporarily powered off. The Shadow VM is then upgraded to a full VM (with the same amount of RAM and number of vCPUs as the original) and the new VM is started up. The whole process takes a couple of minutes.

4.2.2.4 Failing Back a VM

As soon as the original cloud is stable again the VM owner is able to fail back the VM. Failback is a two stage operation. First you press "Synchronise Data" – this causes a new shadow VM to be created on the original cloud. If the original vDisk is still present then this can be associated, but the system assumes that all data may have been lost. Once the Shadow VM is running the data is synchronised back to the source. As soon as



Welcome, investigations@onapp.com!

Global Footprint

6
SERVICE PROVIDER ZONES

3
DR ZONES

13
VIRTUAL MACHINES

3
FAILOVER VIRTUAL MACHINES

1
VIRTUAL MACHINE WITH BROKEN REPLICATION

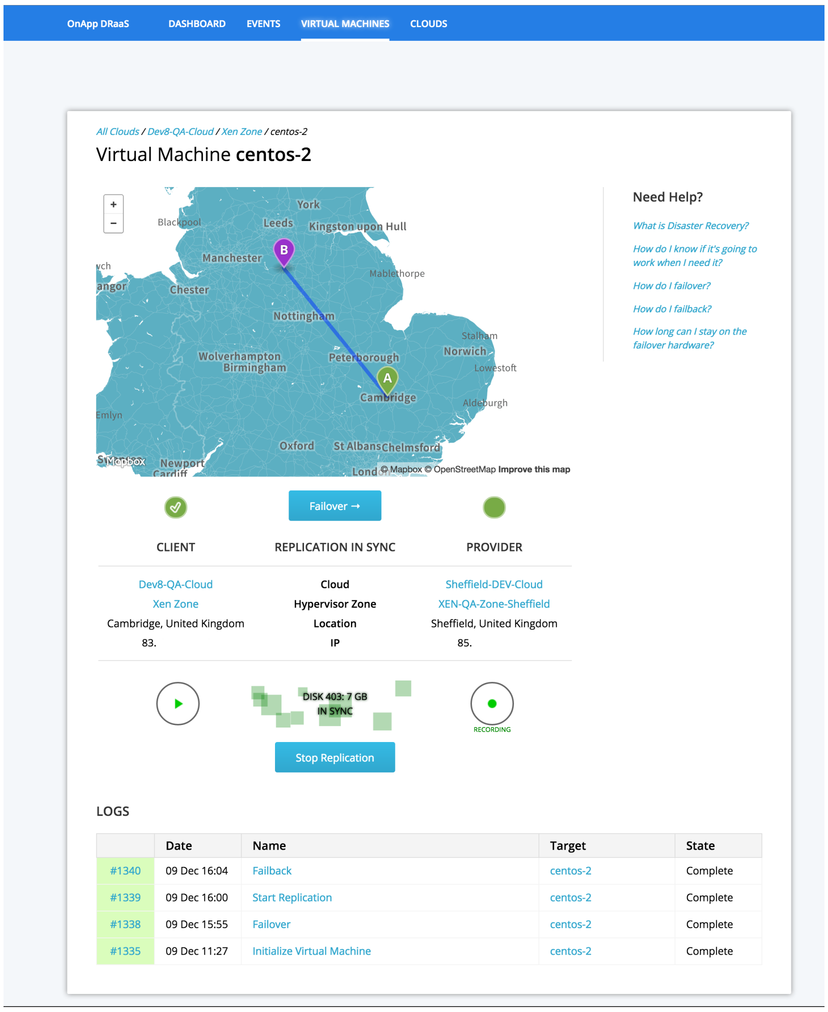
2
UNAVAILABLE CLOUDS

Need Help?

What does the map show?

© 2015 OnApp Ltd. All rights reserved.
0.2.5+42eebbe

Figure 4.1: The OnApp DRaaS dashboard.



All Clouds / *Dev8-QA-Cloud* / *Xen Zone* / *centos-2*

Virtual Machine **centos-2**

Need Help?

What is Disaster Recovery?

How do I know if it's going to work when I need it?

How do I failover?

How do I failback?

How long can I stay on the failover hardware?

CLIENT	REPLICATION IN SYNC	PROVIDER
Dev8-QA-Cloud	**Cloud**	Sheffield-DEV-Cloud
Xen Zone	**Hypervisor Zone**	XEN-QA-Zone-Sheffield
Cambridge, United Kingdom	**Location**	Sheffield, United Kingdom
83.	**IP**	85.

DISK 403: 7 GB
IN SYNC

LOGS

	Date	Name	Target	State
#1340	09 Dec 16:04	Failback	centos-2	Complete
#1339	09 Dec 16:00	Start Replication	centos-2	Complete
#1338	09 Dec 15:55	Failover	centos-2	Complete
#1335	09 Dec 11:27	Initialize Virtual Machine	centos-2	Complete

Figure 4.2: A VM homepage on the DRaaS dashboard

the data is in synch the VM owner is able to press the "Failback Now" button. This stops the Failover VM, removes replication, starts the full VM on the original source and finally re-establishes the replication in the other direction. The reason for the two stage process is to allow the user to verify that the original cloud is fully functional before they fail back to it.

