Published as:
A. Schmidt and T. Herfet, “Transparent transmission segmentation in software-defined networks”, *2017 IEEE Conference on Network Softwarization (NetSoft)*, Bologna, 2017, pp. 1-5. DOI: 10.1109/NETSOFT.2017.8004213

© 2017 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Transparent Transmission Segmentation in Software-Defined Networks

Andreas Schmidt, *IEEE Student Member* and Thorsten Herfet, *IEEE Senior Member*
Telecommunications Lab
Saarland Informatics Campus, D-66123 Saarbrücken, Germany
Email: {andreas.schmidt, herfet}@cs.uni-saarland.de

***Abstract*—Traditionally, network core devices are simple and the complexity is in the end-hosts. With the rise of Software-Defined Networking, this changes and complex functions are moved into the network core. This paper presents *Transparent Transmission Segmentation* (TTS), which is able to improve performance by executing parts of network functions at the core. An implementation for segmenting TCP connections in SDNs is presented, including the network integration and traffic steering process, as well as evidence on its positive effects on latency.**

***Index Terms*—Traffic Engineering and QoS in SDN/NFV, 5G Functional Decomposition and Infrastructure slicing**

I. Introduction

While the interest in multimedia streaming applications is increasing over the last years, and predicted to continue growth [1], many of these are still designed following the *End-to-End* (E2E) design as proposed in Saltzer et al. [2]. Transport protocols, such as TCP, are in use that consider the network path as a single virtual connection, accumulating all parameters of intermediate links in one black box.

Consider a typical scenario, as depicted in Fig. 1, in which a smartphone that is associated with a home WLAN access point is used to consume a video hosted at NetFlix. The access point connects to the Internet via DSL and the video is streamed via *Dynamic Adaptive Streaming over HTTP* (DASH) [3]. Consequently, the first link is resilient in terms of packet loss but has high delay, in contrast to the second one which is lossy but has low delay. As DASH uses HTTP, which by now is mainly implemented on top of TCP, the error control function uses *Automated Repeat reQuest* (ARQ). Given the above link parameters, it is likely that packets from the server arrive at the local access point, but get lost on the last segment. Retransmissions from the source take significantly longer than they would over the second link only. Furthermore, on their route through the Internet, they again contend with other packets for use of links, potentially causing congestion.

The contribution of this paper is threefold:

- We introduce *Transparent Transmission Segmentation* (TTS) as a general concept that can tackle challenges caused by pure E2E operation.
- We describe the implementation and depyloment of *Relays,* that are used to run TTS in SDNs.
- We quantify improvements caused by TTS on latency in different network scenarios.

978-1-5090-6008-5/17/$31.00 © 2017 IEEE

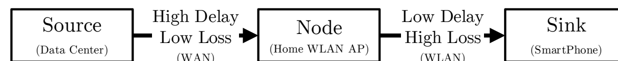

Figure 1. Streaming Scenario with Suboptimal Characteristics for E2E

The rest of this paper is structured as follows: First, Sec. II describes the concept of TTS, and identifies affected network functions and domains. The implementation is described in Sec. III with details on the segmentation software, deployment and integration using SDN, and the segmentation process. Empirical evidence for TTS being beneficial for transmission is given in Sec. IV. Our approach is compared with existing approaches in Sec. V and the paper is concluded in Sec. VI.

II. Transparent Transmission Segmentation

Modern applications have increased throughput requirements and often demand real-time performance. TTS allows to reduce latency and increase throughput, by separating the network into domains.

A. Requirements

Before detailing how TTS can be implemented, it is imperative to properly state the requirements: First, the approach should be transparent to the end-hosts, so that they cannot distinguish between segmented and not segmented connections, except for changes in the observed transmission quality. Neither applications nor operating systems have to change, to achieve broad adoption with ISPs. Furthermore, the implementation effort should be minimal, considering special software to be installed on switches, additional protocols to be run or more devices to be installed. A feasible TTS solution keeps the network mostly untouched, and only adds small software or hardware components. Finally, it should be possible to segmentent transmissions independently, because the performance gain varies depending on link parameters.

