This project has received funding from the European Unions Horizon 2020
research and innovation programme under grant agreement No 688421.

Measurement and Architecture for a Middleboxed Internet

H2020-ICT-688421

Final Middlebox Model, Experimentation and Evaluation Report

Author(s):	ETH	Tobias Bühler, Brian Trammell, Mirja Kühlewind
	ULg	Benoit Donnet, Korian Edeline, Justin Iurman
	UNIABDN	Gorry Fairhurst, Ana Custura, Tom Jones
	ZHAW	Stephan Neuhaus
	SRL	David Ros (ed.)
	ALCATEL	Thomas Fossati
	UC3M	Pedro A. Aranda

Document Number:	D2.2
Internal Reviewer:	Stephan Neuhaus
Due Date of Delivery:	31 December 2018
Actual Date of Delivery:	21 December 2018
Dissemination Level:	Public

Disclaimer

The information, documentation and figures available in this deliverable are written by the MAMI consortium partners under EC co-financing (project H2020-ICT-688421) and does not necessarily reflect the view of the European Commission.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user uses the information at its sole risk and liability.

Contents

List of Acronyms

ECDF Empirical Cumulative Density Function

MAMI Measurement and Architecture for a Middleboxed Internet

RTT Round-Trip Time

TCP Transmission Control Protocol

UDP User Datagram Protocol

CBR Constant Bitrate

DHCP Dynamic Host Configuration Protocol

DNS Domain Name Service

DPDK Data Plane Development Kit

eNB e-Node B

ETSI European Technical Standards Institute

IRTF Internet Research Task Force

KPI Key Performance Indicator

LLT Loss/Latency Trade-off

LoLa Loss/latency tradeoff

LTE Long–Term Evolution

NEMO Network Modelling

NFV Network Functions Virtualisation

NSD Network Service Descriptor

ODL OpenDaylight

OS Operating System

OSM Open Source MANO

OSS Operations Support System

QoE Quality of Experience

QoS Quality of Service

RAN Radio Access Network

RO Resource Orchestrator

SDN Software-defined Networking

SW Software

TFT Traffic Flow Template

UE User Equipment

VIM Virtual Infrastructure Manager

VM Virtual Machine

VNF Virtual Network Function

VNFC VNF Component

VNFD VNF Descriptor

VPP Vector Packet Processing

Executive Summary

Work Package 2 had two high-level goals: (1) To develop a taxonomy and a behavioural model of Internet middleboxes. Taking measurements performed in WP1 as inputs, this would provide a better understanding of the behaviour and impairments to be expected from middleboxes in the wild, and a basis for designing and testing of protocol mechanisms embodying the middlebox-cooperation architecture developed in WP3. (2) To evaluate the applicability and deployment feasibility of such mechanisms by means of experiments, not only in lab settings but also using Internet testbeds such as the MONROE mobile broadband platform.

This deliverable provides a report on the WP2 activities carried out during the last reporting period (July 2017–December 2018). Some of these activities wrapped up work previously reported in Deliverable D2.1 that targeted the first high-level goal, whereas other activities focused on experiments to evaluate different protocol mechanisms reported in Deliverable D3.3. These experiments, targeting the second high-level goal of WP2, allowed us to assess various performance and functional aspects of such mechanisms. In particular, the viability of both the Path MTU discovery for UDP applications and passive RTT measurement approaches was validated by experiments run across arbitrary Internet paths.

1 Introduction

This document is the final report on the work done in Work Package 2 of the MAMI project, "Experimentation: Middlebox Modeling and Testing", as part of the three WP2 tasks that were active during the final reporting period (July 2017–December 2018), i.e., T2.2 (Middlebox modeling), T2.3 (Model/NFV-based experimentation) and T2.4 (Testbed-based validation of approach). The deliverable summarises the main outcomes and findings of this WP in this reporting period.

Section 2 presents the design of a modular, easy-to-configure middlebox simulator, `mmb`, developed as part of Task T2.2. We describe its architecture and provide experimental results obtained in a controlled lab environment.

Several of the activities and results presented next correspond to the experimental evaluation of protocols or protocol extensions, based on explicit path-to-sender or sender-to-path signaling, that were developed in WP3 and which are described in Deliverable D3.3. These results are summarised in Section 3, reporting on work done as part of Task T2.4 as follows:

- An algorithm and methods to discover the maximum Path MTU for UDP-based applications (Section 3.1). Datagram Packetization Layer Path MTU Discovery (DPLPMTUD), described in section 3.2.2 of D3.3, was implemented then evaluated both in the lab and with real-world tests across Internet paths.
- An experimental assessment of three passive latency measurement approaches (Section 3.2). Two of these approaches, the latency spin signal in QUIC and TCP and the Packet Serial Number/Echo in PLUS, are described in section 3.1.3 of D3.3 and in section 1.3.1 of D3.2, respectively. The experiments used both lab setups as well as real Internet paths, including over the MONROE platform.
- A comparative evaluation of the Loss-Latency tradeoff (LoLa) mechanism, introduced in section 3.1.2 of D3.3, and alternative approaches for supporting a low-latency service in the network (Section 3.3).

Section 4 outlines Task T2.3 work on preparing several measurement and experimentation components developed in MAMI for virtualised deployments. The section also discusses extensions to the Network Modeling (NEMO) language to accommodate recursive VNF Descriptors. Finally, Section 5 concludes the document.

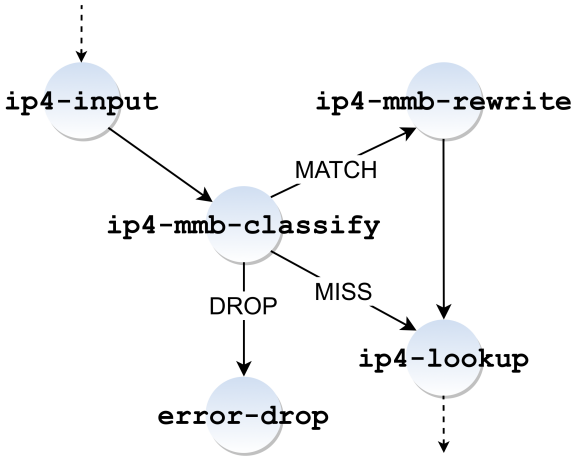


Figure 1: VPP nodes for `mmb`.

2 VPP Middlebox Simulator

VPP (Vector Packet Processor [4]) is a kernel-bypassing framework for building high-speed data plane functionalities. It leverages techniques such as batch-processing, Receive-side Scaling (RSS) queues, and Zero-Copy by allowing userspace applications to have Direct Memory Access (DMA) to the memory region used by the NIC and I/O batching to reduce the overhead of NIC-initiated interrupts. While those techniques have been implemented in other kernel-bypassing frameworks (e.g., FastClick [6], MiddleClick [5]) and have been shown to drastically improve performance [5], VPP attempts to surpass those by introducing particular coding practices (e.g., branch prediction, memory prefetching, cache-fitting processing nodes) to maximize low-level parallelism and cache locality.

This chapter introduces `mmb` (**M**odular **M**iddle**B**ox), a VPP plugin that performs stateless and stateful classification and rewriting based on the middlebox policies taxonomy previously introduced by the MAMI project [12]. It achieves stateless packet matching based on any combination of constraints on network or transport protocol fields, stateful TCP and UDP flow matching, packet mangling, packet dropping and bidirectionnal mapping. `mmb` is protocol-agnostic by allowing to match and rewrite fields `ip4-payload`, `udp-payload`, and `tcp-opt`.

2.1 Architecture

`mmb` consists in two nodes, a *classification* and a *rewrite* node, as shown in Fig. 1. When `mmb` is enabled, its nodes are connected to the processing graph. The classification node is placed right after the `ip4-input` (or `ip6-input`) node, that validates the IPv4 header checksum, verifies its length and discards packets with expired TTLs.

Depending on the outcome of the classification step (that can either be *match*, *miss*, or *drop*), packets are forwarded respectively to the `mmb-rewrite`, `ip4-lookup` or `error-drop` nodes.

The role of the `ip4-lookup` node is to perform the Forwarding Information Base (FIB) lookups, and then dispatch packets to the corresponding processing path. In the case of a middlebox,

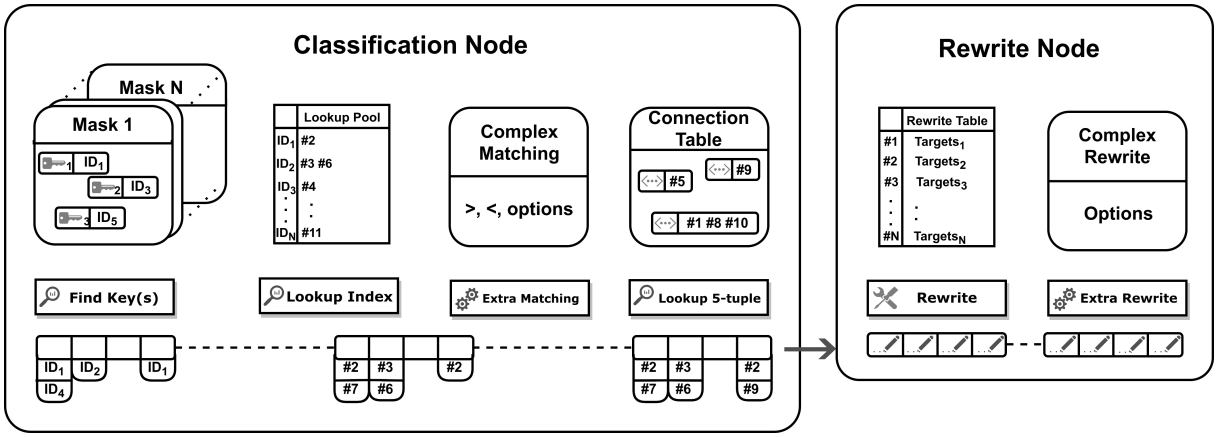


Figure 2: `mmb` processing path.

it would be the `ip4-rewrite` (or `ip6-rewrite`) node, that performs the MTU computation, and updates the TTL and checksums.

Overall, `mmb` consists in three processing paths which can each be traversed or not by packet vectors, depending on the input policies (Fig. 2). A fast path relies on VPP bounded index hash tables and implements the mask-based matching operation using binary operators. This path is enabled when a rule without any TCP options nor IPv6 Extension Header is entered. Moreover, it restricts the conditions to **==** (isequal). The stateful flow matching is using this fast path. A first slow path is used for rules with complex conditions ($\neq$, $<$, $>$, $\leq$, $\geq$), and a second slow path for the linked list parsing required when classifying based on TCP options or IPv6 Extension Headers, as well as when rewriting them.

2.1.1 CLI Grammar

One of the main goals of `mmb` is to be easily configurable, generic, and to allow defining various middlebox policies [12]. To this end, we define the following grammar [1]:

`mmb <add-keyword> <match> [<match> ...]  <target> [<target> ...]`

- `<add-keyword>`: `add|add-stateless|add-stateful`

  The keyword determines if `mmb` has to keep track of connections matching a given rule or not. `add/add-stateless` rules apply their targets at the packet level, each packet has to match the rule in order to apply the targets. `add-stateful` rules apply their targets at the connection/flow level. The `<match>` list of a stateful rule is used to add entries to the connection table. Once a connection is added to the table, the `<target>` list is applied to all packets of the connection, even if they do not match the rule. Additionally, `add-stateful` allows for special targets `map` and `shuffle`.

- `<match>` : `[!] <field> [[<condition>] <value>]`

[1] The complete list of available fields and commands is available on the `mmb` repository [13].

This parameter is a constraint that determines the packets on which the rule will operate.

If a `<match>` is composed of a `<field>` alone, the constraint is that the packet should contain the field. If it is composed of a `<field>` and a `<value>`, the constraint is that the packet should contain the field and it should be set to the specified value. If it is composed of a `<field>`, a `<condition>`, and a `<value>`, then the constraint is that the packet should contain the field, and the condition on the value should be true.

The ! operator applied on one constraint performs the logical NOT of the constraint. Multiple constraints can be inputted for the same rule, the resulting constraint is the logical AND of all inputted constraints.

- `<target>` : `mod [...]|strip [...]|add [...]|drop [...]|map [...]|shuffle [...]`

  This parameter determines the action(s) to apply on matched packets.

  - `mod <field> <value>`
    Modify a field on a packet.
  - `add <field> <value>`
    Add a tcp-opt to the packet.
  - `strip [!] <field>`
    Strip options from a packet.
    If the ! operator is placed after the strip keyword, the following option will be added to the whitelist (the only authorized options), if not it will be added to the blacklist (the forbidden options).
    The special keyword `all` can be used in a `strip` target to strip all options from the matched packet.
  - `drop [<rate>]`
    Drop a packet with optional probability `<rate>` given as a percentage, with a maximal precision of 0.01%. The default rate is 100%.
  - `map <field> <value>`
    Perform a bidirectionnal mapping of the given `<field>` to a given value. Valid fields are: `ip-saddr`, `ip-daddr`, `ip6-saddr`, `ip6-daddr`, `tcp-sport`, `tcp-dport`, `udp-sport`, `udp-dport`, `ip-id`, `ip6-flow-label`.
  - `shuffle <field>`
    Perform a bidirectionnal mapping of the given `<field>` to a random value. Valid fields are: `tcp-seq-num`, `tcp-ack-num`, `tcp-sport`, `tcp-dport`, `udp-sport`, `udp-dport`, `ip-id`, `ip6-flow-label`.

2.1.2 Classification

`mmb` packet processing is displayed in Fig. 2. The classification node consists in four distinct steps: a mask-based constraint matching step, an index lookup pool, a complex matching step, and a connection table.

The mask-based matching determines if each packet satisfies constraints on fixed offset fields. For this, we create one classification table per packet mask (e.g., per combination of fields in

$$Result_{Classif.} = (Packet \ \& \ Mask) \ \oplus \ Key \tag{2.1}$$

$$Result_{Rewrite} = (Packet \ \& \ Mask) \mid Key \tag{2.2}$$

Figure 3: Binary operations for packet classification and rewrite.

the match constraint), sized from 16 bytes to at most 80 consecutive bytes. We create one key for a given table per value for its associated packet mask. For each table, the search for a key matching a given packet is a dichotomic hash-based search with a logarithmic complexity.

The matching operation consists of two binary operations (AND and XOR), as shown in Eqn. 2.1 (see Fig. 3), which are applied to consecutive chunks of 16 bytes, starting from the first nonzero byte in the mask. Results are OR'ed into a 16-byte variable, that is compared to zero to verify if the matching operation was successful. This operation is illustrated in Algorithm 1.

Then, for each packet that matched at least one mask-key combination, `mmb` checks if an additional matching is needed, with a constant-time lookup, and performs it. Additional matching is necessary for constraints on linked-list based fields such as TCP options and IPv6 Extension Headers.

Finally, each packet is matched to a connection table via its 5-tuple. The connections table keeps track of every connection that matched at least one stateful rule, and implements a flag tracking and a timeout mechanism. This allow for, e.g., reflexive policies.

If a packet successfully matches at least one rule with a drop target, it is immediately forwarded to the `error-drop` node. If the packet matches only non-drop rules, it is forwarded to the `mmb-rewrite` node, if the packet does not match any rule, it is handed to the next non-`mmb` node, i.e., `ip4-lookup`.

Algorithm 1 Matching operation

```
function MATCH(pkt, mask, key, skip, chunks)
    res ← (pkt[skip] & mask[0]) ⊕ key[0]
    switch chunks do
        case 5
            res ← res | ((pkt[skip + 4] & mask[4]) ⊕ key[4])
        case 4
            res ← res | ((pkt[skip + 3] & mask[3]) ⊕ key[3])
        case 3
            res ← res | ((pkt[skip + 2] & mask[2]) ⊕ key[2])
        case 2
            res ← res | ((pkt[skip + 1] & mask[1]) ⊕ key[1])
        case 1
            break
        default
            abort()
    if zero_byte_mask(res) = 0xffff then
        return 1
    else
        return 0
```

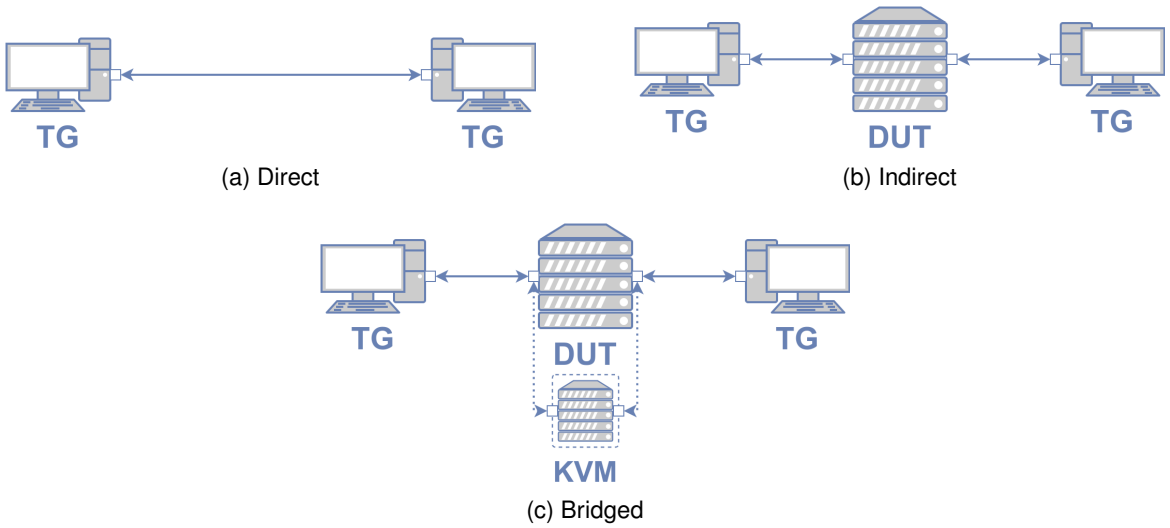


Figure 4: Measurement Setups. TG = Traffic Generator. DUT = Device Under Test. Plain arrows are physical connections, Dotted arrows are bridge networks and the machine surrounded by dots is a virtual machine.