4.2.2.5 Commercial Models

DRaaS will be launched as a full commercial product in February. It will offer several pricing models depending on the specific use:

- OnApp DRaaS: For small providers there will be the option to replicate to clouds managed by OnApp. For this there will be a monthly per-VM charge that reflects the size of the VM. Careful consideration has been given to how to enforce fair use of DRaaS. The enforcement model chosen is that VMs will be allowed to fail over for a certain amount of time per month but if they stay failed over for too long then there will be a potential surcharge.

- Private Mode: Cloud providers that operate multiple data centres will be charged a flat fee to be able to use the DRaaS technology to replicate between their own data centres. In this case it is up to the operator to police the use of failover.

- Federated Market: Longer term cloud providers will be able to trade as DRaaS Destination Providers on the Federated Market. This will allow them to set their own pricing model for offering the DRaaS service. This model should act as a market in economic terms, resulting in Cloud Liquidity becoming a commodity.

In the longer term the DRaaS mobility approach will be integrated into the Federated Market. The same storage and compute liquidity techniques will be used to enable VM mobility across the Federated Market and to extend the current DRaaS system to allow owners to failback to different locations.

4.2.3 DRaaS as an implementation of Trilogy 2 concepts

DRaaS is a working embodiment of several of the core ideas of the Trilogy2 project. DRaaS is a cross-layer implementation of Storage Liquidity that in turn enables Compute Liquidity across different providers. While it doesn't currently utilise any of the network liquidity tools that have been developed in this project ultimately the intention is to use techniques such as Secure MPTCP to provide better connectivity between the source VM and Shadow VM. This should serve to increase the speed of the initial replication as well as improving data security.

4.2.4 Summary

OnApp's Disaster Recovery as a Service is a real-world implementation of the Cloud Liquidity use case. It combines a number of concepts that have been developed in the Trilogy2 project into a working commercial product. Key to DRaaS are the concepts of storage and compute liquidity. DRaaS is provider agnostic, only requiring that both the source and destination clouds run OnApp's distributed storage system.

5 Conclusions

In this deliverable we described the deployment of Trilogy2 use cases in testbeds and real operational networks, and presented extensive evaluation results from these. In all, the project has created a wide range of liquidity tools (MPTCP, Minicache, ClickOS, Irmin and federated clouds, to name a few) that target a number of different areas: wide area networks, operator infrastructure and mobile devices. We believe that these contributions to the wider research community and industry fields will help enhance liquidity in future networks. In some cases, as with MPTCP, this is already happening in current networks; it is our hope that this and other tools created in this project will continue to proliferate.

Bibliography

[1] 3GPP. IP Multimedia (IM) Subsystem Cx and Dx Interfaces; Signalling flows and message contents. TS 29.228, 3rd Generation Partnership Project (3GPP), Oct 2014.

[2] T. Bates, Y. Rekhter, R. Chandra, and D. Katz. Multiprotocol Extensions for BGP-4$. Technical Report 2858, IETF Secretariat, June 2000.

[3] Meeyoung Cha, Pablo Rodriguez, Jon Crowcroft, Sue Moon, and Xavier Amatriain. Watching television over an ip network. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08, pages 71–84, New York, NY, USA, 2008. ACM.

[4] Yung-Chih Chen, Yeon-sup Lim, Richard J Gibbens, Erich M Nahum, Ramin Khalili, and Don Towsley. A measurement-based study of multipath tcp performance over wireless networks. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 455–468. ACM, 2013.

[5] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. Poster: Evaluating android applications with multipath tcp. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 230–232. ACM, 2015.

[6] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. Wifi, lte, or both?: measuring multi-homed wireless internet performance. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 181–194. ACM, 2014.

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

[8] Erlang on Xen. Erlang on Xen. `http://erlangonxen.org/`, July 2012.

[9] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A first look at traffic on smartphones. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 281–287. ACM, 2010.

[10] Simone Ferlin, Thomas Dreibholz, and Özgü Alay. Multi-path transport over heterogeneous wireless networks: Does it really pay off? In *Proceedings of the IEEE GLOBECOM*, Austin, Texas/U.S.A., December 2014. IEEE.

[11] S. Ferlin-Oliveira, T. Dreibholz, and O. Alay. Tackling the challenge of bufferbloat in multi-path transport over heterogeneous wireless networks. In *Quality of Service (IWQoS), 2014 IEEE 22nd International Symposium of*, pages 123–128, May 2014.

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

[13] 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.

[14] Hanquez V Gazagnaire T. OXenstored an efficient hierarchical and transactional database using functional programming with reference cell comparisons. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, pages 203–214, 2009.

[15] Aaron Gember, Ashok Anand, and Aditya Akella. A comparative study of handheld and non-handheld traffic in campus wi-fi networks. In *Passive and Active Measurement*, pages 173–183. Springer, 2011.