B. Domains

The single virtual segment mentioned before does not consider the differences in link parameters and consequences of heterogeneities. TTS divides the network into segments, creating units that are more homogeneous regarding different domains. Considering E2E *latency*, many factors influence the

results, including propagation and processing delays. *Contention*, the major reason for congestion control, effectively happens across the complete path, causing many parties to contend for resources, even though they might share only one common link. Network paths consist of many *buffers*, differing in size, the layer on which they are implemented and fill-states. Finally, *capacities* vary across the network, while the bottleneck capacity is limiting a transmission's performance.

C. Network Functions

The effects of segmenting domains have to be considered with regard to the network functions. TTS is universally applicable to protocols with advanced network functions, regardless of the network layer at which they are implemented. In the following we focus on the functions that are provided by TCP.

Error Control is done in TCP using ARQ. Smaller segments lead to local retransmissions with lower added latency and earlier ACKs, reducing the likelihood of unnecessary retransmissions due to timeouts, which has been proven by Karl et al. [4] and is further evaluated in Sec. IV-C.

Congestion Control probes for available data rate over a path, avoids congestion events and enables multiple parties to agree on a fair data rate share. RTT-dependent TCP congestion control algorithms (CCA), such as Tahoe or Reno [5], can benefit from the RTT reduction due to local small segments. Receiving ACKs in shorter intervals increases the throughput, as more packets can be in flight. Loss events now only affect the local segment's throughput, localizing contention domains, and reducing the likelihood of induced backpressure or buffer underflows as the remaining segments send independently.

Flow Control can, e.g. in cases of embedded IoT end-points, lead to underutilized links, as the throughput is limited by the size of the receiving buffer. Segmenting adds additional buffers at intermediate nodes, which are filled at higher rate and maximize link utilization. Fill levels are communicated quicker, leading to faster reaction of the sending side.

III. Integration into Software-Defined Networks

SDN, together with *Network Function Virtualization* (NFV), provides a general architecture for implementing new networking approaches and paradigms. The TTS approach can be considered as such a network function and hence can be integrated using SDN.

A. Softswitches

Fig. 2 depicts the structure of the switches used, where a single physical device running Linux is turned into a network node with virtual end-hosts. As a general purpose operating system, Linux allows to build lightweight solutions, only containing software necessary for switching and implementing the virtual network. We have used Ubuntu 16.04[1], which ships with *Open vSwitch*[2] (OVS), offering extended switching capabilities and in particular OpenFlow support.

[1]http://releases.ubuntu.com/16.04/
[2]http://www.openvswitch.org/

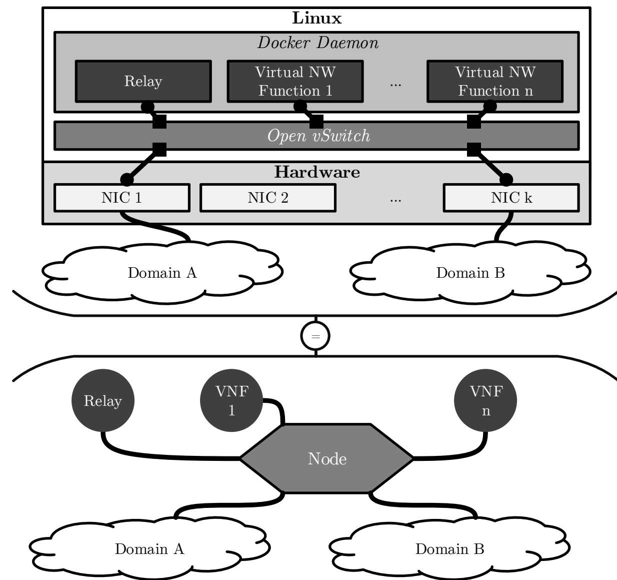

Figure 2. Softswitch (Physical and Logical View)

Docker[3] containers are widely used in the area of data center and cloud operations, as they are a lightweight alternative to traditional *virtual machines* (VM). NFV with virtual machines suffers from the overhead caused by booting and emulating complete systems, which is against the requirements of network functions to be deployed fast. Docker only virtualizes resources required by the specific application, e.g. frameworks or libraries, and reuses resources that are shared between the applications, e.g. the kernel. Thereby it provides a lightweight virtualization solution that keeps an NFV's footprint small and enables short startup times.