2.1.3 Rewrite

The `mmb-rewrite` node consists in two operations: a mask-based rewrite step that works on the fixed offset fields, similarly to the first step of the classification node, and a complex rewrite step for linked-list based fields.

To perform the rewrite operation, or application of targets, we build a target mask and a target key when the rule is added. The rewrite is then performed with two binary operations (AND and OR), as shown in Eqn. 2.2 (see Fig. 3).

2.2 Performance

In this section, we compare the performance of `mmb` in several use cases with respect to other state-of-the-art tools. In addition, we discuss `mmb` performance in realistic use cases.

2.2.1 Measurement Setup

Our testbed is composed of three machines with Intel Xeon CPU E5-2620 v4 @ 2.10GHz 8 Cores 16 Threads, 32GB RAM, running Debian 9.0 with 4.9 kernels. Each machine has a 2 port Intel XL710 40GB QSFP+ NIC with Receive-Side Scaling enabled (RSS), which is connected to a Huawei CE6800 switch using one port each for traffic generators and both ports for the device under test.

The Device Under Test (DUT) runs VPP 18.10, DPDK 18.08 with 20 1GB transparent hugepages, and a kvm/QEMU 2.8.1 hypervisor with a Ubuntu 18.04 guest. The Traffic Generators (TGs)

run iperf3, nginx 1.10.3 and wrk 4.0.2.

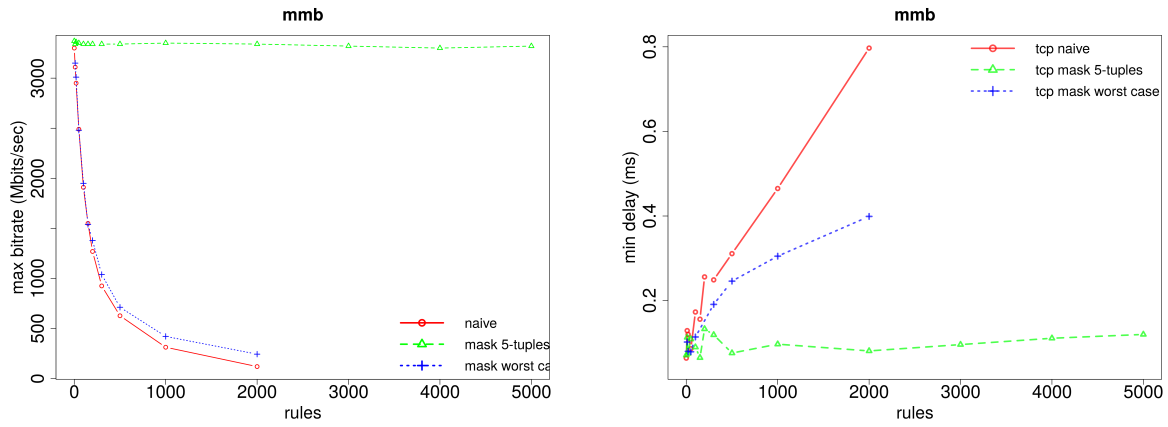
The device under test is configured to maximize its performance. The scaling governor is set to run the CPU at the maximum frequency. 15 threads out of 16 are isolated from the kernel scheduler to make sure that no other tasks are being run on the same physical CPUs, and pinned to the process under test. We enable adaptative-ticks CPUs to omit unnecessary scheduling-clock ticks for CPUs with only one runnable task, which we ensure by setting the CPU affinity for VPP, and we enable RCU callback offloading.

We configured our testbed into three different setups: (1) a *direct* client-to-server communication setup, shown in Fig. 4a, that is used to evaluate bandwidth baselines and rule out sender-bounded experiments; (2) an *indirect* setup (Fig. 4b) in which the DUT forwards traffic between sender and receiver by running code on its host OS; (3) a *bridged* setup (Fig. 4c) where the guest OS interfaces are connected to the host OS interfaces using two bridges.

Given that iperf is bounded to a single CPU, we have to make sure that we measure the performance of the DUT and not the TGs. To this end, we run a single pair of iperf client-server using the direct setup, and we add iperf client-server pairs until the bandwidth reaches the maximum capacity. We found that a single iperf pair would not exceed a threshold bandwidth of 18 Gbps. Therefore, the following experiments are performed with at least 3 iperf pairs. This allows us to reach a consistent 37.7Gbps of bandwidth, which is close to the maximum capacity of the NICs. The wrk+nginx traffic generators consist in one TG running nginx, hosting files of different sizes (1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, and 256KB), and the other TG running 16 wrk threads, each opening 128 connections and transferring the same file. We notice that, when transferring files from 1KB to 32KB, the bandwidth is linearly increasing. From the 32KB file, the bandwidth reaches a threshold and then stabilises for experiments transferring files from 64KB to 256KB. We arbitrarily choose to transfer the 128KB file for all wrk+nginx experiments. For both iperf and wrk+nginx traffic generation, the experiments last for 20 seconds and omit the first second, in which TCP is still running the slow start algorithm. Each experiment result is averaged over 1,000 runs.

First, we evaluated the performance gain of the mask-based approach (see Fig. 5). The red curve is the naive classification approach, where each packet is parsed and each field is sequentially checked for matching to each rule. The green curve is `mmb` filled with rules matching solely on five-tuples, which is the best possible case for the mask-based approach, allowing to use a single classification table. The blue curve is the worse possible case of `mmb`, in which each input rule deliberately matches on different field combinations, forcing it to use one table for each single rule. All input rules are matching on five fields. The blue curve stops at 2,000 rules because the maximum number of masks is bounded by the number of fields. In this experiment, each rule defined matching constraint on 5 fields from 14, which gives us a bound of $C(14, 5) = 2,002$ masks. The red curve also stops at 2,000 rules because the additional delay caused by the naive processing causes certain iperf client processes to hang indefinitely, which makes the experiments inconvenient to complete.

We observe that the green curve is not suffering in term of bandwidth when adding 5,000 rules (see Fig. 5a). On the other hand, the performance in both the “naive” and “mask-based worst case” scenarios (i.e., red and blue curves, respectively) drops significantly fast when the number of rules increases. The mask-based worse case is slightly better than the naive approach. We explain it by the fact that the latter has to loop over each field separately, while the mask-based matches all five fields at once. The latencies (see Fig. 5b) follow a very similar trend, where red and blue curve display a substantial increase of the delay when matching many



(a) Max Bitrate

(b) Min Delay

Figure 5: Performance in Virtualized Setup. Red = naive packet parsing classification, green = `mmb` +VPP with rules exclusively classifying packets based on their 5-tuples, blue = `mmb` +VPP with each single rule matching on different combinations of fields.

rules. The delay increases more than three times when matching 2,000 rules with different masks.

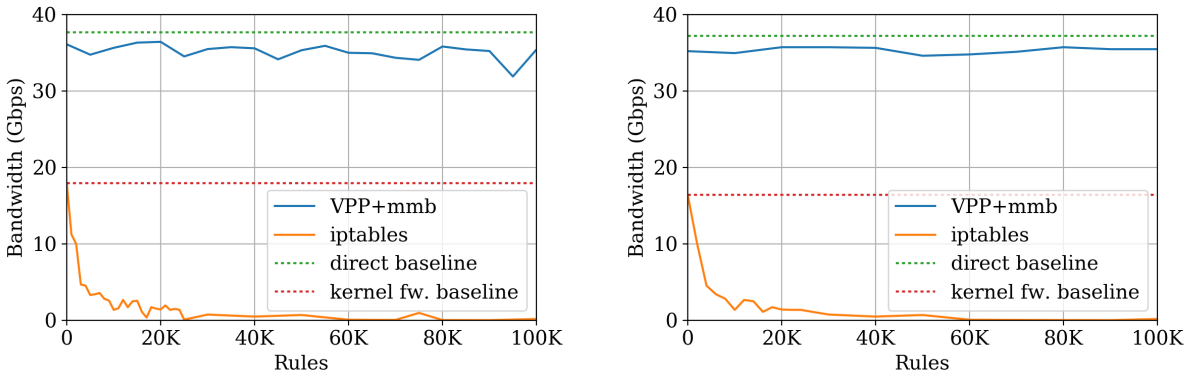
This shows that overhead of the mask-based approach (i.e., green curve) is constant with respect to the number of rules. We believe that the performance of `mmb` in real-life use cases, if the rules are designed carefully, will follow this curve.

2.2.2 Firewall

For these experiments, we configured `mmb` as a firewall and compared it to kernel forwarding and iptables. To do this, we generated `mmb` stateless rules that classify packets based exclusively on five-tuples, to ensure that it only uses one single table. We also make sure that no rules are matching the traffic from the TG. The results are shown in Fig. 6.

The *direct* baseline consists in two machines communicating directly, and is used to evaluate the TG bottleneck bandwidth. For the iperf experiment, we observed a *direct* baseline bandwidth of 37.7 Gbps, and for the wrk+nginx, 37.2Gbps. Then, we evaluated the *indirect* baseline by running the exact same experiment with the testbed configured following the *indirect* setup (see Fig. 4). This experiment allows us to measure the forwarding performance of the Linux kernel. During the iperf experiment, traffic was forwarded at 18Gbps, and during the wrk+nginx experiment, at 16.4bps.

We then injected to iptables and `mmb` from 0 to 100,000 rules, as described earlier. For the iperf experiment, the bandwidth achieved while checking packet five-tuples using iptables is higher than 10Gbps until 2,200 rules are injected. After this point, the performance with iptables very poor, from 20,000 rules on the bandwidth never reached more than 2Gbps. We observe a similar behavior for the nginx+wrk experiment during which the bandwidth went under 10Gbps at 1,600 rules and kept decreasing.



(a) Traffic generated with iperf.

(b) Traffic generated with wrk and nginx.

Figure 6: Firewall 5-tuple filtering, *indirect* setup. Dotted-line experiments do not increase the rule count.

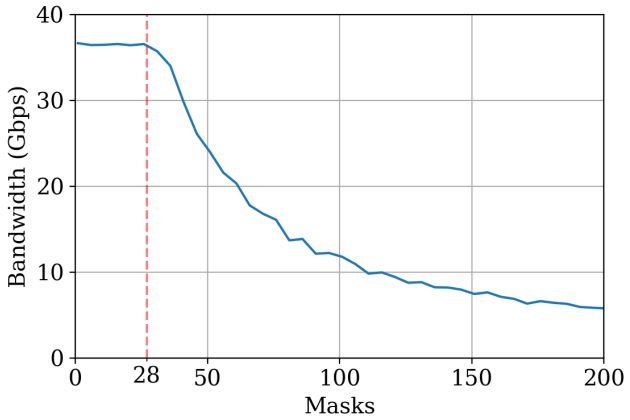
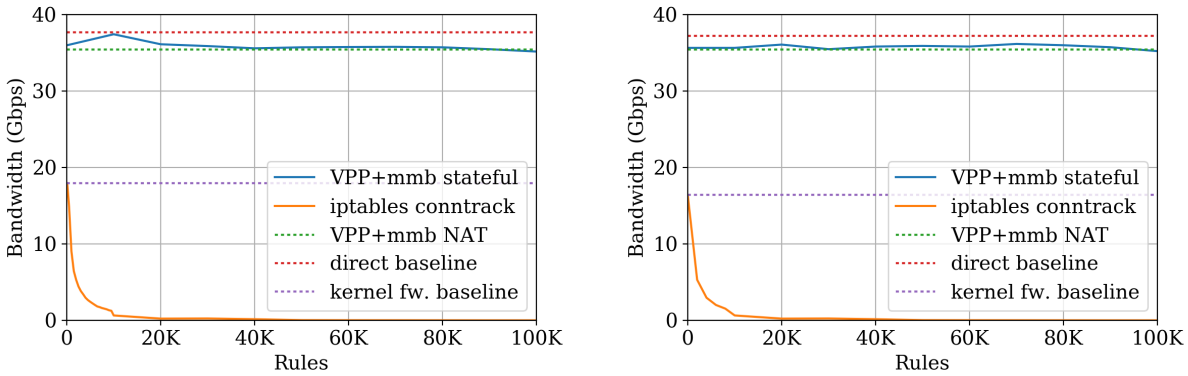


Figure 7: Different masks for each rule, *indirect* setup, iperf traffic.

We tested `mmb` the same way, and found that from 0 to 100,000 rules, the achieved bandwidth for the iperf experiment is oscillating between 33 and 37Gbps, and for the nginx+wrk experiment, between 35 and 36Gbps. Both experiments are roughly 1Gbps slower than the direct baseline, which corresponds to the overhead introduced by the forwarding process. The firewall seems to introduce no additional overhead. We made a few experiments with 500K and 1M rules and found no significant performance reduction. We did not conduct a complete test matrix to 1M rule because the process of adding a large amount of rules is very slow. This experiment indicates that the mask-based *fast path*, when relying on a single table, has a very limited impact on the maximum achievable bandwidth of the forwarding device.

We conducted another experiment to observe the performance when using multiple masks. We generated rules similarly to the previous experiment, but instead of matching exclusively on five-tuples, we forced each rule to match on a different combination of fields, thus using a different table. As shown in Fig. 7, performance remains stable until 28 masks, then decreases



(a) Traffic generated with iperf.

(b) Traffic generated with wrk and nginx.

Figure 8: Stateful connection tracking, *indirect* setup, Dotted lines experiments do not increase the rule count.

fast. We believe that we can increase this limit with more careful cache management. However, it is very likely that realistic use cases would not require this amount of masks.

Finally, we evaluated the performance of `mmb` flow tracking. To this end, we added stateful rules that match on the five-tuples with an `accept` target, to make sure that no modifications are applied to packets. Moreover, we made sure that each iperf or wrk TCP flow is matched by at least one rule, to make sure that the flow tracking module is called for every processed packet. For the wrk experiment, we added an extra rule matching on the three-tuple (server address, client address and transport protocol). The NAT experiment consists of a single stateful rule matching all outgoing packets and rewriting the source address and the source port (i.e., a SNAT). The iptables conntrack experiment is similar to the stateful iperf experiment, and enables the conntrack module of iptables. The results are shown in Fig. 8. For the stateful experiment, it shows the absence of correlation between the bandwidth and the number of rules. Moreover, we observe a slight under-performance of the NAT experiment, 300Mbps when averaged on all runs, compared to the stateful experiment for both iperf and wrk+nginx. This is likely to be caused by the `mmb` rewrite node. Finally, we see that the performance of the iptables conntrack module is worse than that of the iptables stateless firewall. The measured bandwidth when packets cross the iptables conntrack and the stateless firewall falls under 10Gbps with respectively more than 1,600 and 2,200 rules.

2.3 Conclusion

This chapter introduced `mmb` (**M**odular **M**iddle**B**ox), a VPP plugin implementing the middlebox policies taxonomy previously introduced in the MAMI project [12]. `mmb` provides the basis for more extensive evaluation of new protocols with a large set of middlebox impairments that may hinder deployment on the Internet.

We have demonstrated that operations performed by `mmb` over packets are efficient compared to the state of the art.

One of the `mmb` key points is its ease of use. Middleboxes can be easily built, in real time, based on a CLI interface. In addition, we provide `mmb` with a GUI allowing to easily setup network topologies including various network devices (servers, middleboxes, switches, routers). Those topologies might be used for scientific purposes (e.g., testing the performance of new protocoles with respect to path impairment introduced by middleboxes) but also for pedagogical purposes (e.g., allowing students to build and assess middleboxes impacts during an advanced networking course).

3 Experiments on Middlebox Cooperation Approaches

The MAMI project performed experimental evaluations of a set of middlebox cooperation and path signaling mechanisms, proving the feasibility and benefits of these example signals. As each such signal has different properties and practical implications, evaluation was performed on a per-signal basis. This section presents the main results and conclusions from such experiments, which allowed us to validate three important protocol mechanisms developed in WP3: (a) Path MTU discovery for UDP-based applications, (b) protocol signals that enable passive round-trip time measurements, and (c) support of low-latency service.

3.1 UDP-based Path MTU discovery

Applications and protocols using UDP as a substrate have several common tasks they must perform. Path MTU Discovery (PMTUD) is an important task for any packetization layer to perform for upper layer protocols. Classical PMTUD uses ICMP PTB error messages as a network signal that indicates the MTU estimate is incorrect. It is very common for networks to block or discard ICMP error messages, for a PMTUD mechanism this presents as a network 'black hole' where packets vanish without any signal of their fate. ICMP black holing breaks Classical PMTUD.

RFC4821 [32] defines Packetization Layer Path MTU Discovery (PLPMTUD), PLPMTUD is implemented at the point that the transport breaks down data into packets. PLPMTUD does not rely on ICMP error messages and is able to search for the optimal MTU in the absence of network error signals.

RFC4821 defines PLPMTUD for TCP and SCTP, the mechanisms it describes are targeted at TCP and it focuses on how to probe within TCP's congestion control context. UDP and datagram protocols require more than what is offered in RFC4821.

draft-ietf-tsvwg-datagram-plpmtud [17] defines Datagram Packetization Layer Path MTU Discovery (DPLPMTUD) and specifies the requirements and a set of algorithms for any datagram protocol to implement PLPMTUD. DPLPMTUD provides PLs that are able to send probe packets a way to reliably discover the Maximum Packet Size (MPS) supported on a path.

draft-ietf-tsvwg-udp-options [42], presented in Deliverable D3.3, adds transport option support to UDP. With the MSS and ECHO UDP Options it is possible to implement DPLPMTUD on UDP without using an upper layer protocol.