[16] Domenico Giustiniano, Eduard Goma, Alberto Lopez Toledo, Ian Dangerfield, Julian Morillo, and Pablo Rodriguez. Fair wlan backhaul aggregation. In *Proceedings of the Sixteenth Annual International Conference on Mobile Computing and Networking*, MobiCom '10, pages 269–280, New York, NY, USA, 2010. ACM.

[17] Benjamin Hesmans and Olivier Bonaventure. Tracing Multipath TCP connections. *SIGCOMM Comput. Commun. Rev.*, 44(4):361–362, Aug 2014.

[18] Benjamin Hesmans, Hoang Tran-Viet, Ramin Sadre, and Olivier Bonaventure. A first look at real Multipath TCP traffic. In Moritz Steiner, Pere Barlet-Ros, and Olivier Bonaventure, editors, *Traffic Monitoring and Analysis*, volume 9053 of *Lecture Notes in Computer Science*, pages 233–246. Springer International Publishing, 2015.

[19] Junxian Huang, Qiang Xu, Birjodh Tiwana, Z Morley Mao, Ming Zhang, and Paramvir Bahl. Anatomizing application performance differences on smartphones. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 165–178. ACM, 2010.

[20] ITU-T. Network grade of service parameters and target values for circuit-switched services in the evolving ISDN. Recommendation E.721, Telecommunication Standardization Sector of International Telecommunication Union (ITU-T), May 1999.

[21] Srikanth Kandula, Kate Ching-Ju Lin, Tural Badirkhanli, and Dina Katabi. FatVAP: aggregating AP backhaul capacity to maximize throughput. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 89–104, Berkeley, CA, USA, 2008. USENIX Association.

[22] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. Mptcp is not pareto-optimal: Performance issues and a possible solution. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 1–12, New York, NY, USA, 2012. ACM.

[23] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, Jun 2014. USENIX Association.

[24] Yeon-sup Lim, Yung-Chih Chen, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. How green is Multipath TCP for mobile devices? In *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, &#38; Challenges*, pages 3–8. ACM, 2014.

[25] Ioana Livadariu, Simone Ferlin, Ozgu Alay, Thomas Dreibholz, Amogh Dhamdhere, and Ahmed Mustafa Elmokashfi. Leveraging the IPv4/IPv6 Identity Duality by using Multi-Path Transport. In *Proceedings of the 18th IEEE Global Internet Symposium (GI)*, Hong Kong/People's Republic of China, Apr 2015.

[26] M. Luckie, B. Huffaker, k. claffy, A. Dhamdhere, and V. Giotsas. AS Relationships, Customer Cones, and Validation. In *Internet Measurement Conference (IMC)*, pages 243–256, Oct 2013.

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

[28] R. Mahy, B. Biggs, and R. Dean. The session initiation protocol (sip) "replaces" header, September 2004. RFC3891.

[29] P. Manzoni, D. Ghosal, and G. Serazzi. A simulation study of the impact of mobility on tcp/ip. In *Network Protocols, 1994. Proceedings., 1994 International Conference on*, pages 196–203, Oct 1994.

[30] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, Apr 2014. USENIX Association.

[31] Marco Mellia, Andrea Carpani, and Renato Lo Cigno. Tstat: Tcp statistic and analysis tool. In *Quality of Service in Multiservice IP Networks*, pages 145–157. Springer, 2003.

[32] Netflix. Can I stream Netflix in Ultra HD? `https://help.netflix.com/en/node/13444`, June 2015.

[33] Anthony J. Nicholson, Scott Wolchok, and Brian D. Noble. Juggler: Virtual networks for fun and profit. *IEEE Transactions on Mobile Computing*, 9(1):31–43, Jan 2010.

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

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

[36] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of multipath tcp schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, pages 27–32. ACM, 2014.

[37] Christoph Paasch, Ramin Khalili, and Olivier Bonaventure. On the benefits of applying experimental design to improve Multipath TCP. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 393–398, New York, NY, USA, 2013. ACM.

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

[39] Quagga Routing Software Suite. `http://www.nongnu.org/quagga/`, jun 2015.

[40] C. Raiciu, D. Niculescu, M. Bagnulo, and M. Handley. Opportunistic Mobility with Multipath TCP. In *ACM MobiArch 2011*, 2011.

[41] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

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

[43] Rutube. From Zero to 700 Gbit per Second – How One of the Russia's Largest Video-Hosting Service Uploads its Videos [S nulya do 700 gigabit v secundu kak otgruzhaet video odin iz krupneishih video-hostingov Rossii]. `http://habrahabr.ru/company/rutube/blog/269227/`, Oct 2015.

[44] SungHoon Seo. Kts giga lte. Presented as the 93th IETF in Prague, Czech Republic, 2015.

[45] G.J. Sullivan, J. Ohm, Woo-Jin Han, and T. Wiegand. Overview of the high efficiency video coding (hevc) standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668, Dec 2012.

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

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

[48] N. Williams, P. Abeysekera, N. Dyer, H. Vu, and G. Armitage. Multipath TCP in Vehicular to Infrastructure Communications. Technical Report 140828A, CAIA, Swinburne University of Technology, August 2014.

[49] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 8–8, Berkeley, CA, USA, 2011. USENIX Association.