B. Relay Implementation (NFV)

TTS is implemented using the *Relay*, which is a pure software solution. This makes it flexible in terms of deployment on both dedicated hardware or inside virtual environments, e.g. using a Docker container as mentioned before. The software is written in plain C so it can be used on many systems even without virtualization. The only dependencies are the *glibc*, providing socket functionality, and *pthreads* for concurrency.

Fig. 3 depicts the internal structure of the relay including its interfaces. Firstly, there are two sides labelled according to the roles of TCP connections, where a client actively opens a connection to the server. Even though this naming indicates a unidirectional stream, the relay is also forwarding data back from the target to the source. Consequently, the relay has a client-side socket and a server-side socket to use for communication. Furthermore, a control socket allows communication with the controller which programs new relayed connections. The relay keeps track of opened ports and recycles those of completed connections.

[3]https://www.docker.com/

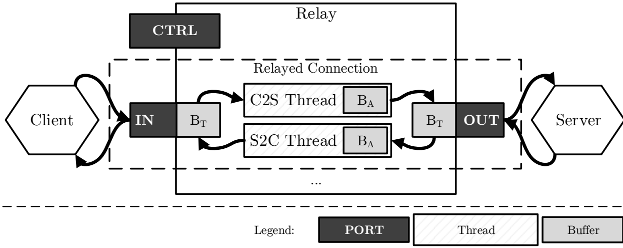

Figure 3. Relay Architecture

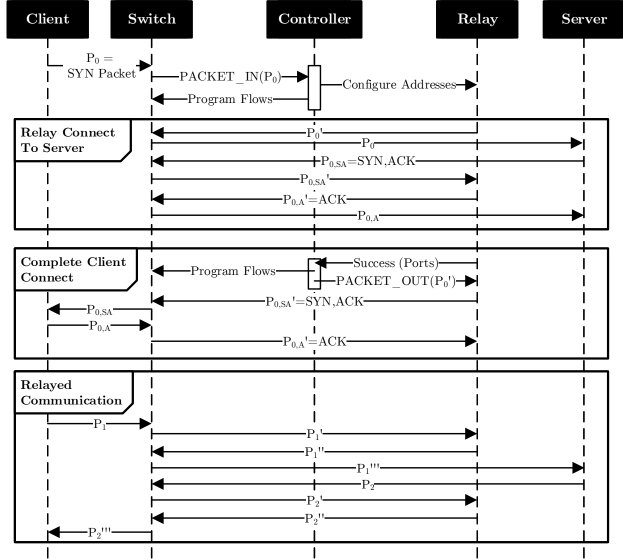

Figure 4. Relaying Establishment Process

The control flow is separated across multiple threads. The main thread is used to wait for incoming controller messages and updates the list of active relayed connections. Every relayed connection has two additional threads, one for each direction of the original TCP connection, to serve both directions at the same time on separate cores.

There are six buffers per relayed connection, where one application-layer buffer per direction serves as a means for the relay to receive data from one socket, store it, and then send it to the opposing socket. There are two transport-layer buffers per connection, namely the TCP send and receive buffer.

C. Relaying Process (SDN)

Before establishing a connection, the controller must be aware of a service (`host:port`) to which traffic should be relayed and where the relays are. While the first information is programmed into the relaying module upfront, using e.g. REST-ful interfaces, the latter is propagated through LLDP messages the relay is sending to announce itself. Relay information includes IP and MAC addresses, control port, and to which node it is attached.

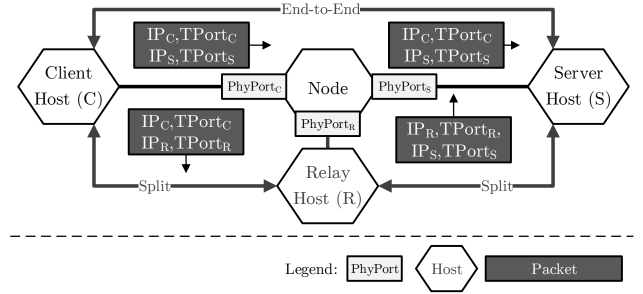

Figure 5. Packet Rewriting (opposite direction is analogue)