This section describes an experimental set up to evaluate the core algorithms described in D3.3 and [42]. Also, the section presents results for verifying this setup in both lab and real world network environments.

3.1.1 Implementation and evaluation setup

The DPLPMTUD experiment is performed in two phases and in two modes, i.e., with and without ICMP PTB error support. First we verify the implementation by testing both modes in a lab environment. Second, the implementation is tested in a real world environment using virtual machines hosted by a cloud provider.

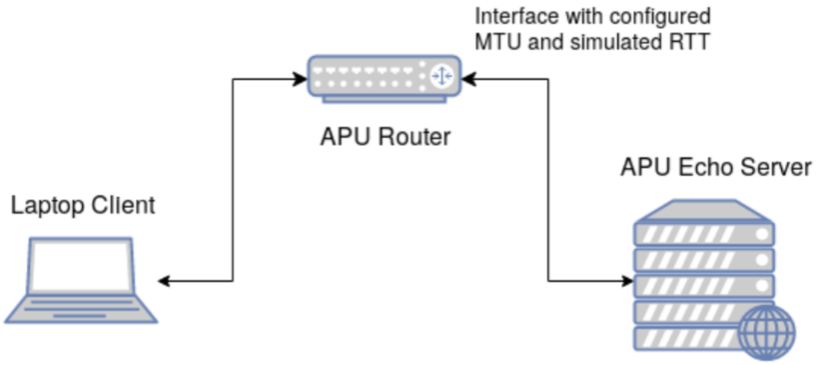


Figure 9: Datagram PLPMTUD Experiment Testbed setup

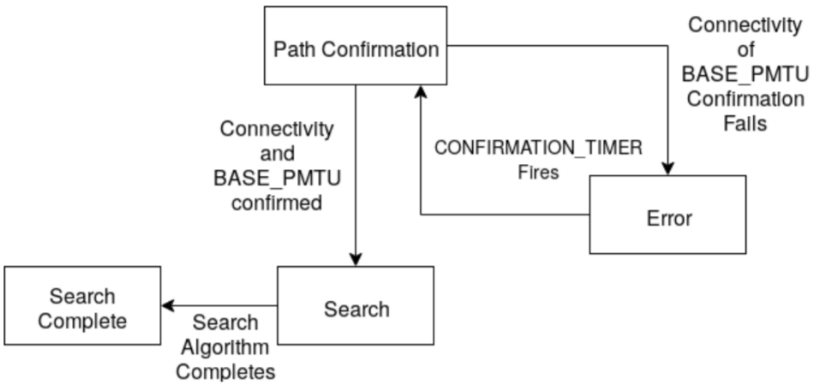


Figure 10: Datagram PLPMTUD Algorithm

For the evaluations we used a simple custom UDP Application protocol. This protocol sends datagrams that carry an authentication token, a request token unique to each request and the parameters of the experiment, the rest of the transmitted data sent is padding. The use of a custom protocol allows us to simplify implementation and to evaluate the userspace application interaction with DPLPMTUD.

The implementation consists of a Python-based client and server. The server side acts as an echo server, responding to any authenticated request packet. The client implements the algorithm described in Section 5 of draft-ietf-tsvwg-datagram-plpmtud[1] and has modular search algorithms. Modular search algorithms allow many different strategies to be tested in experiments.

Three search strategies have been implemented for use alongside the algorithm when testing: (1) an additive, linear search strategy, (2) a binary search algorithm and (3) a table based

[1] See `https://tools.ietf.org/html/draft-ietf-tsvwg-datagram-plpmtud-05#page-16`.

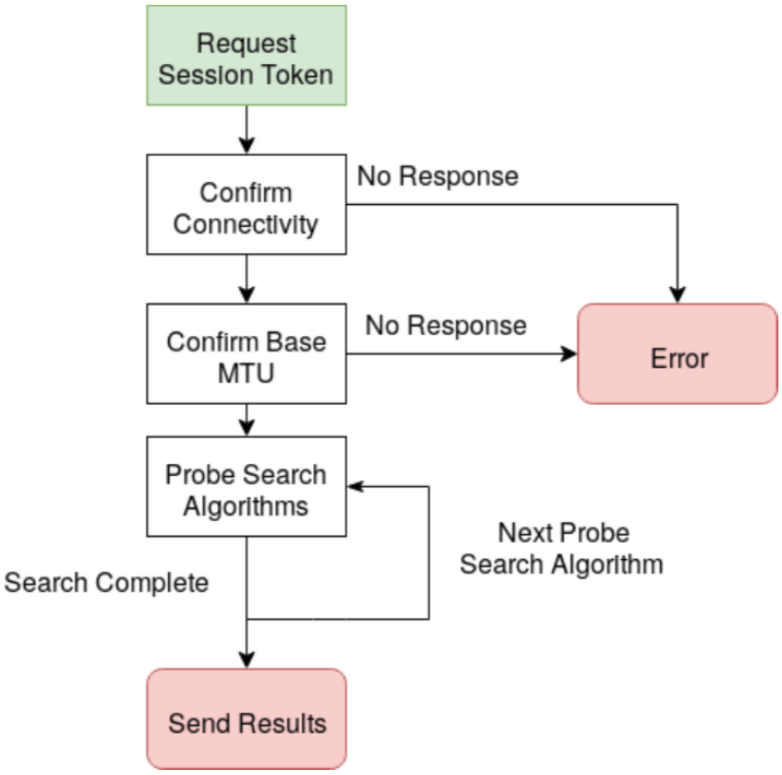


Figure 11: Datagram PLPMTUD Algorithm

search strategy. All search algorithms can accept parameters, different variants of step sizes and tables can be tried by the tool.

3.1.2 Evaluation Process

The DPLPMTUD tool runs through a set of stages when performing experiments. Figure 11 shows the stages the experiment works through. First the tool establishes connectivity to the measurement server by requesting an authentication token, this token is used to authenticate each probe in the experiment.

A simple search is performed to discover the path MTU to establish a baseline measurement for further searches, this is the highest value we expect any search algorithm to measure in any experiment. After baseline measurement is complete, the tool runs the algorithm with a set of different search algorithms each with different parameters.

The results of each search and the baseline are submitted back to the measurement server and the search is concluded.

Probes Received	Time to complete (s)	MPS found	Configured link size	PTB Status
273	14.3	1472	1500	Not Enabled
265	15.3	1464	1492	Not Enabled
253	14.7	1452	1480	Not Enabled
233	13.7	1432	1460	Not Enabled
227	13.3	1426	1454	Not Enabled
145	9.0	1344	1372	Not Enabled
273	14.3	1472	1500	Enabled
265	14.0	1464	1492	PTB = plpmtu
253	13.4	1452	1480	PTB = plpmtu
233	12.3	1432	1460	PTB = plpmtu
227	12.0	1426	1454	PTB = plpmtu
145	7.7	1344	1372	PTB = plpmtu

Table 1: Results of tests of the DPLPMTU with the Step Size search algorithm with a step size of 1.

Probes Received	Time to complete (s)	MPS found	Configured link size	PTB Status
6	0.3	1450	1500	Not Enabled
6	0.3	1450	1492	Not Enabled
6	0.3	1450	1480	Not Enabled
5	1.7	1400	1460	Not Enabled
5	1.7	1400	1454	Not Enabled
3	1.6	1300	1372	Not Enabled
6	0.3	1450	1500	Enabled
6	0.3	1450	1492	Enabled
6	0.3	1450	1480	Enabled
6	0.3	1432	1460	PTB $>$ plpmtu
6	0.3	1426	1454	PTB $>$ plpmtu
4	0.2	1344	1372	PTB $>$ plpmtu

Table 2: Results of tests of the DPLPMTU with the Step Size search algorithm with a step size of 50.

3.1.3 Results

Three search algorithms, Step Size (with step 1 and 50), Table derived and Binary Search, were tested with DPLPMTUD in both lab and real world networks. In each of the test runs the configured link MTU was recorded and the Maximum Packet Size (MPS) found by the tool was reported back to the server. As the tool runs in userspace over UDP 28 bytes are lost to headers in each probe packet (20 bytes for the IPv4 header, 8 bytes for the UDP header).

Tables 1 and 2 give the lab results for the Step Size search with 1 and 50 step size parameters, respectively. The step size 1 search was used to establish a working baseline measurement for subsequent searches and to demonstrate the function of the DPLPMTUD machine. Step Size 50 is more representative of a real search method. Step Size 50 when compared to Step Size 1 shows the trade off in improved search speed for lower resolution in the discovered PLPMTU.

Black hole detection with DPLPMTUD can take up to 10 RTTs, both Step Size tables show

Probes Received	Time to complete (s)	MPS found	Configured link size	PTB Status
7	0.3	1472	1500	Not Enabled
6	1.8	1464	1492	Not Enabled
5	1.7	1412	1480	Not Enabled
5	1.7	1412	1460	Not Enabled
5	1.7	1412	1454	Not Enabled
2	1.6	1332	1372	Not Enabled
7	0.3	1472	1500	Enabled
6	0.4	1464	1492	PTB = plpmtu
6	0.4	1452	1480	PTB > plpmtu
6	0.3	1432	1460	PTB > plpmtu
6	0.3	1426	1454	PTB > plpmtu
3	0.2	1344	1372	PTB > plpmtu

Table 3: Results of tests of the DPLPMTU with the Table Derived search algorithm.

Probes Received	Time to complete (s)	MPS found	Configured link size	PTB Status
1	0.05	1472	1500	Not Enabled
9	83.3	1464	1492	Not Enabled
9	3.3	1452	1480	Not Enabled
9	3.3	1432	1460	Not Enabled
9	4.8	1426	1454	Not Enabled
9	167.4	1344	1372	Not Enabled
1	0.5	1472	1500	Enabled
2	0.3	1464	1492	PTB > plpmtu
2	0.2	1452	1480	PTB > plpmtu
2	0.2	1432	1460	PTB > plpmtu
2	0.2	1426	1454	PTB > plpmtu
2	0.1	1344	1372	PTB > plpmtu

Table 4: Results of tests of the DPLPMTU with the Binary Search algorithm.

a search time benefit when used in networks that generate ICMP PTB messages when the search size reaches the bottleneck MTU. The time benefit here can be attributed to the DPLPMTUD machine using the PTB message when it arrives rather than having to wait for black hole timeouts to expire.

Table 3 shows the performance DPLPMTUD when using a Table Derived based search method. The Table Derived method uses a table of common Path MTU values advertised in networks based on measurements performed in Work Package 1 of the MAMI project. The MTU search table consists of the following values: $\{1232, 1332, 1372, 1398, 1412, 1464, 1472\}$.

The Table Derived search works upwards through values in the table, trying a point from the table. The search terminates either when a table value black holes or there is a verified PTB message.

Table 4 shows the performance of DPLPMTU when using a Binary Search strategy. The Binary Search uses the Base MTU as its lower bound and 1500 (the Ethernet MTU) as its upper bound.

Table 5 shows the performance of the three search algorithms in real world tests. The same

Search Type	Probes Received	Time to complete (s)	MPS found	Estimated RTT (ms)
Step Size 1	273	24.2	1472	88.2
Step Size 50	6	0.5	1450	88.2
Derived Table	7	0.7	1472	88.2
Binary Search	1	0.07	1472	88.2

Table 5: Results of tests of the DPLPMTU test tool in a real world environment with each of the four search algorithms. The test network did not generate any ICMP PTB messages.

test procedure as the lab tests was performed between Virtual Private Servers (VPS) hosted by cloud providers to an end point in the research network at the University of Aberdeen. These results confirms the lab results in a real test network. When considering the results it is important to notice that the performance of the algorithms is tightly related to the network configuration. In Table 5 the binary search is able to conclude with a single search hop, this is due to the MTU of the network being equal to the first search step. Equally the Table Derived search performs well due to the network configuration matching its search table. The two Step Size searches again show their trade offs in overhead and speed for the search performed.

The tested set of search algorithms show the benefit of not requiring a specific search algorithm in the standard. Further work is required to evaluate other search algorithms and hybrid methds such as binary search using a table of values to see if there are more optimal approaches.

The Datagram PLPMTUD testing was able to demonstrate the correct function of the DPLPMTUD algorithm in lab test networks and in real world scenarios. The testing showed that the core algorithm is able to discover the PMTU of a number of paths on networks with and without ICMP PTB error messages and the possible time improvement from allowing ICMP PTB traffic in networks.

3.2 Protocol-independent Mechanisms to Support Passive Network Measurements

In this section, we evaluate various passive Round-Trip Time (RTT) measurement techniques for TCP, PLUS and QUIC traffic. Of special interest are RTT measurements based on the *latency spin signal*, a technique developed by us and presented in the paper “Three Bits Suffice” [11].

RTT is a key metric in Internet measurement for network operators and researchers. Passive latency measurements reduce the bandwidth overhead compared to large-scale *active* RTT measurements. Unfortunately, such passive approaches are often limited to transport-specific features (e.g. Transmission Control Protocol (TCP) timestamps) or exploit properties of commonly deployed congestion and flow control algorithms. Our evaluation shows that passive measurements based on the latency spin signal can produce good latency estimations even under difficult network conditions (e.g. packet loss) while being easily integrated in various transport protocols using only three bits.

Before we present the results, we first introduce our high-performance middlebox implementation in Vector Packet Processing (VPP) [41] which we used to perform all the passive RTT measurements.

3.2.1 VPP-based passive latency measurement implementation

To evaluate various passive latency measurement techniques, we implemented a middlebox[2] based on VPP [41], a library for high-speed packet processing in userspace. The implementation adds a new node to the existing VPP tree. For each packet, it performs the following five main steps: *(i)* **detecting** supported transport protocols; *(ii)* **retrieving or creating state** for the observed flows using the 5-tuple as a hash key; *(iii)* **extracting** the required header fields to perform the latency measurements; *(iv)* **estimating and reporting** the latency estimations; and (optional) *(v)* **performing 2-way network address translation** to route traffic through the middlebox allowing for "on-path" measurements (as highlighted in 3.2.3).

The middlebox performs latency measurements using:

- the latency spin signal [11] in QUIC, PLUS and TCP traffic as described in D3.3 (Section 3.1.3) and evaluated in 3.2.2, 3.2.3 and 3.2.4;
- the TCP timestamp values [40] evaluated in 3.2.3;
- and the Packet Serial Number/Echo in PLUS traffic [29] evaluated in 3.2.4.

The VPP plugin is configured using various CLI commands. The most important ones are:

`latency interface <interface> [disable]` enables (or disables) the latency plugin on a specific interface.

`latency stats` prints a summary of all currently observed flows with the newest RTT estimations.

`latency quic_port <port>` declares User Datagram Protocol (UDP) traffic from/to a specific port as QUIC traffic. This command can be repeated with different ports.

3.2.2 Latency Spin Signal in QUIC

To evaluate the latency spin signal introduced in D3.3 Section 3.1.3, we added the spin signal to an open-source QUIC implementation[3] written in Go. Fig. 12 shows the traffic generation and the measurement setup. The network (on the left) consists of a client, a server, four switches and an observer in the middle of the path. We emulated the network in Mininet [30] which uses NetEm [24] to introduce network impairments. Two of the links (in blue) add static traffic shaping consistent over the entire measurement period. The two links between the observer and the switches (in green) generate dynamic traffic shaping allowing us to change e.g. the RTT after a certain amount of time.

During a measurement, client and server exchange QUIC traffic containing the latency spin signal and the on-path observer saves all recorded packets in pcap files. In a next step, we replay the packets from the pcap files to our middlebox implementation in VPP (on the right). We chose this two-step approach to ensure measurement reproducibility. Generated pcap files from all performed measurements are available on GitHub[4].

[2] `https://github.com/mami-project/VPP-latency-middlebox`
[3] `https://github.com/pietdevaere/minq`
[4] `https://github.com/mami-project/three-bits-suffice`

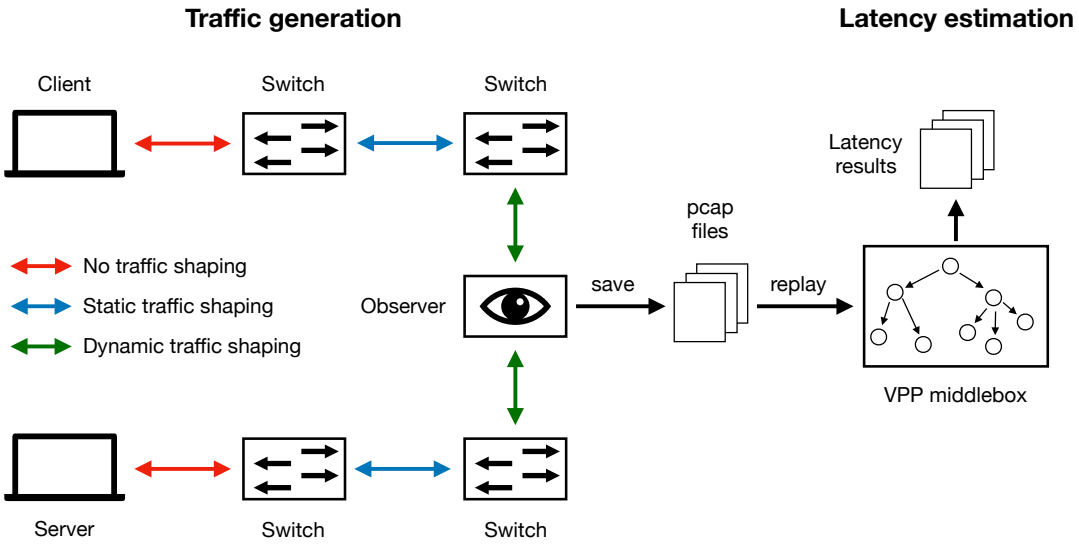


Figure 12: Emulated network and traffic generation on the left. RTT estimations on the right.

To evaluate and compare the performance of the latency spin signal under network impairments, we implemented four mechanisms to passively observe the RTT:

Spin bit The observer monitors only the spin *bit* to estimate the RTT.

Packet number The observer uses the packet sequence information to reject reordered packets. Note that depending on the used QUIC implementation, the packet numbers can be encrypted and are therefore not visible for on-path observers.

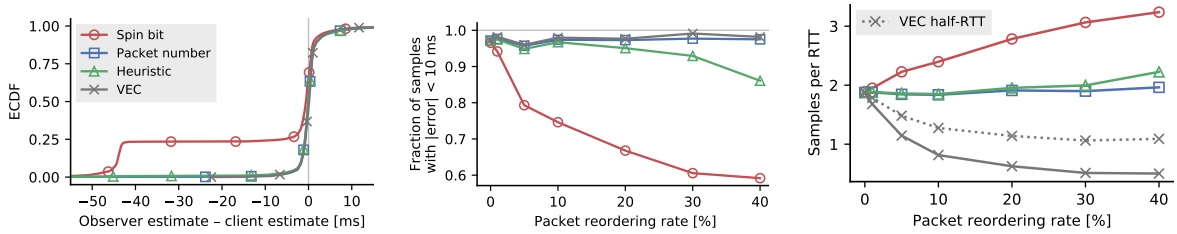
Heuristic The observer monitors only the spin *bit*, but rejects generated RTT samples below one tenth of the current estimate.

VEC The observer takes the full spin signal into account (i.e. spin bit and VEC), and rejects invalid edges based on the VEC value.

We evaluate the quality of the spin signal with two metrics. First, the error relative to client estimated RTT as per RFC 6298 [36] and second, the number of samples obtained per RTT. In addition, we also consider how many samples can be taken by the VEC observer when up- and downstream delays are measured separately. Because our observer can see both flow directions, they should ideally measure two samples (one for each direction) per RTT. However, spin bit transitions with VEC values below two can lower the sample rate, while superfluous transitions can increase it incorrectly.

To evaluate the tolerance of the spin signal to packet reordering, we use NetEm to randomly delay a configured fraction of packets by 1 ms. For each measurement, the network's RTT was 40 ms.

Fig. 13a shows the distribution of the RTT estimation error at a 10% reordering rate. It can be seen that the spin-bit-only observer often produces RTT estimates with an error around -40 ms. As this corresponds to the network's RTT, the observer took many near zero RTT samples. This behavior is expected as reordered packets can cause rapid transitions in the spin bit signal. All the other observers are able to filter out these erroneous samples. For reordering rates above 10% (evaluated in Fig. 13b), also the heuristic-based observer starts to deteriorate.

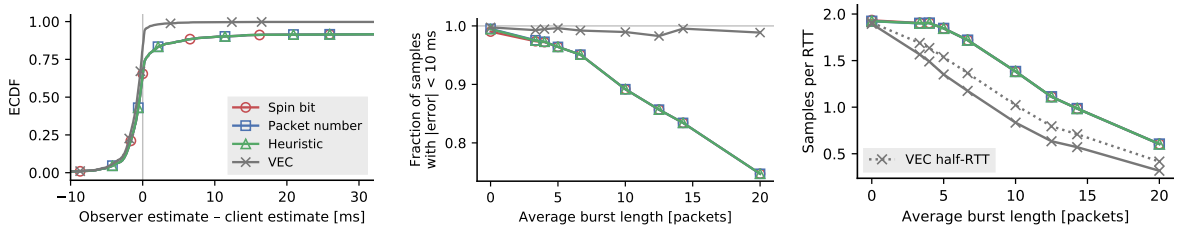


(a) Empirical Cumulative Density Function (ECDF) for a 10% packet reordering rate.

(b) Error in the observed RTTs for increasing reordering rates.

(c) Normalized number of RTT samples for increasing reordering rates.

Figure 13: The effects of reordering on spin bit based RTT measurements of a flow with a 40 ms RTT.



(a) ECDF for an average burst length of 10 packets.

(b) The error in the observed RTTs for increasing loss burst lengths.

(c) The normalized number of RTT samples for increasing loss burst lengths.

Figure 14: The effects of burst loss on spin bit based RTT measurements of a flow with a 40 ms RTT.

Even though Fig. 13b indicates that the VEC and packet number observer have a similar error performance, they behave fundamentally different: an RTT sample based on a disturbed spin signal is dropped by the VEC observer, preventing potentially wrong estimations. On the other hand, the packet number observer continues to take RTT samples, leading to additional error in the sample. This behavior is visible in Fig.13c.

Finally, we also evaluated the loss tolerance of the different observers. We configured NetEm to emulate burst loss using the simple Gilbert model [22]. The good reception periods have an average length of 100 packets. The average length of the loss bursts is varied. Fig. 14 shows the results.

In Fig. 14a we can observe that burst loss leads to overestimated RTTs. Whenever a packet carrying a spin edge is lost, the RTT measurement is not stopped until the next packet with the new spin bit value is observed. The long tail in the ECDF is caused by retransmission timeouts. Given that these timeouts are significantly longer than the network's RTT, they also reduce the number of RTT samples that can be taken as shown in Fig. 14c. As seen in Fig. 14b, only the VEC observer remains accurate under burst loss, as the VEC indicates that the spin signal was disturbed. The observer can then reject these incorrect samples.

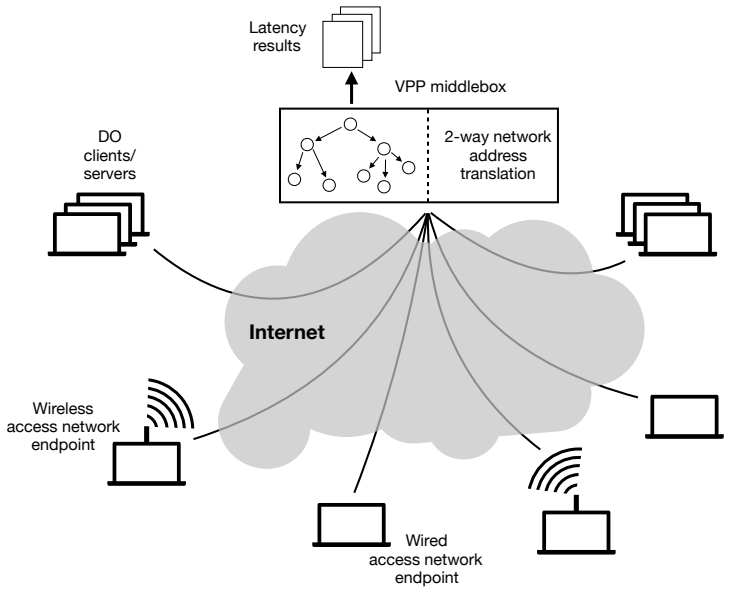


Figure 15: Setup for the TCP-based latency estimations. All flows are forwarded over the VPP middlebox using the 2-way address translation functionality.

3.2.3 Latency Spin Signal and Timestamp in TCP

To be able to directly compare RTT estimates from the latency spin signal with samples based on existing methods, we created patches to add latency spin signal support for TCP to the Linux kernel[5]. For these modified TCP flows we can compare RTT samples based on the latency spin signal with samples based on TCP timestamps as described in [40].

Fig. 15 illustrates the measurement setup. We deployed our VPP middlebox implementation on an ETH server connected to the Internet. Using the 2-way address translation features of the VPP plugin, we can forward TCP traffic from multiple sources (wired and wireless access networks as well as Digital Ocean[6] machines) over the middlebox to perform 2-way, on-path measurements over the “real” Internet.

We performed all measurements on 25 May 2018. In the end, we collected 53 traces coming from wired and wireless access networks as well as Digital Ocean machines, all going through our observer. Figure 16 shows typical RTT measurementsfor different scenarios. In each subfigure we compare the RTT measurements using TCP timestamps with estimations based on the latency spin signal.

Subfigure (a) shows a typical inter-datacenter trace. The spin signal measurement stays fairly close to the minimum of the noisier TS measurements. Subfigure (b), taken from a node connected via Ethernet to a residential access router, shows a more typical situation with larger buffers. Finally, Subfigure (c), from a wireless network with a bad case of bufferbloat, shows an extreme situation with high delay and moderate loss. Here, the latency spin signal’s sample rate reduces as the RTT increases; yet, it still provides accurate enough information for rough intra-flow measurement.

[5] https://github.com/mami-project/three-bits-suffice/tree/master/tcp/kernel_patches
[6] https://www.digitalocean.com/

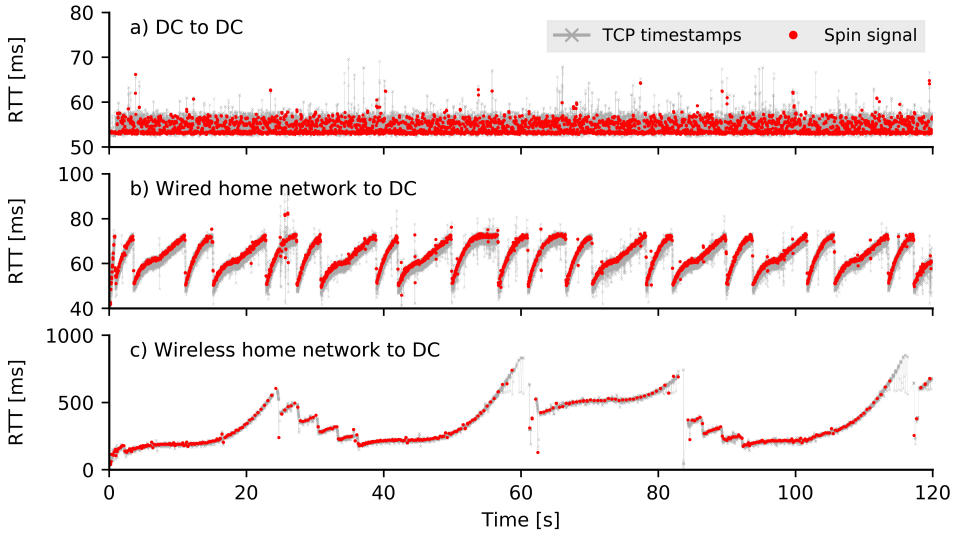


Figure 16: Spin- vs. TS-based RTT estimation over time.

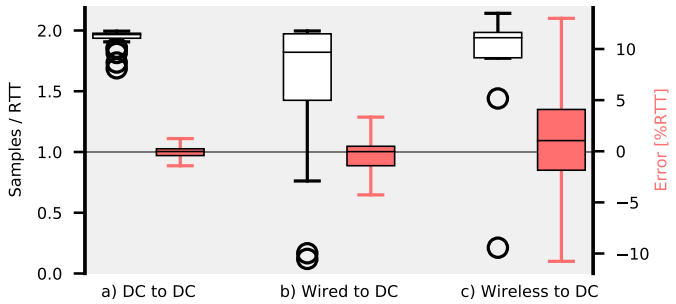


Figure 17: Sample rate per RTT (left plot/axis) and relative error between VEC- and TS-based estimation (right plot/axis).

Finally, Figure 17 evaluates the number of spin signal samples (axis on the left) as well as the relative error (axis on the right). As we do not have ground truth for the end-to-end latency, we compare each TS-based RTT sample to the latest value derived from the spin signal.

The median error (right-side boxplots) for inter-datacenter measurements is -0.03% and -0.04% for measurements from wired access networks. With 1.04%, the median error is slightly higher in case of wireless access measurements. The spin signal overestimates the RTT measured by the method based on timestamps. However, in wireless networks RTT is highly variable and in some measurement runs we also observed fewer valid samples due to reordering or loss.

Looking at the median number of samples per estimated average RTT (left boxplots) we observe a high number of samples for the wireless (1.94) and data center (1.97) traces. One of our wired access nodes expired a high amount of packet reordering, probably due to traffic shaping, which led to a relatively high number of invalid VEC edges and a median sample rate of 1.82 with high variance. This shows the expected behavior, invalid samples are filtered out.

Figure 18 shows three additional traces. Further raw measurement data and scripts to generate the plots are available on GitHub[7].

[7] https://github.com/mami-project/three-bits-suffice/tree/master/

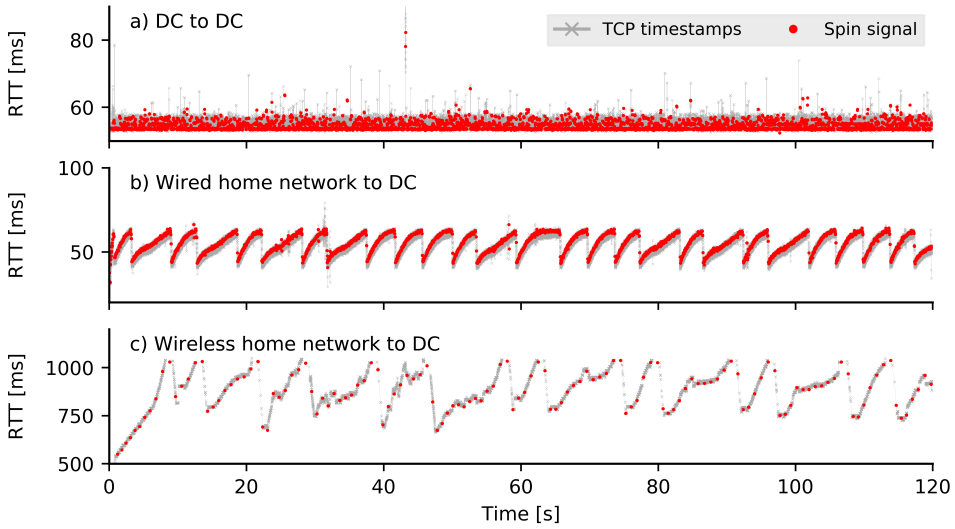


Figure 18: Additional traces for spin- vs. TS-based RTT estimations.

3.2.4 PSN/PSE in PLUS over Monroe

Finally, we also compare RTT estimations based on the Packet Serial Number (PSN) and the Packet Serial Echo (PSE) values found in PLUS [29] traffic with estimations based on the latency spin signal. In general, PSN and PSE are very similar to the timestamp values in TCP. There is only one slight difference: depending on the timestamp precision and the packet rate, two (or more) TCP packets could have the same timestamp value. That is not possible for PSN values. By design, they are increased by one for each packet including retransmissions.

We added the latency spin signal to our PLUS reference implementation in Go described in D3.3 Section 2.1. To that end, we removed three bits from the "magic" number used to identify PLUS traffic. Instead of `0xd8007ff`, the magic number is now `0xd8007f8 (0x1b000ff)`. One of these bits carries the spin signal, the other two are used for the VEC. It is important to note, that we could also use the PLUS Extended Header to transport the latency spin signal, although with slightly more overhead in terms of header space.

The measurements run over the Monroe[8] platform (EU Horizon 2020 project No 644399). Monroe consists of fixed and mobile nodes connected to real Mobile broadband (MBB) networks allowing to perform measurements and experiments over MBB networks.

For our experiment, we deployed Docker[9] containers on 21 different Monroe nodes located in Italy, Norway, Sweden and Spain. Each container runs the previously described modified PLUS client and downloads data from a Digital Ocean server. More precisely, each node downloads a 16 MiB file 20 times in a row. We use the 2-way address translation functionality of the measurement plugin in the same way as described in Section 3.2.3 to route the traffic over our observer. For better comparison with the TCP measurement results, we also perform some measurements for which both PLUS end points are deployed on Digital Ocean machines (clients in 8 different locations). All measurements were performed on 16 December 2018.

Monroe users have a limited bandwidth and time quota they can use to perform experiments

[8] `https://www.monroe-project.eu/`
[9] `https://www.docker.com/`

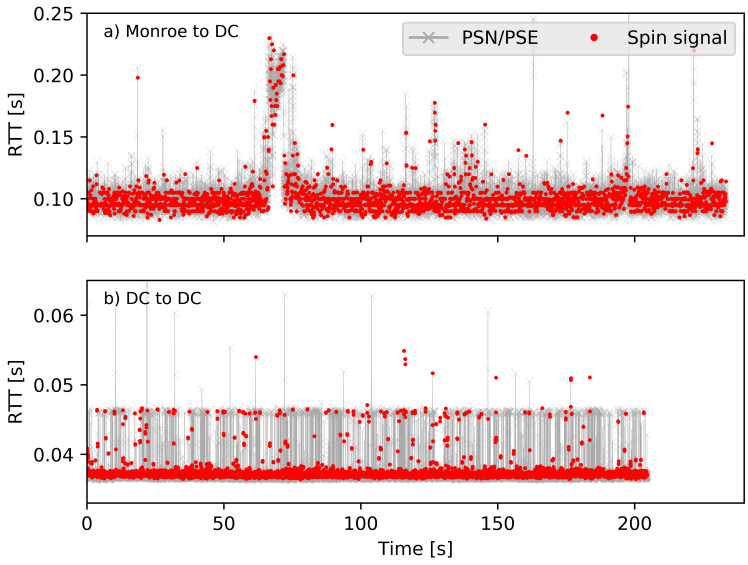


Figure 19: Typical RTT trace for RTT estimations based on PSN/PSE value as well as the latency spin signal in PLUS traffic.

on the platform. We therefore did not try to reach the highest throughput possible, but rather transmitted PLUS traffic with a fixed window/rate. This is also the reason why the traces in the following plots do not show common patters due to congestion control algorithms (as it is e.g. the case in Figure 16). For similar reasons, PLUS clients or servers will sometimes not immediately respond to a received packet which can artificially increase the measured RTT using passive techniques. In the following paragraphs, we will see how PSN/PSE and the latency spin signal observer handle these situations.