When a client is establishing a connection to one of these services, the process depicted in Fig. 4 is executed. This increases the time to setup a connection, but it is common in SDN with reactive routing that the initial packets take longer to reach the destination, while subsequent packets are delivered at normal speeds. The first action is triggered when the client's SYN-packet reaches the first switch, which in turn forwards to the controller (raising a `PACKET_IN` event) and asks how to proceed further. The controller detects that the client wants to consume a service to which communication should be relayed. Consequently, the controller sets the first flows and starts configuring the relay for this new relayed connection. This involves setting ports and buffer sizes, but also telling the relay to establish a connection to the server. While establishing the route, the packets are already rewritten. As soon as the relay connected to the server successfully, additional flows are programmed and the initial SYN is forwarded to the relay. The client can afterwards complete the handshake by having the relay answer the messages. Finally, both parties communicate as usual, unaware that the relay is involved in the communication and forwards the data. Eventually, close messages are relayed, freeing resources at the relay.

This solution can be considered as a man-in-the-middle, hence it needs to be ensured that controller and nodes are not compromised, which is already a necessity for proper operation of the network. The only additional attack vectors introduced by TTS are vulnerabilities in the relay implementation and the TCP stack it uses. As this is a relatively small piece of software with limited capabilities, it is straightforward to secure it.

For proper integration of the relay into connections, network nodes must rewrite and reroute the packets. This guarantees that the transmission is split, even though both parties believe they communicate directly and packets in transit also look like this. In contrast, the relay has direct TCP connections with both end-hosts. Fig. 5 shows this for a single packet from client to server. The node is redirecting the packet to the relay and again redirecting the answer that is coming back. On this short connection to the relay, the packets look as if they were fulfilling these criteria.

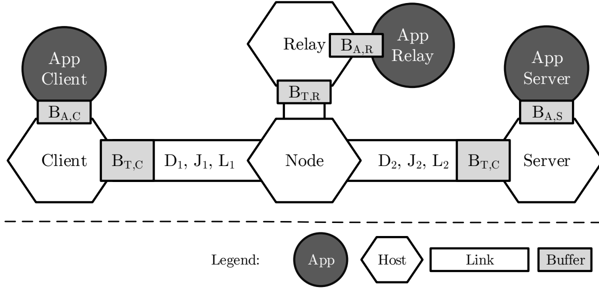

Figure 6. Evaluation Scenario

IV. Evaluation

While Sec. II gives reasons why TTS is superior to E2E, practical evidence is needed to validate this. By first describing the way measurements are executed and how metrics are generated, it is shown that TTS improves performance in typical network scenarios.

A. Measurement Setup and Evaluation Methodology

All measurements are carried out on a desktop PC using Mininet[4] to create a virtual network environment as in Fig. 6. The simulation system has 8 cores and 8GB RAM, of which only fractions are used throughout the tests, so that it is ensured that the simulation results are not due to resource limitations. The link parameters have been set using `netem` on the respective interfaces. Mininet's virtual hosts run on the network stack of the host, sharing its congestion control algorithm, which was TCP CUBIC, the default in Linux. In order to measure the performance, a simple TCP application, written in Go[5], has been used that measures how long it took to send a payload of certain size.

In order to prove the superiority of TTS over E2E in certain scenarios, the following metric is chosen: The transmission tool mentioned above starts a measurement by taking a timestamp and sending a configurable size of TCP payload data into a socket. The server component awaits this data and in particular the byte indicating the end of data. When this is received, the server answers with an application-layer ACK to the sender. As soon as the sender receives this, it takes the second timestamp and computes the difference. Consequently, the result is one application layer RTT, including the transmission time for the packets and including potential retransmits and timeouts. In all scenarios, this time is given in milliseconds if not noted differently.

The measurements per scenario are taken in an interspersed way, hence after each E2E test, the TTS test is performed and vice versa. This ensures that load changes on the simulation system affect both sets of trials, in contrast to a procedure where first E2E and afterwards TTS tests are executed.