Figure 19 shows traces of RTT estimations based on PSN/PSE values and the latency spin signal. The measurement in a) was performed from a Monroe node. We observe the expected bumpy behavior from a MBB network. Between 50 and 100 seconds, the measured RTT doubles for a short amount of time. During the entire measurement period, the samples from the latency spin signal closely follow the estimations based on PSN/PSE values. In b) we observe a much more stable RTT evolution from a measurement between two Digital Ocean machines. As previously described, the PSN/PSE estimations show RTT observations that are artificially increased due to a PLUS client or server no immediately responding to a received packet. In most cases, the latency spin signal is able to filter these samples out and does not report on the increased RTT.

Similar to the evaluation in Section 3.2.3, we analyzed the amount of valid spin signal samples as well as the relative error compared to the estimations based on PSN/PSE values. On the left axis of Figure 20, we observe a median sample rate of 1.85 for the Monroe measurements (with a high variance) and 1.95 for the DC to DC measurements. This shows the expected behavior for the Monroe setup, the latency spin signal is able to filter invalid samples out.

The median error (right-side boxplots) for the Monroe measurements is 0.08% and 0.13% for the DC to DC measurements. The slightly higher median error for DC to DC observations can be explained due to fact that the PSN/PSE estimations (which build the ground truth for the error

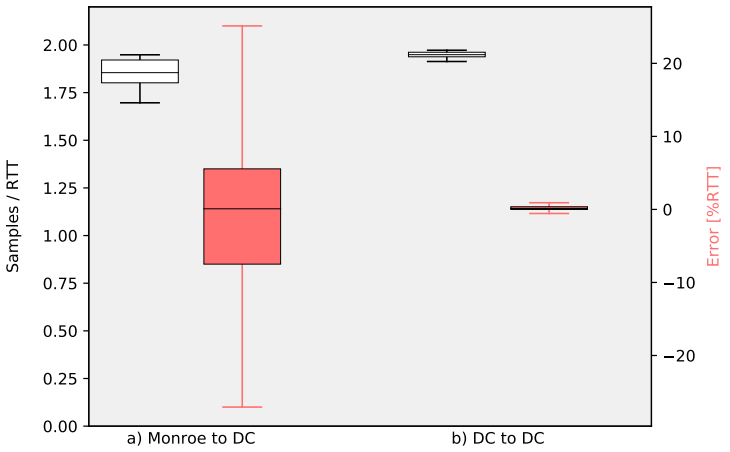


Figure 20: Sample rate and relative error.

analysis) are often artificially increased. Overall, the variance for DC to DC measurements is very small. That is not the case for Monroe measurements, where we observe a variance that is even larger than the TCP wireless measurements in Figure 15. This once more shows the challenging behavior of MBB network. Nonetheless, nearly all spin latency signal estimations have an error smaller than +/-8%.

3.2.5 Latency Plugin Performance Evaluation

To evaluate the performance of the VPP latency plugin, we used MoonGen [14] [10], a scriptable high-speed packet generator using Data Plane Development Kit (DPDK). A simple Lua script generates bi-directional PLUS traffic (without the latency spin signal).

We then assembled the following measurement setup: two servers are connected with two 1 GB links. On one server, we generate and receive traffic with MoonGen, the other server runs VPP with or without the latency plugin. We use VPP version 17.10 with DPDK 17.08.0. The server has an Intel Xeon E5620 CPU @ 2.40GHz with 8 cores and a total of 16 threads. Two cores are permanently pinned to the VPP process. In addition, we configured 1024 2 MB huge pages.

With the help of MoonGen, we generated 1 Gbit/s PLUS traffic on each of the two links forwarded over the VPP plugin on the other server. On average, VPP processes 1.74 million packets per second. To eliminate any delay introduced by the packet generation (MoonGen), a base delay – measured when connecting the two interfaces on the MoonGen server with a direct link – is subtracted. For each plugin configuration, we three times generated ten million PLUS packets and measured the delay.

Figure 21 shows the delay distribution introduced by the VPP plugin. The left-most boxplot shows the delay of an unmodified VPP installation *without* our latency plugin. We observe a median delay of around 0.22 ms. In a next step, we added the latency plugin but without the

[10] https://github.com/emmericp/MoonGen

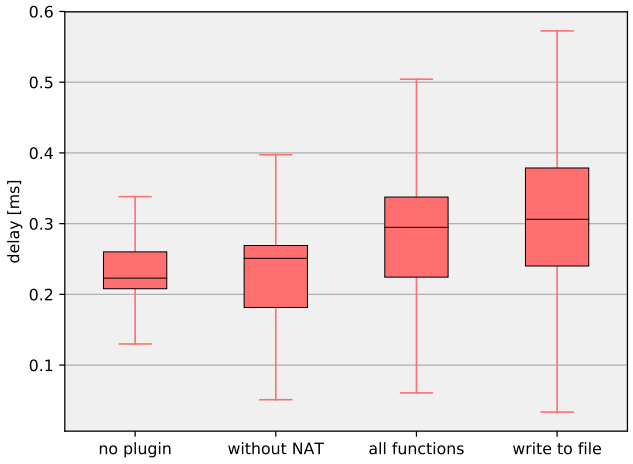


Figure 21: Delays introduced by the latency measurement VPP plugin.

two-way address translation used to perform on-path measurements. With this approach, the plugin does not have to re-write IP addresses. We achieve a median delay of around 0.25 ms. Enabling the two-way address translation results in a median delay of 0.29 ms. Finally, we evaluate the delay if VPP writes RTT estimations to a file (disk access) as soon as a new estimation is available, instead of saving results in memory. We achieve a median delay of around 0.31 ms.

As expected, the delay increases with additional plugin functions. Nonetheless, the delay introduced by our measurement plugin (compared to VPP without the plugin) is small and below 0.1 ms in the median case.

It is important to note, that neither the used hardware nor the server setup is specially fine-tuned for VPP. Nonetheless, the measurements show that a VPP-based implementation is able to process millions of packets each second introducing only a small delay.

3.2.6 Conclusion

In this section, we evaluated the performance of various passive RTT measurement techniques. Of special interest was the performance of the latency spin signal compared to existing approaches. As we have shown, the latency spin signal is able to produce precise RTT estimations for various transport protocols (TCP, PLUS, QUIC) in highly different network settings (wired and wireless access networks as well as mobile networks). Furthermore, the latency spin signal successfully filters out erroneous RTT samples, such as samples produced by reordered packets.

Given that the IETF recently decided to include the *Spin Bit* (although not the two additional VEC bits) in the first QUIC version, these results are very promising. Network operators and researcher should be able to measure the RTT even for the encrypted QUIC traffic. Nonetheless, given that we do not have the two VEC bits, detecting erroneous samples due to packet loss or reordering will be more challenging. Future work should focus on developing better heuristics to detect packet loss and reordering based on the spin bit signal alone.

3.3 Low Latency support in Mobile Networks

The increasing deployment of real-time voice and video communications, such as WebRTC, over both mobile and terrestrial network connections, has increased the importance of providing a low latency service in the Internet. These applications represent a qualitatively different class of traffic than the TCP traffic that dominates the Internet. Since their framing and codecs can compensate for loss, they are less loss-sensitive but often require low end-to-end latency. The emerging Augmented and Virtual Reality (AR/VR) traffic class, with applications ranging from gaming to tele-medicine, has even more extreme requirements in terms of the delay budget that is allowed at queueing nodes [23]. However, currently there exists no simple method for devices on path to provide them with a different treatment.

While for traditional bulk transfer applications, such as file downloads or most web-browsing, only the total transmission time is of actual importance, for interactive and real-time services, the delay of every single packet counts: a few milliseconds of additional end-to-end delay can render an interactive service unusable. The International Telecommunication Union's (ITU's) Recommendation G.114 [28] specifies a one-way delay of 150ms as the upper bound for interactive voice applications to provide acceptable Quality of Service (QoS) to their users. A base transmission one-way delay of 100ms, e.g., on an Internet path from Europe to the US west coast, leaves at maximum another 50ms for any additional delay, including all local and network processing. This suggests that, for this kind of traffic, every queue in the network should operate with a very low delay budget.

In mobile networks the situation is even more complex given that multiplexing is used to serve multiple users in one cell. Often, data is not sent continuously but in time slots that require a certain amount of buffering to fill the provided capacity when it becomes available. Therefore, just reducing the buffering may not be the best choice for all traffic, even if modern Active Queue Management (AQM) schemes are used that aim to maintain a low average queueing delay but provide enough buffer to handle traffic bursts.

In this section we evaluate different approaches for providing low-latency support in a range of scenarios, using ns-3 simulations. In the first set of scenarios, we investigate AQM and service differentiation schemes in a fixed network scenario, concluding that a differentiation scheme with two queues, separating low latency traffic, supports both low latency and high throughput better than modern single queue AQM approaches. In the second set of simulations, we concentrate therefore on the Loss-Latency Tradeoff approach, as described in more detail in D3.3, with two queues and use the Long–Term Evolution (LTE) component of the ns-3 simulator. These simulations also provide reference data for experimentation on the 5TONIC testbed, as described in Appendix A.

3.3.1 Approaches for providing low-latency support

We discuss next three approaches to better support low-latency applications in the Internet, that have been proposed in the literature and/or in standardization efforts by the project or others, differing in implementation complexity and deployment strategy.

3.3.1.1 Active Queue Management

Today, networks that are optimized for high throughput often implement large buffers to minimize loss. Further, mobile networks typically employ buffers that are even larger than those commonly found in fixed networks. In order to optimize the utilization of radio channels, sufficient data must be stored at all times in the radio layer buffers. If the buffers are empty, or not enough data is available, the radio channels may become underutilized. Obviously, this causes long delays which in turn impacts latency sensitive applications.

Recently, various AQM mechanisms have been developed to address this problem by keeping the buffer occupancy low, while still maintaining high link utilization and accommodate traffic bursts. Instead of dropping packets when the buffer overflows, AQM detects a continued build up of the buffer and drops packets early in order to avoid excessive loss and queuing delays. AQM can be used with Explicit Congestion Notification (ECN) [19] to mark packets as Congestion Experienced (CE) instead of dropping them.

In our experiments, we investigate the use of RED [18] as well as PIE [33, 34] as a strawman scheme for modern, latency-aware AQMs. RED calculates the drop or mark probability based on the current queue length. If a minimum threshold min_thresh is exceeded, RED starts dropping/marking packets at dequeue time with a low probability that linearly increases to a maximum value of max_prob with a growing queue until a second threshold max_thresh is reached. If the current queue length is above max_thresh all packets are dropped/marked. PIE, in contrast, randomly drops packets at enqueueing based on a drop probability p_{PIE} that is calculated periodically (every 15ms) depending on the current queueing delay (sojourn time) and a target queueing delay d_{target} as well as on the observed trend in latency, as follows:

$$p_{PIE} \leftarrow p_{PIE} + \alpha * (d_q - d_{target}) + \beta * (d_q - d_{last}) \tag{3.1}$$

As recommended, we use $\alpha = 0.125$ and $\beta = 1.25$ in our single-queue simulations [33, 34]. Additionally, PIE implements mechanisms to prevent dropping/marking if the queue is almost empty or if short bursts are detected that disappear in less than 150ms again. In our simulations, as we compare the use of PIE with L4S, we set d_{target} to a low value of 5ms, similar as proposed for DualQ schemes (see further below). To achieve a comparable low delay in our simulation setup for RED while still being able to fill the link, we use a min_thresh of 11 packets and a max_thresh of 34 packets with the recommended value of 0.02 (2%) for max_prob.

3.3.1.2 Loss-Latency (LoLa) tradeoff

The Loss-Latency (LoLa) tradeoff, originally proposed in [43] and further described in deliverable D3.3, is one specific example of an explicit sender to path signalling scheme. Traffic belongs to one of two classes: loss-sensitive (Lo) or latency-sensitive (La). Applications explicitly mark their flows as belonging to one of the classes, using a respective DiffServ codepoint (DSCP), and thereby signal their intended service treatment to the network. The network should treat these two classes differently, e.g. by using two queues with different configurations: one short queue (or one that uses an AQM optimized for low queueing delay) which might induce higher loss rates, and one larger queue that is optimized for high throughput and low loss. With this trade-off there is no incentive for the participants to lie in the cooperative game it creates: users cannot get an advantage in the treatment of their flows by cheating about their real nature because they would have to suffer either self-inflicted high delay or high loss as

a consequence of the incorrect marking. Other interesting properties of the scheme are: its incremental deployability and net-neutrality friendliness.

3.3.1.3 Low Latency, Low Loss, Scalable Throughput (L4S)

Similar to the LoLa approach, L4S [9] differentiates incoming traffic into two classes. However, L4S separates the low-latency and best-effort traffic for differentiated treatment based on the use of ECN codepoints as an identifier [38]. For this purpose the two ECN-Capable Transport (ECT) codepoints that ECN provides are used, ECT(0) and ECT(1). Further, while the La class assumes traditional, e.g., TCP traffic, L4S aims to provide low latency, low loss and high throughput under the assumption that the sender uses a scalable congestion control that, however, needs to be separated from traditional traffic to fairly co-exist.

With L4S, traffic separation is also implemented with the use of two queues, however, it is proposed to couple the calculated marking probability based on the DualQ Coupled AQM scheme [39] together with the use of a strict priority scheduler for the low latency traffic. That means the best-effort queue implements an AQM scheme and the marking probability p_L of the low-latency queue is computed depending on the mark/drop probability p_C of the classic, best-effort queue, as follows:

$$p_C = (p_L/K)^2 \tag{3.2}$$

This coupling together with the use of scalable congestion control provides throughput equivalence, even though the low-latency queue is prioritized. To achieve this, the coupling factor K depends on assumptions about which congestion control scheme is used in each traffic class. For the low-latency class, a scalable scheme such as Data Center TCP (DCTCP) [1, 7] is assumed, where the throughput is inversely proportional to the marking rate. DCTCP achieves this by implementing the same additive increase function for the sending rate as traditional congestion control (in this case one packet per RTT) but calculates the rate reduction in case of congestion dynamically based on the current congestion level. In more detail, it calculates a window decrease factor a based on a moving average of the congestion level that is estimated from the fraction of bytes sent that encountered congestion during an observation window (M):

$$a = a * (1 - g) + g * M \tag{3.3}$$

where g is the estimation gain. If congestion is detected DCTCP reduces the sending rate by a instead of using a fixed factor of, e.g., 0.5 or 0.7, as done today with traditional congestion control schemes such as NewReno [25] or Cubic [35], respectively. When the drop probability of the best-effort queue is set proportional to the square of the marking probability of the low-latency queue, as indicated in equation 3.2, DCTCP's throughput is the same as for classic AIMD congestion control even though it receives more congestion markings per RTT which, however, provides DCTCP a more fine grained feedback signal for rate adaption. For the case of DCTCP for low-latency traffic and NewReno for the best effort traffic, a scaling factor K of 1.64 achieves this fair share [39]. For simplicity, a scaling factor of $K = 2$ is recommended for all scenarios and used in our simulations.

In addition to DCTCP we also investigate the used of Relentless congestion control [31], which is another scalable congestion control scheme that simply reduces the congestion window by a fixed factor for every congestion marking received. Relentless does not perform any smoothing as DCTCP does but reacts to each congestion indication immediately. We implemented Relentless in an adapted version for use with ECN based on the DCTCP implementation in Linux.

That means we use the same feedback mechanism as DCTCP does but use a fixed reduction factor of half a packet per received marking. This value is selected to halve the window if all packets in an RTT are marked, similar as DCTCP does.

For the AQM used in the DualQ Coupled approach we investigate three different proposed schemes, namely PIE, Proportional Integral Controller Squared (PI2) [10], and Curvy RED (CRED) [8, 37]. PI2 simply uses the square of PIE's drop/mark probability, thus $p_C = p_{PIE}^2$. Respectively, when used with equation 3.2, to calculate the probability of the low-latency queue p_L, p_{PIE} only needs be multiplied with K. As recommended for the calculation of p_{PIE} for use with PI2 in L4S, we use a target delay for the classic, best-effort queue of 15ms, an update period of 16ms, a maximum drop probability of 0.25 for the best-effort traffic, $\alpha = 10$, and $\beta = 100$, aiming for much lower delay in the low-latency queue.

CRED is an adaptation of the RED scheme that uses an exponential increase function instead of a linear one and calculates the drop probability based on the smoothed queuing delay $\widetilde{d}_q$ as follows:

$$\widetilde{d}_q \leftarrow f d_q + (1-f)\widetilde{d}_q \tag{3.4}$$

$$p_{drop} = (\widetilde{d}_q / D_q)^U \tag{3.5}$$

where U is the curviness factor and D_q defines the slop as the value of d_q where p reaches 100%. This means, this function is bounded to 1 for a queue length above D_q. In the DualQ Coupled approach the best-effort probability is calculated with $U = 2$:

$$p_C = (\widetilde{d}_q / S_C)^2 \tag{3.6}$$

while the low-latency probability is calculated with $U = 1$ and based on the actual, not smoothed queuing of the best-effort queue d_q:

$$p_L = d_q / S_L \tag{3.7}$$

To realize the desired coupling with $K = 2$ of equation 3.2, S_C is set to 0.5 and S_L to 0.25. Further, a smoothing factor of $f = 1/32$ is used.

Moreover for PIE with L4S, we also introduce an additional step marking threshold in the low-latency queue, similar as also proposed for PI2 and CRED, where all packets get marked if the low-latency queueing delay exceeds the target queue delay of the classic queue. This is similar to the threshold marking that is proposed for use with DCTCP and is needed for cases where only low-latency traffic is present and therefore the best effort queue is empty, which would otherwise lead to a marking probability of permanently zero in both queues.

3.3.2 Base Comparison of Different Schemes in Fixed Network Setup

As a baseline we present simulation results for two single-queue schemes, namely Random Early Detection (RED) [18], a traditional AQM scheme, and PIE (Proportional Integral controller Enhanced) [33, 34], as an example of a modern AQM scheme that is optimized for low latency. We compare these with approaches that use two queues instead to separate low latency traffic from other traffic, namely a Loss-Latency (LoLa) tradeoff setup and Low Latency, Low Loss, Scalable Throughput (L4S) [9], and evaluate them with regards to latency and throughput in a baseline fixed network scenario using simulation. In our evaluation we show that lower latency can be achieved in all scenarios. We show that, while there is a throughput trade-off with the use of AQM in a single queue, more complex schemes utilizing two queues to differentiate

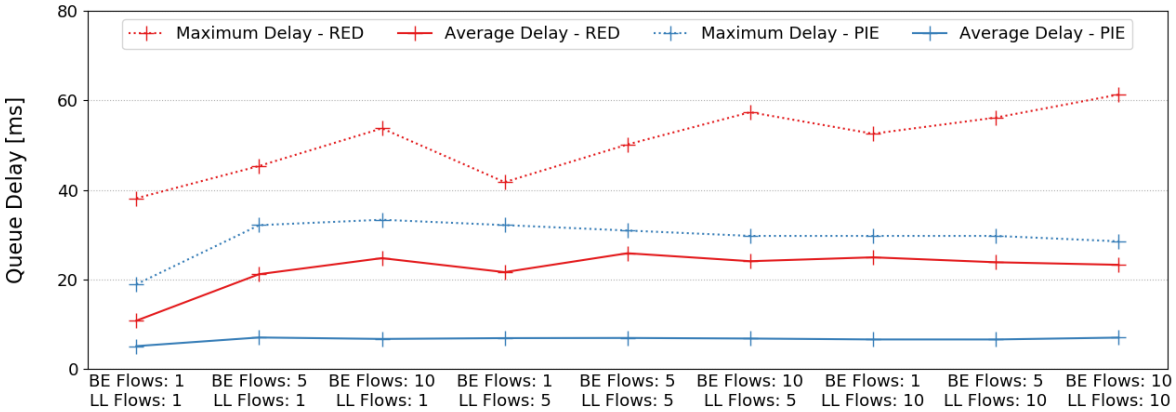


Figure 22: Average and maximum delay with Single Queue AQM on fixed link; all flows using Cubic with ECN

low-latency traffic from best-effort traffic can achieve both lower latency to support the needs of emerging interactive applications as well as full link utilization.

3.3.2.1 Simulation Setup and Mechanism

The network setup consists of three endpoint nodes that run Linux using ns-3's Direct Code Execution (DCE) and one network bottleneck node with a fixed link bandwidth of 10Mbps, One of the endpoint nodes, the client, is receiving downstream traffic from two other nodes, the remote servers, one for each traffic class. The two remote nodes are connected over two links with a bandwidth of 1 Gbps and a one-way delay of 50 ms. In this evaluation, we vary the number of flows (1, 5, and 10) in each class and present the results for all combinations of number of flows in each class and with the use of different AQM schemes. To avoid problems with ECN fairness, we only show results where ECN is always used in both classes.

We evaluated three different setups: 1) a single queue implementation with either no AQM, RED or PIE, 2) two queues that are not coupled but use different parameter settings, again for RED and PIE, and 3) DualQ Coupled AQM for L4S with PIE, PI2, and CRED.

3.3.2.2 Single Queue AQM

Figure 22 shows the average and maximum queuing delay for simulations with a single AQM for all traffic flows and the use of either RED or PIE, with different numbers of flows in each class. Note that each point is a separate simulation; the points are only connected for better visual recognition. For the results shown, all flows use Cubic congestion control. As expected based on our configuration, RED has in all scenarios a higher average and maximum queuing delay than PIE. However, if we would configure even lower thresholds the throughput would suffer even more. RED caused a link underutilization of about 3% with an average queuing delay of 18ms to 28ms (except for the case with only two flows where it is 11ms). PIE could achieve a lower average delay of 5ms to 8.8ms (with a configured target delay of 5ms) with a similar link underutilization of 3% to 5%. With the use of NewReno instead of Cubic, the underutilization

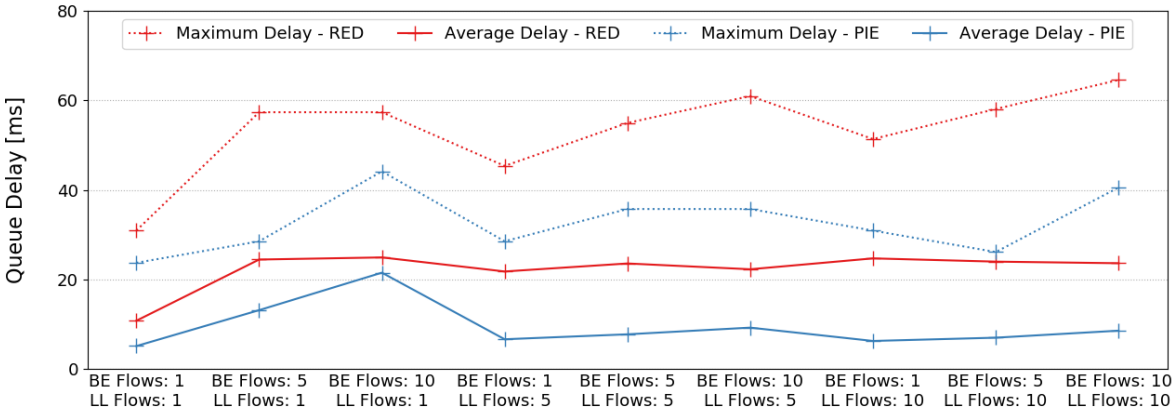


Figure 23: Average and maximum delay of the low latency queue with Uncoupled Queues AQM on fixed link; all flows using Cubic with ECN

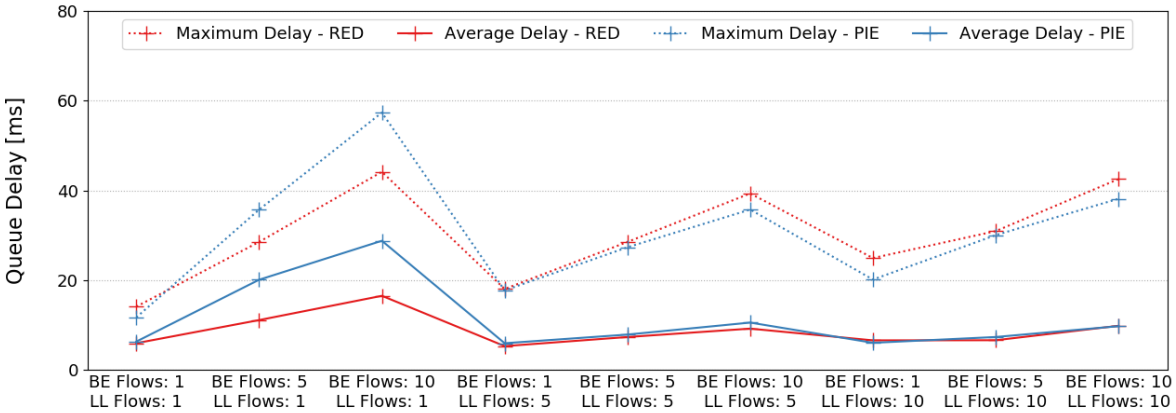


Figure 24: Average and maximum delay of the low latency queue with Uncoupled Queues AQM on fixed link; best-effort flows use Cubic and low-latency flows use DCTCP, both with ECN

was much worse which, however, leads also to lower average delays, as the queue is more often empty, but also higher maximum delays, as traffic is more spiky.

3.3.2.3 LoLa-based Uncoupled Queues

In this section we discuss our result for a setup with two queues that implements separation of the best-effort (BE) and low-latency (LL) traffic classes, providing an AQM setup with a low target delay for the latency-sensitive traffic and a much higher value for the best-effort traffic. We used same parameters as with the single queue AQM for the low-latency queue, but much higher thresholds for the best-effort queue, with $min\_thresh = 100$ and $min\_thresh = 300$ for RED and $q_{target} = 500ms$ for PIE. Further, we use a weighted round-robin scheduler, where the weight is set based on the number of flows in each class. In our simulation we know the exact number of flows, however, in a real-life setup the weight could be set based on a long-term average value.

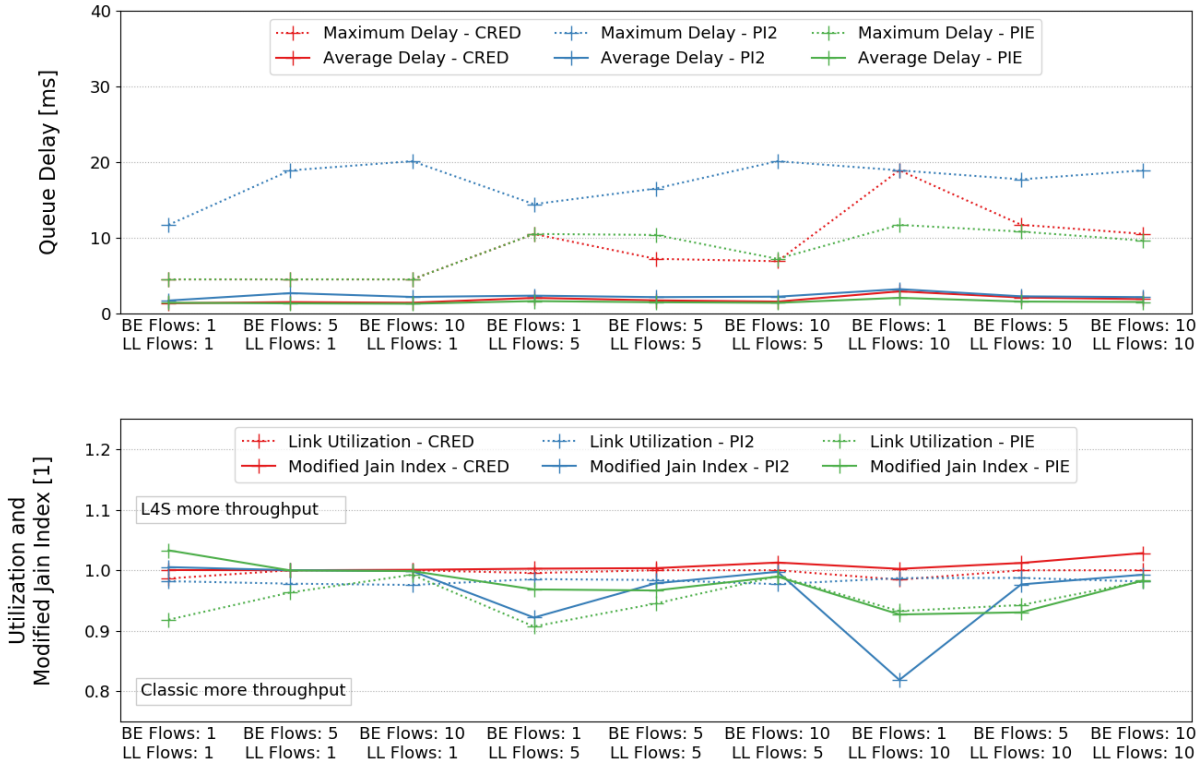


Figure 25: Average and maximum delay as well as link utilization and modified Jain's fairness index of low latency queue with Dualq Coupled with L4S on fixed link; best-effort flows use Cubic and low-latency flows use DCTCP, both with ECN

As it can be seen in Figure 23, again with use of Cubic for all flows, both RED and PIE achieve equally low delays for most of the scenarios, but this time with full link utilization in all scenarios, as there are always some packets in the best-effort queue to send. Further, due to the weighted round-robin scheduler fairness is still above 99%.

To be comparable with the results for the L4S setup, that we will discuss in the next section, we also evaluated this setup using two uncoupled queues with the use of scalable congestion control for all flows in the latency-sensitive traffic class. The results are shown in Figure 24 with DCTCP as scalable congestion control. This time we can observe, still, full link utilization with lower delays for RED, and lower maximum delays for PIE, as scalable congestion control reacts to the provided congestion feedback more fine-grained based on the current level of congestion. In this figure it can also be seen more clearly that with the uncoupled two queue scheme there is a dependency of the delay observed by the flows in the low-latency class on the number of flows in the best effort class. This is because we configure the weighted round-robin scheduler based on this ratio of number of flows in each class.

3.3.2.4 DualQ Coupled with L4S

As L4S is designed to enable deployment of scalable congestion control in co-existence with classic congestion control, we use either DCTCP or Relentless for the low-latency traffic. Figure 25 shows the maximum and average delay for the traffic in the low-latency class as well as

throughput and fairness of all traffic when using Cubic for the best-effort class. With a low target delay of 15ms for the AQM of the best-effort class, the average delay for the low-latency traffic is in all scenario is far below that, with 1.1ms to 2.1ms for PIE, 1.7ms to 2.6ms for PI2, and 1.0ms to 3.1ms for CRED. PIE has the lowest average as well as the lowest maximum delay of 4.5ms to 19ms, however, PIE slightly suffers in many cases of low link utilization of only 90.7% to 99.0%. PI2 suffers less but can still not reach full utilization and is always between 97.5% and 98.7%, especially in cases where there is only a low number of classic, best-effort flows but more scalable, low-latency ones which is also reflected by the unfairness in these scenarios. CRED in contrast usually reaches full link utilization, expect for the cases where there is only one low-latency flow in which the utilization goes down to 98.2%.

With the use of Relentless instead of DCTCP for the low-latency traffic, we achieved very similar results for the delay with an average between 1ms and 2.9ms for all schemes. However, this setup could reach even higher utilization again for all schemes with a minimum value for PI2 of 96% with 10 flows in the classic, best-effort traffic class and only one flow in the low-latency class. This is because Relentless reacts more directly without smoothing to the congestion signal. However, this also let to higher throughput, and therefore unfairness, in the classic, best-effort queue, especially with the use of PI2 and a large number of low-latency flows compared to best-effort flows. Again, throughput goes down in all scenarios if NewReno is used instead on Cubic for the best-effort traffic flows.

3.3.2.5 Discussion

In these simulations, we see that while the use of a modern AQM like PIE, that is optimized for low delays, provides reasonable good results for scenarios with fixed bandwidth, these schemes can suffer from low utilization in scenarios with a small number of concurrently active flows which is often the case in mobile networks as traffic is separated in per-user queues.

The use of traffic separation with two queues and different configurations, namely high target delay values for classic, best-effort traffic and low target delays values of 5ms for traffic that requires low latency, addresses the problems with the underutilization that single-queue AQM showed. This traffic separation relies on a signal provided by the sender to indicate which class to use, as proposed for DiffServ with two codepoints with a tradeoff between low latency and high throughput/low loss. While DiffServ marking is not deployable over a full Internet path, due to DSCP bleaching and rewriting, it may be feasible to use such a signal between a client and the access provider in mobile networks.

L4S is a system that is designed to make deployment of scalable congestion control, like DCTCP, feasible on the Internet while still being able to fairly coexist with classic congestion control schemes as deployed today. It addresses thereby the same problem of supporting new, emerging services with demanding low latency requirements. We evaluated L4S with three different AQM schemes in the proposed Dualq Coupled setup that is also using two queues but where the drop/mark probability of the low-latency queue depends on the calculated probability of the classic, best-effort queue. Thereby, L4S with all schemes, could reach even lower delay for both the low-latency traffic and classic, best-effort traffic, however, was not always able to reach full link utilization.

In summary, all approaches provide low latency but differ in their complexity and achieved throughput. This first evaluation shows promising results, especially for a simple service differentiation scheme with two queues based on LoLa trade-off signaling. For L4S a strict priority

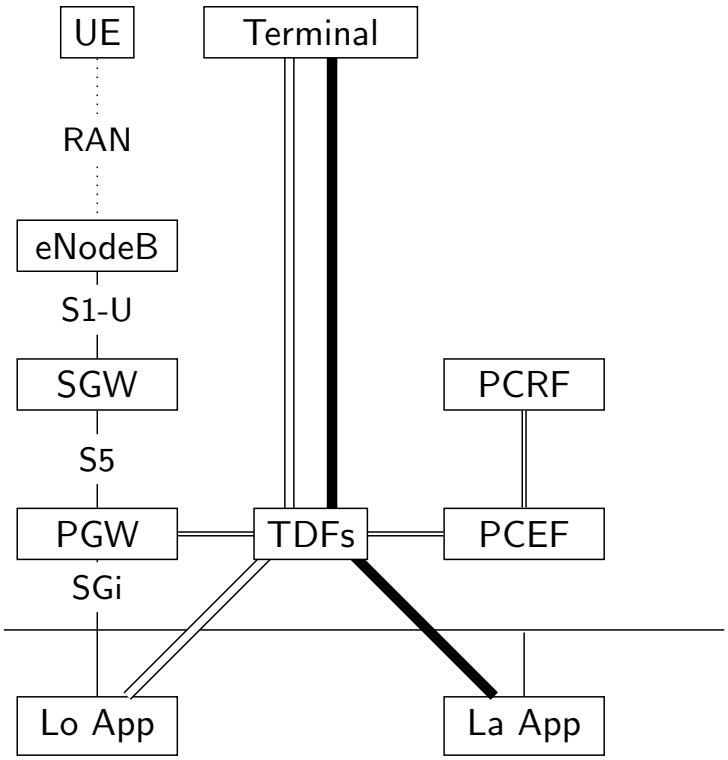


Figure 26: Scenario used in the simulations

scheduler is used, however, this does not seem suitable for mobile networks, as mobile resources are expensive, approaches like proportional-fair scheduling are usually used where the current signal strength is taken into account for the scheduling decision. Future work to apply these kind of scheduling schemes to L4S is needed.