[4]http://mininet.org/
[5]https://golang.org/

Table I
Lossy Last-Mile Results (200 trials per approach)

$L_1[\%]$	$L_2[\%]$	μ_{E2E}	μ_{TTS}	σ_{E2E}	σ_{TTS}	A_{12}
10^{-6}	1	6.534	5.923	0.782	0.073	0.884
10^{-6}	10^{-2}	5.864	5.672	0.261	0.071	0.755
10^{-6}	10^{-6}	5.924	5.722	0.295	0.078	0.748

B. Significance Testing

In [6], the comparison between E2E and TTS has been made based on differences in statistical measures of Gaussian distributions. While this is legitimate as a first means to quantify the effects of TTS, it proved that the distribution of delivery times is not strictly Gaussian. As a variety of protocols interact over different operating systems and routing nodes, the theoretical distribution cannot be specified with sufficient confidence to gain reliable results. Consequently, non-parametric statistics have been added to compare E2E and TTS, as well as evaluate the significance of the results.

Comparisons between the two approaches have been carried out using the A_{12} metric as defined by Vargha et al. [7]. Measurements series for E2E (Index 1) and TTS (Index 2) are combined in the following way: Each element in the first series is put together with each element of the second into a tuple. Each tuple is afterwards transformed into either a zero, if E2E is smaller than TTS, a one if TTS is smaller, and 0.5 if they are the same. The mean of this list is then the resulting metric. If the result is close to zero, E2E is better, while it being close to one means that TTS is better. On these results, the Mann-Whitney U-test [8] is used to give the likelihood of the observations being due to pure luck. As it is also only considering whether a certain variable is stochastically larger than another, it is well suited to support the A_{12} measure by providing a p value.

C. TTS Effects on Lossy Last-Mile

Regarding error control in a scenario as mentioned in Sec. I, the expectation is that the higher the loss rate on the second link is, the better TTS is compared to E2E. In order to provide further evidence that error control is positively affected, the following scenario is evaluated: Delays and Jitters are kept small ($D_1 = J_1 = D_2 = J_2 = 1ms$), so that other effects due to shortened RTT do not play a role, and the data rate is $R = 100MBps$. The application layer buffer at the relay is $B_{A,Relay} = 3000$, which is twice the MTU of 1500. Transport layer buffers are $B_{T,Relay} = B_{T,Recv} = 5MB$ which is also the size of the payload to be sent so that flow control has no impact. Regarding error probabilities, the one on the first link is intentionally kept small $L_1 = 10^{-6}\%$, while the one on the second link is varied. The remaining parameters and results are in Tab. I, where also the mean values $\mu, [\mu] = sec$ and standard deviation $\sigma, [\sigma] = sec$ are given. The confidence values are omitted from the table as they are $p < 10^{-10}$ for all cases.

From the results, it is evident that TTS can keep the variation of delays low, because the packets over the first link

Table II
HIGH JITTER RESULTS (200 TRIALS PER APPROACH)

J_1	J_2	μ_{E2E}	μ_{TTS}	σ_{E2E}	σ_{TTS}
1.0	1.0	11.697	6.685	0.010	0.116
10.0	10.0	11.746	7.174	0.039	0.139
100.0	100.0	35.134	8.085	4.652	1.148

are only sent when they have been lost there. Intuitively, the lower the error rate the lower the variations should get. The last scenario does not follow this trend, most likely due to imprecisions on the simulation system, especially because both E2E and TTS faced higher delays.

D. TTS Effects with High Jitter

It is anticipated that TTS is likely to increase jitter compared to E2E due to reordering. This scenario proves that for small jitter values this is true, but as soon as the jitter increases and timer expiries come into play, TTS is less affected than E2E. Again, the links had $R = 100MBps$ and low loss $L_1 = L_2 = 10^{-6}\%$. The base delay was $D_1 = D_2 = 100ms$ and the jitter has been varied to be 1%, 10% and 100% of the delay. Payload and buffer sizes are chosen as in the previous evaluation, hence $Payload = B_{T,Relay} = B_{T,Recv} = 5MB$ and $B_{A,Relay} = 3000$. Tab. II shows the different jitter values and the results in terms of mean $\mu, [\mu] = sec$ and standard deviation $\sigma, [\sigma] = sec$ for both cases.

As can be seen, the standard deviation is higher for TTS, but grows slower with the link jitter in comparison with E2E. Regarding the evaluation metric of TTS causing smaller delivery times, it was $A_{12} = 1.0$ for all cases and $p < 10^{-10}$.