In the next section we therefore focus on the LoLA approach and provide simulation results in order to evaluate more realistic network conditions in mobile networks.

3.3.3 Evaluation of Loss-Latency Tradeoff Signal for Mobile Networks

We use a number of ns-3 based simulations to measure the impact of Loss/latency tradeoff (LoLa) signalling coupled with a simplified LTE Quality of Service (QoS) setup that uses one dedicated low-latency bearer in addition to the default bearer.

The crucial characteristic of this QoS framework is that cheating (i.e., using a certain marking to obtain an advantage over competing flows) makes no sense to the end user because it may actually degrade their Quality of Experience (QoE). This property is explicitly proven by one of our experiments.

In our implementation, we use the low-latency DiffServ Code Points (DSCPs) defined in [43]. This choice makes it trivial to specify the matching Traffic Flow Template (TFT) in the simulation environment. It should be noted though that the use of the Loss/Latency Trade-off (LLT) marking scheme in this context is just exemplary and, as long as the marker and the classifier agree on

Table 6: Configuration used in mobile simulations

LTE-uu	FDD SISO, 6 RB baseline latency:	downlink peak: 4.4Mbit/s 3ms
S1/S5/S8	Data rate: propagation latency:	5Mbit/s 0ms
SGi	data rate: propagation latency:	10Gbit/s 1ms
eNB	proportional fair MAC scheduler	
Bearers	default: dedicated low-latency:	QCI 9 QCI 7

the position and semantics of the signal used to identify low latency flows, the specific kind of marking is irrelevant.

Fig. 26 shows the different elements and flows that are used in the experiments. The white flow represents a greedy TCP flow that is not application limited (e.g., a large file download). The black flow represents a one way real-time flow with a bandwidth of 64kbps (e.g., a real-time audio flow). Marking, classification and the corresponding queuing strategy depend on the scenario we test. For each set of QoS settings we provide a base measurement where we apply the traffic to the network with these settings disactivated and the experiment proper, where they are activated and traffic is respectively marked as low latency.

The configuration of the SGi-LAN, LTE core and radio segments used in all the experiments is described in Table 6.

To evaluate the experiments, we use latency (delay and jitter) statistics. In order to calculate the delay, we include a timestamp in the low-latency packets and calculate the delay with the arrival timestamp. The jitter is calculated from the set of measured delays $d_1, ..., d_N$ using the following formula [2],

$$jitter = \frac{1}{N-1} \sum_{i=1}^{N-1} |d_{i+1} - d_i|$$

3.3.3.1 Experiment 1: The Honest Marker

In this experiment, we use two flows simulating a file download and a real-time flow. We characterise the flows in the control with no marking: the mobile network will put both on the same default bearer. Then we enable LLT marking on the real-time flow, which is routed through the dedicated low-latency bearer as a consequence. We simulate audio and video streams by controlling the size of the packets and the rate at which they are generated. In addition to measuring the QoE parameters of the Constant Bitrate (CBR) flow, we also examine the throughput of the TCP flows.

Table 7 compares the latency of the real-time flow in the control scenario and the experiment. With the marking enabled a reduction by 72% in the latency of the audio stream is achieved.

When we now examine the throughput of the TCP flow, Table 8 shows that the measured throughput is slightly reduced when marking is used. However, a variation of -0.39% will likely

Table 7: Real-time flow latency (ms) in experiment 1

Stream type	Run	Delay mean	Delay min	Delay max	Jitter
audio	no marking	15.48	5	24	4.43
	marking	4.32	4	6	0.72
video	no marking	16.15	6	26	4.69
	marking	8.39	7	10	0.66

Table 8: TCP throughput in experiment 1

Run	Throughput
no marking	3.807 Mbit/s
marking	3.793 Mbit/s

have no impact on the experienced QoE.

Takeaway 1. Improved delay and jitter for the real-time flow with negligible decrease in efficiency of the throughput seeking flow (and therefore of the RAN as a whole).

3.3.3.2 Experiment 2: The Cheater

In this scenario, we run two greedy TCP flows (large file downloads) and a concurrent real-time flow. We enable LLT marking on the real-time flow both in the control and the experiment. In the experiment, we also mark one of the greedy TCP flows in order to route it through the low latency bearer together with the real-time flow. This TCP flow simulates the cheater.

In this case, we use TCP retransmissions and the throughput to compare both runs. As shown in Table 9 the cheater ends up retransmitting a lot more (+460%) which implies a substantial decrease in throughput: The cheater gets -27.5% throughput (honest gets a 18.85% boost as a consequence), as shown in Table 10. Additionally, as shown in Table 11, we observe that the real-time flow of the cheater is not affected by the cheating TCP flow.

Takeaway 2. Cheating penalises the throughput of the greedy TCP flow significantly, while *not* degrading the real-time flow.

Table 9: Number of TCP retransmissions in experiment 2

Run	Honest	Liar
no marking	24	25
marking	31	140

Table 10: TCP throughput in experiment 2

Run	Honest	Liar
no marking	2.019 Mbit/s	1.837 Mbits/s
marking	2.400 Mbit/s	1.332 Mbits/s

Table 11: Influence on real–time latency (ms) of the cheater in experiment 2

Run	Mean	Min	Max	Stddev
no marking	5.373	4	9	1.156
marking	5.373	4	9	1.156

3.3.3.3 Experiment 3: Multiple UEs connected to one e-Node B

In this experiment, we want to see how the system behaves when we have many users together. For this experiment, we increase the bandwidth at the S1, S5 and S8 interfaces to 50 Mbps. We assume that marking users are *honest* users and examine the perceived QoE of marking and non-marking users as the number of marking users increases in a constant population that produces an aggregate BW of latency-sensitive flows that does not surpass the bandwidth available at the RAN (i.e., the bottleneck).

We worked with a population of 20 User Equipments per e-Node B (eNB). Each User Equipment (UE) receives a UDP stream simulating a video/audio stream and TCP traffic from a greedy application running in the background. In each run, we had a fraction of the UEs receive the CBR stream through the dedicated low latency bearer, while the rest used the default bearer for both streams. The UDP streams were started at random times uniformly distributed between 1 and 3 s, the streams lasted for 10 s and the simulator ran for 14 s, assuring that the full CBR stream was captured. We simulated an audio stream carried over RTP (over UDP) with the following characteristics separately:

Table 12: CBR stream parameters. Packet sizes are in bytes, bandwidth (BW) is in kbit/s

Stream type	Packet size	PPS	effective BW	IP layer BW	# of nodes
audio	160	50	64	80	20
video	100	400	320	352	10

Fig. 27 shows the delay (d) and jitter (j) measured in each of the UEs. The measurements are grouped by colours depending on whether the node received a marked or an unmarked audio stream. It can be clearly appreciated that when marking was used (i.e., when QCI=7 was used for latency-sensitive traffic) the nodes perceive significantly lower delay and jitter. This implies that using LoLa marking in these streams produced much better QoE. In particular, it is a well known fact [28] that the quality of a voice call degrades rapidly where the mouth-to-ear delay latency exceeds 200 ms.

We then repeated the experiment with 10 nodes and a 320kbit/s UDP CBR (video) flow competing with a greedy TCP flow per node. We observe the same behaviour, i.e., that marking

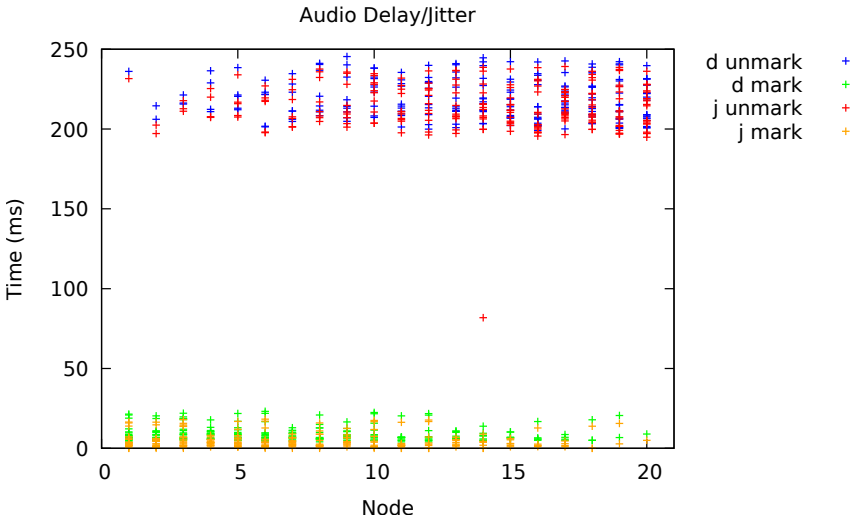


Figure 27: QoE parameters for 20 CBR flows @ 64kbit/s

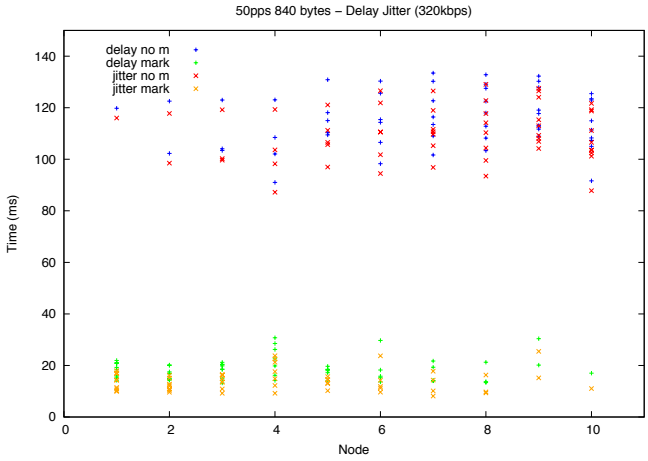


Figure 28: QoE parameters for 10 CBR flows @ 320kbit/s

yields better QoE behaviour than not marking. The results are shown in Fig. 28.

Takeaway 3. The dedicated low-latency bearer produces extremely good results, with delays one order of magnitude less than the default bearer.

3.3.3.4 Discussion

The simulation performed for low latency support mechanism in mobile network show promising results. Based on these simulation results to enable more realistic experimentation, we also produced a first set of experiment specifications for 5G testbeds. These specifications have also been implemented as a set of VNFs and experimentally deployed on the 5TONIC [26] testbed at the premises of the IMDEA Networks Institute in Leganés, Madrid. We expected to profit from the 5G-capable Radio Access Network (RAN), that should have been integrated in

the 5TONIC laboratory during the lifetime of the project. The lack of support for multiple bearers at the end of the lifetime of the MAMI project has made it impossible to reproduce the simulation experiments on it during the project's life time. Further details of the testbed implementation are shown in Appendix A. However, the setup has gained attention from other projects. For example, `trafic` (see Appendix A.1) is currently deployed and used in the 5TONIC facilities for activities like calibrating and testing the infrastructure. In addition, the code has been forked by the 5GinFIRE project (`https://5ginfire.eu`), which intends to use and extend it (cf. Fig. 29).

The simulation results presented in this section have been further presented in standardization and are used for industry dissemination.

4 Support for Virtualised Deployments of MAMI Components

The LoLa experiments originally planned at the 5TONIC premises, as described in Section A, provide a proof-of-concept for a Virtual Machine (VM)-based deployment of components developed in the scope of MAMI. They show the flexibility inherent to a testbed where measurement components are implemented in Virtual Machines that can be deployed and decommissioned on-the-fly. This approach allows the experimentation facility to be flexibly shared by several projects acting as tenants.

Moreover, having predefined component descriptors allow for quick deployment and, provided equivalent platforms are used, repeatability of the experiments. Current tools to deploy scenarios with multiple VMs are based YAML-based description languages which are difficult to generate and maintain. This is where high–level "human–understandable" description languages like NEMO come into the picture.

4.1 Cloud–based Deployments

Maximum flexibility is achieved when the VMs are deployed on the virtualised infrastructure using an orchestration framework like Openstack[1]. In this case, there are two possible paths to implement a functionality: 1. to prepare a specific VM image the contains a software instance of the functionality; or 2. to use a generic Operating System (OS) image available in the virtualisation platform and install the software on-the-fly, when the machine is created. `cloud-init` is a widely used mechanism in OpenStack, that implements the second approach. It started initially as Ubuntu-specific project and is maintained by them. However, it is being adopted by other OSes.

Regarding the Network Functions Virtualisation (NFV) camp, configuration is more complex. The orchestrator platform Open Source MANO (OSM) distinguishes different *levels* of configuration for VNF Components (VNFCs) [20]:

Level	Description	Examples
Day 0	VM configuration and injection of the basic configuration	[21]
Day 1	VM activation and injection of runtime configuration	[27]
Day 2	Service interaction, scaling, etc.	[27]

Open Source MANO allows using `cloud-init` for *day-0* configuration. Although not supported for all Virtual Infrastructure Managers (VIMs) that interoperate with OSM, it is a quick win when using OpenStack as the VIM[2].

`cloud-init`[3] provides a set of functions to configure a VM at boot-time. We use the following to package various tools developed by the MAMI project in order to dynamically deploy them on OpenStack based data-centres and in NFV environments with the OSM Resource Orchestrator (RO) driving an OpenStack VIM:

[1] `https://docs.openstack.org/rocky/`
[2] This is the case in 5TONIC and why `cloud-init` was used there.
[3] `https://launchpad.net/cloud-init`

- **Software (SW) package management**, including repository management, package list update, upgrade of packages installed in the stock cloud image, and installation from new packages
- **system configuration**: user management, management of stock service configuration files (for example Domain Name Service (DNS) management through /etc/resolv.conf)
- **creation of files** at boot time, including ownership and permission creation
- **execution of commands** at different phases of the boot process

Since not all MAMI components are official packages we could directly import into the VM, we create files that mimic the manual installation process described in the github repository of the specific component. Some of these components can be locally deployed using `vagrant`[4]. In this case, we use the file describing the deployed VM (known as `Vagrantfile`) and any files it references as a starting point.

Basic assumption: Always prefer **releases** over HEAD when importing code from `github.com`. The behaviour of a release is supposed to be stable for a given feature set. Using HEAD will import code into the VM that is prone to feature changes or even development hick-ups (*i.e.* malfunctions).

Currently, `cloud-init` configuration templates are available the following components, all using a stock Ubuntu 18.04 (aka bionic) cloud image from the Ubuntu repository:

- **PathSpider**: using release 2.0.1
- **VPP-latency-mb, the VPP based Middlebox emulator**: currently using release 0.1
- **vpp-plus**: currently using HEAD, until a release is produced

`cloud-init` templates are included in the documentation or in the GitHub repository. In order to use them in deployments, the user description part has to be customised with the specific user configuration used in the deployment environment.

In addition to the cloud-init template, the **VPP-latency-mb** project includes templates for the VNFC and Service Descriptors. These descriptors follow the specification for the OSM-RO Release Three [15].

4.2 Recursive VNF Descriptors Using NEMO

The Network Modelling (NEMO) language provides a high–level language to describe network deployments using an intent-based approach. This highly simplifies creating network configurations, since it hides the low–level device specific details. It was initially proposed in a BoF at the IETF-93 in 2015. The initial proposal was accompanied by a NEMO web page[5] and a proof-of-concept development that made it to the core of the OpenDaylight (ODL) Software-defined Networking (SDN) controller as the `ibnemo` plugin [44].

[4] `https://www.vagrantup.com/`
[5] `http://nemo-project.net`

From its very early stages, the NEMO language has been proposed as a high–level language for describing Virtual Network Functions, in order to hide away the complexity of the YAML language used in TOSCA or OSM VNF Descriptors. Scenarios with multi–component VNFs, i.e. VNFs composed of more than one VNF show the merits of using a "human-understandable" description language. A specific plugin for ODL existed[6], however the evolution of the controller framework had obsoleted the code and a major revision of the code was necessary. The new version is available from the MAMI github[7].

We used the NEMO language to describe the 5TONIC LoLa experiment in Section A, which motivated further enhancements to the VNF–specific aspects of the NEMO language. All enhancements are presented in the Internet Research Task Force (IRTF) draft [3]. The draft compares the use of "native" (*i.e.* YAML-based) VNF Descriptors (VNFDs) with NEMO–based descriptors.

Further applications of NEMO in the context of MAMI include extended use of the descriptors presented in Section 4.1, especially in scenarios where they are part of a deployment. The VPP based Middlebox emulator is a quick-win. A basic scenario where it "sits" between two endpoints is beyond the scope of the current descriptor examples provided by OSM [16], while a high–level NEMO–based scenario description is simple to produce.

4.3 Conclusion

More and more middleboxes migrate from physical to cloud or NFV–based deployments. In order to dynamically integrate middleboxes in such scenarios, reference descriptors allow for quick and reproducible deployments. As NFV service descriptors are complex, a description language as NEMO helps to reliably generate service descriptors for network services integrating middleboxes with other network functions. In the MAMI project we have applied these approaches to the software components developed in the project, enabling other projects to easily apply these components in NFV–based deployments for further experimentation and potential use in network management.