V. RELATED WORK

Saltzer et al. [2] state that to reliably implement a network function across hosts, it has to be done E2E, not relying on intermediate systems to provide it. Nevertheless, performance gains can be achieved when intermediate systems do so, which is further addressed by Moors [9]. The approach presented here is an incarnation of *Performance Enhancing Proxies* (PEP), which have been first proposed in RFC 3135 [10]. Among the first implementations on the transport layer is *Split-TCP* [11], tackling challenges with special network scenarios, e.g. links with large bandwidth-delay product. This approach does not provide transparency as our approach does, but instead replaces original TCP. RTT-independent congestion control algorithms such as TCP CUBIC, solve some issues, but leave others open that can only be addressed with segmentation. As SDN allows to implement certain functions inside the network, our PEP is integrated using SDN and deployed via NFV, in contrast to other solutions that change the router implementation [12]. SDN and NFV are typically employed inside and in between data centers (DC) to improve performance and reduce costs, which is why Pathak et al. [13] using Split-TCP for DC to DC communication to reduce service time for end-users. The relay implementation is similar to Siracusano et al. [14], but using Docker instead of a Xen minikernel to virtualize the function. The authors use cloud providers to install relays on the Internet, while our approach works inside one autonomous system, for instance one SDN operated by an ISP.

VI. CONCLUSION

Transparent Transmission Segmentation is a paradigm that should be thoroughly studied, especially in the context of fog-computing, where more of the systems' intelligence and complexity is moved close to the end-devices and in particular inside the network core. Especially with lightweight virtualization platforms, such as Docker, the deployment is simplified which can lead to broader adoption.

ACKNOWLEDGMENTS.

This work was partly supported by the German Research Foundation (DFG) as part of the priority programme SPP 1914 "Cyber-Physical Networking" under grant HE 2584/4-1. We thank the reviewers for their valuable feedback. We thank the Telecommunications Lab team at Saarland Informatics Campus for in-depth discussions on the TTS paradigm, its implications and implementation.

REFERENCES

[1] Cisco Systems, "Visual Networking Index," tech. rep., 2015.
[2] J. H. Saltzer, D. P. Reed, and D. D. Clark., "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, 1984.
[3] T. Stockhammer, "Dynamic adaptive streaming over HTTP: standards and design principles," in *Proceedings of the second annual ACM conference on Multimedia systems*, pp. 133–144, ACM, 2011.
[4] M. Karl and T. Herfet, "Transparent Multi-hop Protocol Termination," *28th IEEE International Conference on Advanced Information Networking and Applications (AINA-2014)*, 2014.
[5] M. Allman, V. Paxson, and E. Blanton, "RFC 5681 - TCP congestion control," tech. rep., 2009.
[6] A. Schmidt and T. Herfet, "Improving Multimedia Streaming from the Network's Core," *International Conference on Consumer Electronics (ICCE) - Berlin*, 2016.
[7] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
[8] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.
[9] T. Moors, "A critical review of 'end-to-end arguments in system design'," in *IEEE International Conference on Communications (ICC)*, vol. 2, pp. 1214–1219, IEEE, 2002.
[10] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby, "RFC 3135 - Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations," tech. rep., 2001.
[11] S. Kopparty, S. V. Krishnamurthy, M. Faloutsos, and S. K. Tripathi, "Split tcp for mobile ad hoc networks," in *Global Telecommunications Conference. GLOBECOM'02*, vol. 1, pp. 138–142, IEEE, 2002.
[12] S. Ladiwala, R. Ramaswamy, and T. Wolf, "Transparent TCP acceleration," *Computer Communications*, vol. 32, no. 4, pp. 691–702, 2009.
[13] A. Pathak, Y. Wang, and C. Huang, "Measuring and evaluating TCP splitting for cloud services," *PAM'10 Proceedings of the 11th international conference on Passive and active measurement*, pp. 41–50, 2010.
[14] G. Siracusano, R. Bifulco, S. Kuenzer, S. Salsano, N. B. Melazzi, and F. Huici, "On-the-Fly TCP Acceleration with Miniproxy," *arXiv preprint arXiv:1605.06285*, pp. 44–49, 2016.