[6] https://github.com/telefonicaid/vibnemo
[7] https://github.com/mami-project/nemo

5 Conclusion

This deliverable has summarised the final work done in MAMI's WP2, regarding both middlebox modeling, NFV-based experimentation and experimental evaluation of proposed protocol mechanisms.

First, we presented the effort to design and develop a middlebox simulator, drawing from earlier results in this WP on a middlebox taxonomy and behavioural model. The evaluation study shows that the simulator can efficiently reproduce a range of middlebox behaviors, offering better performance than equivalent state-of-the art tools.

Next, we described three groups of experimental activities on middlebox cooperation approaches as well as work contributions to NFV-based experimentation which allows rapid deployment for further testing. The experiments used a wide range of performance evaluation tools and settings, from simulations to controlled lab setups to tests over real Internet paths and large-scale testbeds, and considering wired and/or wireless environments as appropriate. These experiments allowed to validate the viability of several protocol mechanisms coming from WP3, i.e., Path MTU discovery for UDP applications, protocol-independent mechanisms to support passive network measurements of end-to-end latency, and signaling for low-latency support. In particular, real-world results obtained across Internet paths have confirmed the feasibility of the DPLPMTUD and the passive RTT measurement approaches developed in MAMI. Also, as further explained in deliverable D4.4, results from these experimental activities have contributed to advance related standardisation activities on UDP options, Path MTU discovery, DiffServ codepoints, and the QUIC protocol.

A Testbed-based Setup for Mobile Experimentation

In-order to conduct experiments for evaluating LoLa support in a near-to-real 5G network, MAMI was given access to the 5TONIC experimentation testbed in Legan's, Madrid [26]. To run these experiments the `trafic` tool[1] was developed by the MAMI project and deemed mature enough to be deployed in the testbed. This experiment should have complemented the LoLa simulations discussed in Section 3.3.3, however as explained in that section, the current lack of support for multiple bearers made it impossible to reproduce the simulation experiments during the project's life time.

A.1 trafic

In the experiments, we need to have controlled and reproducible traffic mixes. Using `iperf3` [2] to generate individual flows, `trafic` is a traffic flow scheduler to create flow mixes. Additionally, it uses the Key Performance Indicators (KPIs) generated by `iperf3` to generate time series that are stored in an `influxdb` database for further processing.

Currently, the software dependencies of `trafic` are:

1. For the traffic generator/sink/sniffer functionality
   - `golang v1.10`
   - `iperf3 >=3.5`
   - `tshark`
2. For the database
   - `influxdb 1.5.3`

Additionally, the `trafic` package includes a utility called `flowsim` that generates three kinds of specific flows:

1. HTTP get (implemented as a generic TCP client/server)
2. QUIC get
3. UDP CBR flow

A.2 The Experimental Setup

The functionality was split up into four VMs, as shown in Fig. 29: two implementing the traffic source and sink respectively (based on `trafic`), one to store statistics that can be obtained directly from them, and one implementing traffic sniffing for in-depth packet analysis. This deployment has also been used as an example in the NEMO recursive VNF draft presented at the IETF-102 [3].

[1] `https://github.com/mami-project/trafic`
[2] `https://software.es.net/iperf/`

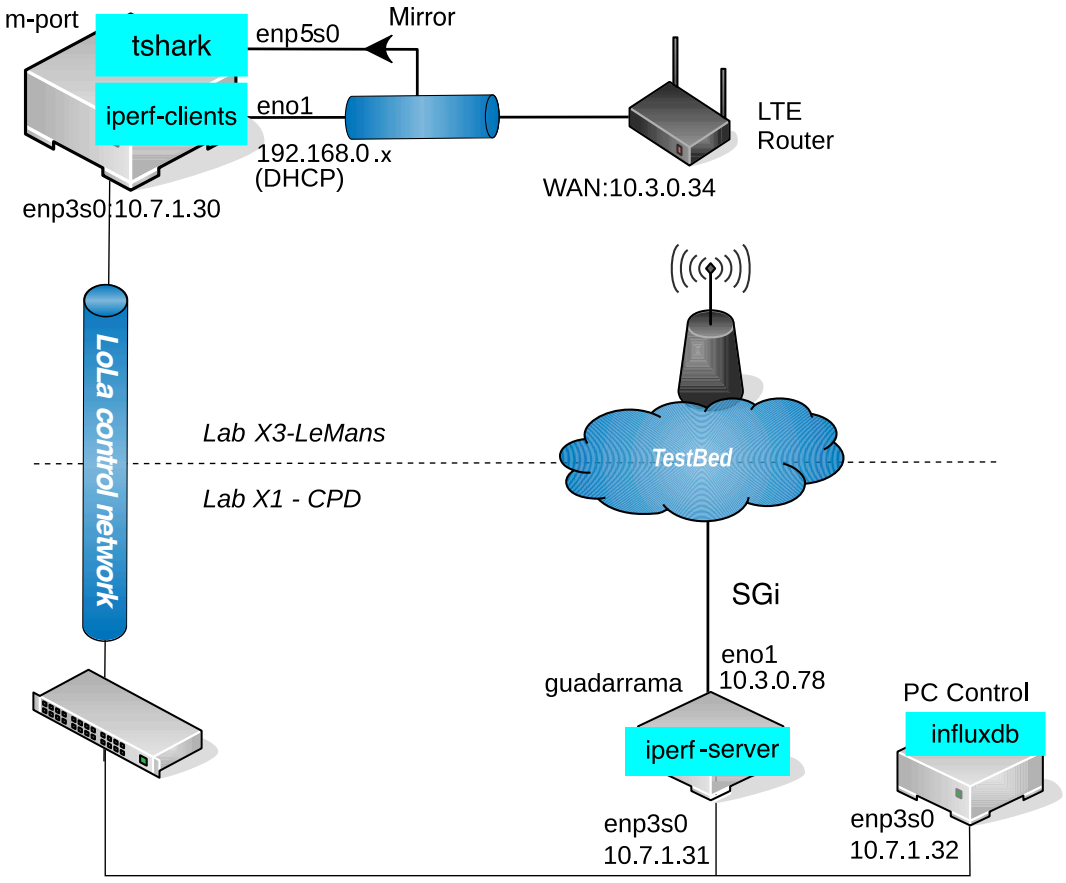


Figure 29: The LoLa testbed deployed at 5TONIC (**Note:** addressing information is just for reference)

Our main aim was to obtain VM definitions that can be used as so-called *Day 0* configurations, i.e., configurations that set up the VM and leave it ready for further control by the Operations Support System (OSS). In the case of the LoLa experiment, we implement external control using WebHooks, thus providing a proof-of-concept for what is known as *Day 1* configuration in the ETSI OSM world [27].

A basic requirement in the experiments is that the VMs run a Linux distribution that covers the dependencies listed above. Depending on the amount of disk and memory available on the virtual infrastructure, two distributions are recommended:

- **Alpine Linux v3.8** Alpine Linux is recommended in systems with limited resources. The boot time is quite low and the responsiveness of the system is good. Images are shipped with all components pre-installed because on-the-fly configuration with `cloud-init`[3] is still in a very poor state. This means that a set of four images need to be prepared and shipped and boot-time configuration relies exclusively on classic network configuration mechanisms like Dynamic Host Configuration Protocol (DHCP). Note: The development and initial tests of the LoLa setup was done using Alpine Linux 3.7/edge, which then has

[3] `https://launchpad.net/cloud-init`

crystallised in the September 2018 release of Alpine Linux 3.8

- **Ubuntu Linux 18.04LTS** Ubuntu Linux is recommended for systems with a reasonable amount of resources. The boot time is reasonable. The recommended installation procedure is to use the standard cloud image from [4] and provide `cloud-init` scripts for each of the 4 VMs in the setup. This allows the whole system to be deployed by orchestration frameworks like OSM or OpenStack. It is also the simplest way to generate OSM compatible descriptors at either VNF or VNF component level that can be then combined into a Network Service Descriptor (NSD) for deployment.

During our measurements, we have faced different difficulties: 1. we discovered that iperf3 behaves erratically when requested to fix the amount of bytes transferred in a TCP session. This bug was reported to the iperf3 project[5]. As of writing, there is no reaction from the developers except the indication that setting an arbitrary value (greated than 1) for variable `testp->multisend` seems to be the culprit. The issue page includes a simple work-around, which, however does not fully correct the problem. This has to be taken into account when using iperf3 in `trafic` to generate the *video-abr* and *webshort* flows described in the `trafic` documentation. 2. In addition, the 5TONIC testbed couldn't provide an implementation of a multi-bearer radio interface in time. We overcame the iperf3 bug, implementing an additional flow generator for `trafic` that allowed us to have controlled TCP flows. However, since we had no control over the 5TONIC deployment schedule, we were finally not able to conduct the experiments in time.

A.3 Support Beyond the Lifetime of MAMI

While this setup was not as heavily used in MAMI experimentation as originally planned, `trafic` has attracted the attention of different research projects that will use, maintain and further develop it beyond the lifetime of the project. `trafic` is currently deployed and used in the 5TONIC facilities for activities like calibrating and testing the infrastructure and the code has been forked by the 5GinFIRE project (`https://5ginfire.eu`), which intends to use and extend it (cf. Fig. 30).

[4] `https://cloud-images.ubuntu.com`
[5] `https://github.com/esnet/iperf/issues/768`

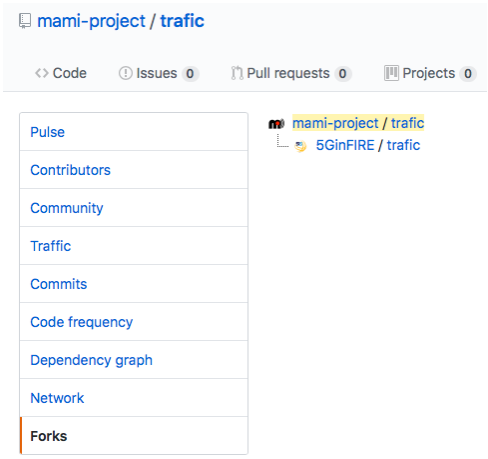


Figure 30: Fork by the H2020 5GinFIRE project

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *SIGCOMM Comput. Commun. Rev.*, 41(4), Aug. 2010.

[2] E. Amir. Unreliable transport. `http://nms.lcs.mit.edu/~hari/papers/CS294/paper/node5.html`, may 1996.

[3] P. Aranda Gutierrez, D. Lopez, S. Salsano, and E. Batanero. High-level VNF Descriptors using NEMO. Internet-Draft draft-aranda-nfvrg-recursive-vnf, Internet Engineering Task Force, Aug. 2018. I-D Exists.

[4] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communication Magazine*, 2018. to appear.

[5] T. Barbette, C. Soldani, R. Gaillard, and L. Mathy. Buliding a chain of high-speed VNFs in no time. In *Proc. IEEE International Conference on High Performance Switching and Routing*, June 2018.

[6] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, May 2015.

[7] S. Bensley, D. Thaler, P. Balasubramanian, L. Eggert, and G. Judd. Data Center TCP (DCTCP): TCP Congestion Control for Data Centers. Technical Report 8257, IETF Secretariat, October 2017.

[8] B. Briscoe. Insights from Curvy RED (Random Early Detection). Technical report TR-TUB8-2015-003, BT, May 2015.

[9] B. J. Briscoe, K. D. Schepper, and M. Bagnulo. Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture. Internet-Draft draft-ietf-tsvwg-l4s-arch-00, Internet Engineering Task Force, May 2017. Work in Progress.

[10] K. De Schepper, O. Bondarenko, I.-J. Tsang, and B. Briscoe. PI^2 : A Linearized AQM for both Classic and Scalable TCP. In *Proc. ACM CoNEXT 2016*, pages 105–119, New York, NY, USA, Dec. 2016. ACM.

[11] P. De Vaere, T. Bühler, M. Kühlewind, and B. Trammell. Three Bits Suffice: Explicit Support for Passive Measurement of Internet Latency in QUIC and TCP. In *Proceedings of the 2018 Internet Measurement Conference*. ACM, 2018.

[12] B. Donnet, K. Edeline, I. R. Learmonth, and A. Lutu. Middlebox classification and initial model. Deliverable D2.1, Measurement and Architecture for a Middleboxed Internet (MAMI), June 2017.

[13] K. Edeline and J. Iurman. mmb repository, 2018. `https://github.com/mami-project/vpp-mb`.

[14] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[15] ETSI. OSM Release THREE Wiki. `https://osm.etsi.org/wikipub/index.php/OSM_Release_THREE`, march 2017.

[16] ETSI OSM. OSM: Reference VNF and NS Descriptors. `https://osm.etsi.org/wikipub/index.php/Reference_VNF_and_NS_Descriptors`, 2017.

[17] G. Fairhurst, T. Jones, M. Txen, and I. Ruengeler. Packetization Layer Path MTU Discovery for Datagram Transports. Internet-Draft draft-ietf-tsvwg-datagram-plpmtud-06, Internet Engineering Task Force, Nov. 2018. Work in Progress.

[18] S. Floyd and V. Jacobson. Random Early Detection gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, pages 397–413, Aug. 1993.

[19] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Sept. 2001.

[20] G. Garca. Day-0, day-1 and day-2 configuration in OSM. https://osm.etsi.org/wikipub/images/7/72/Day-1_and_day-2_configuration_in_OSM_Zero_Touch_Car_-_Gerardo.pdf, March 2018. Talk at the Zero Touch Congress, March 2018, Madrid, Spain.

[21] GianpietroLavado. OSM Hackfest: Adding day-0 configuration to VNFs. https://osm-download.etsi.org/ftp/osm-4.0-four/3rd-hackfest/presentations/20180624%20OSM%20Hackfest%20-%20Session%204%20-%20Adding%20day-0%20configuration%20to%20VNFs.pdf, June 2018.

[22] E. N. Gilbert. Capacity of a burst-noise channel. *The Bell System Technical Journal*, 39(5):1253–1265, Sept 1960.

[23] L. Han and K. Smith. Problem Statement: Transport Support for Augmented and Virtual Reality Applications. Internet-Draft draft-han-iccrg-arvr-transport-problem-01, Internet Engineering Task Force, Mar. 2017. Work in Progress.

[24] S. Hemminger et al. Network emulation with netem. In *Linux conf au*, pages 18–23, 2005.

[25] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. Technical Report 6582, IETF Secretariat, April 2012.

[26] IMDEA Networks Institute. 5TONIC Web site. https://www.5tonic.org, 2016.

[27] A. Israel. OSM Hackfest - Session 7a: Adding day 1/day 2 configuration to your VNF. https://osm-download.etsi.org/ftp/osm-4.0-four/3rd-hackfest/presentations/20180628%20OSM%20Hackfest%20-%20Session%207a%20-%20Adding%20day-1%20and%20day-2%20configuration%20to%20your%20VNF%20-%20Creating%20your%20first%20proxy%20charm.pdf, June 2018.

[28] ITU-T G.144, One-way transmission time. Recommendation, International Telecommunication Union, May 2003.

[29] M. Kühlewind, T. Bühler, B. Trammell, R. Müntener, S. Neuhaus, and G. Fairhurst. A Path Layer for the Internet: Enabling Network Operations on Encrypted Protocols. In *Proceedings of the International Conference on Network and Service Management (CNSM)*. IEEE, 2017.

[30] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[31] M. Mathis. Relentless congestion control. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.184.6382.

[32] M. Mathis and J. Heffner. Packetization Layer Path MTU Discovery. RFC 4821, Mar. 2007.

[33] R. Pan, P. Natarajan, F. Baker, and G. White. Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem. RFC 8033, Feb. 2017.

[34] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *HPSR*, pages 148–155. IEEE, 2013.

[35] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. CUBIC for Fast Long-Distance Networks. Internet-Draft draft-ietf-tcpm-cubic-07, Internet Engineering Task Force, nov 2017.

[36] M. Sargent, J. Chu, D. V. Paxson, and M. Allman. Computing TCP's Retransmission Timer. RFC 6298, June 2011.

[37] K. D. Schepper, O. Bondarenko, I.-J. Tsang, and B. Briscoe. 'Data Center to the Home': Ultra-Low Latency for All. Technical report, RITE Project, June 2015.

[38] K. D. Schepper and B. J. Briscoe. Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay. Internet-Draft draft-ietf-tsvwg-ecn-l4s-id-00, Internet Engineering Task Force, May 2017. Work in Progress.

[39] K. D. Schepper, B. J. Briscoe, O. Bondarenko, and I. J. Tsang. DualQ Coupled AQM for Low Latency, Low Loss and Scalable Throughput. Internet-Draft draft-ietf-tsvwg-aqm-dualq-coupled-00, Internet Engineering Task Force, May 2017. Work in Progress.

[40] S. D. Strowes. Passively Measuring TCP Round-trip Times. *Queue*, 11(8):50:50–50:61, Aug. 2013.

[41] The Fast Data Project. FD.io - The Universal Dataplane. `https://fd.io/technology/`.

[42] D. J. D. Touch. Transport Options for UDP. Internet-Draft draft-ietf-tsvwg-udp-options-05, Internet Engineering Task Force, July 2018. Work in Progress.

[43] J. You, M. Welzl, B. Trammell, M. Kühlewind, and K. Smith. Latency Loss Tradeoff PHB Group. Internet-Draft draft-you-tsvwg-latency-loss-tradeoff-00, Internet Engineering Task Force, Mar. 2016. Work in Progress.

[44] T. Zhou, Y. Xia, P. Aranda Gutiérrez, and D. López. ODL: NEtwork MOdeling(NEMO). `https://wiki.opendaylight.org/view/NEMO:Main`, may 2015.