HAL
open science

A Zero-Touch Solution for Transport Layer Adaptation to Applications and Networks

El-Fadel Bonfoh, Samir Medjiah, Christophe Chassot

To cite this version:

El-Fadel Bonfoh, Samir Medjiah, Christophe Chassot. A Zero-Touch Solution for Transport Layer Adaptation to Applications and Networks. 20th International IFIP Networking Conference, Jun 2021, Espoo (virtual), Finland. 10.23919/IFIPNetworking52078.2021.9472837 . hal-03270599

HAL Id: hal-03270599

https://laas.hal.science/hal-03270599

Submitted on 25 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Zero-Touch Solution for Transport Layer Adaptation to Applications and Networks

https://vtl-project.net

El-Fadel Bonfoh
LAAS-CNRS
University of Toulouse
Toulouse, France
efbonfoh@laas.fr

Samir Medjiah
LAAS-CNRS
University of Toulouse
Toulouse, France
medjiah@laas.fr

Christophe Chassot
LAAS-CNRS
University of Toulouse
Toulouse, France
chassot@laas.fr

***Abstract*—TCP is not necessarily suitable for all contexts. It is well-known that, in some context, the "simple" fact of replacing TCP with another Transport protocol clearly improves the performance of the application, especially in terms of throughput. However, TCP remains the most widely used Transport layer (L4) protocol on the Internet: nearly 90% of applications are based on it. *Why*? Our analysis is that the need to rewrite the TCP application before using any new protocol "*partly*" explains the low adoption of new protocols despite TCP's performance limits. Indeed, constant modification of the application is tedious and, above all, a potential source of instability. This raises a second major question: *can we, transparently to the application, replace at runtime TCP with another protocol X more suitable to the application requirements and the actual network condition*? In this paper, we design and implement VTL to provide a concrete and practical answer to this question, i.e., the TCP reconfiguration at the runtime and the choice of the best alternative to it. Based on a series of experimental scenarios, we show the correctness of VTL and that it significantly improves TCP applications' performances.**

***Index Terms*—TCP/IP; eBPF/XDP; Runtime reconfiguration; Protocol selection.**

I. Introduction

The Internet communication model, known as the TCP/IP architecture, is based on two fundamental protocol layers: the Network layer implemented by the IP protocol and the Transport layer (L4), which most popular protocol is TCP[1]. Initially designed to meet the requirements of historical Internet applications such as FTP or HTTP on wired networks, it has since been shown that TCP is conceptually unsuitable for taking into account new generation applications' requirements and many emerging networks' properties [1].

In response to TCP limitations, several research efforts have been carried out for several decades. They have led to a plethora of protocol proposals (from the early IETF standards such as DCCP [2], SCTP [3], etc. to more recent proposals like DCTCP [4], QUIC [5], etc.). Applications can invoke each of these protocols via *dedicated* APIs. The IETF working group TAPS [6] has been promoting a *service-oriented* approach for several years to decouple service invocation from protocol invocation. This approach, whose first implementation is the NEAT framework [7], consists of replacing the dedicated APIs with a generic *Transport services interface* (a.k.a. a service-oriented API). However, till today, TCP remains the most widely used protocol (almost 90% of Internet applications are based on TCP [8]). All alternatives to TCP, including its own extensions such as Hybla [9], have seen limited use. *Why?*

We argue that *one*[2] of the main reasons is that replacing TCP requires a modification of the application, especially its access interface to the underlying Transport services. For instance, TCP applications (dubbed as *legacy* applications) are based on the *socket API* [10]. This API is designed so that the application programmers are required to *explicitly* choose the protocol at the application's design-time (i.e., when the code is written). Consequently, programmers must modify their applications' code to adopt any new protocol other than TCP. This latter corollary might be a factor of increasing complexity and a potential source of instability since it is necessary to rewrite the application each time a new protocol solution is released and best matches the application's needs.

Based on these observations, the approach explored in this paper consists to *transparently* intercept and redirect TCP connections to another protocol *X* that best suits the application's requirements and the network state. The *transparency* property refers to the fact that the application requires zero modification. The realization of our approach requires addressing two main questions: *how to replace TCP transparently*? and *which protocol is the best alternative to TCP*? In addition to the thorough evaluations carried out during our journey, answering the latter questions is the core of this paper's contributions which are summarized as follows:

- We propose a technique allowing to replace at runtime TCP by another protocol *X*. We realize it *transparently*, i.e., there is no need to rewrite TCP-based applications.
- We propose a method that must ensure the selection of the best alternative to replace TCP. This choice is driven by a

[1]Throughout this paper, unless otherwise stated, we will use TCP to designate TCP Cubic (the default version of TCP in Linux OS).

[2]Besides, there are often on the data path middleboxes that can drop packets other than TCP or UDP. These concerns are out of the scope of this paper. For now, we preconize systematic fallback to TCP in case of middlebox rejection.

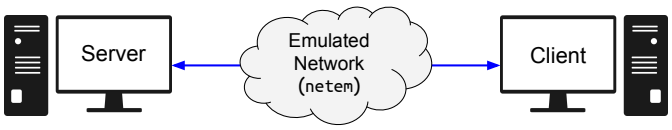


Fig. 1. The considered point-to-point topology for experiments. The link between Client and Server emulates either a classical terrestrial Internet or a satellite Internet link.

set of machine learning models, namely decision trees that we trained to feed our decision algorithms. The attributes of the decision trees are the applications requirements and the network conditions. Therefore, prior to the selection of the most appropriate L4 protocol, we introduce *(1)* a profiling method that allows inferring the requirements of the (legacy) application and *(2)* a parsimonious monitoring that is useful to estimate the state of the network in terms of RTT, loss rate and maximum available bandwidth.

We realize the above contributions within the VTL system that we fully designed and implemented. After this introduction, Section II presents a motivation example to the need to craftily select the alternative protocol to TCP during data transfer. In section III, we presented the design and implementation of the proposed approach. In particular, in subsection 3.A, we detail how we perform the "transparent" replacement of TCP by another L4 protocol. Then, in subsection 3.B, we describe the application profiling method and the network state estimation approach. In Section IV, we carry out thorough evaluations of the proposed approach in order to assess its functional properties and performances, namely the delay of the TCP replacement and data redirection operations, the precisions and recalls of the constructed models, and the estimation of the gain on applications of the proposed solution. Finally, Section V and Section VI conclude the paper with a discussion on the related work and a summary of learned lessons.

II. Motivation Example

Let us consider the following preliminary experiments. Fig. 1 presents the point-to-point topology used for the experiments. Here, the link between the two hosts emulates either the classical terrestrial Internet or a satellite link. Section IV describes in detail the network links configuration and emulation tools as well as the values of the network parameters (delay, bandwidth, and loss rate). The performance criterion under observation is the throughput.

Let us first consider the satellite link. Without packet loss, TCP has equivalent performance to QUIC and to Hybla that is more adapted to satellite links [9]. However, once the link starts by experiencing losses, we notice a significant TCP throughput degradation. As it can be deduced from the results reported in Fig. 2, an application using TCP could, on average, get 3˜4x better throughput on a satellite link if it used Hybla instead. Now, assume that in this context, instead of using Hybla, we selected QUIC. The results (Fig. 2 (a)) show that the performance is not only suboptimal compared to Hybla, but also, and more importantly, that it is worse than the initial

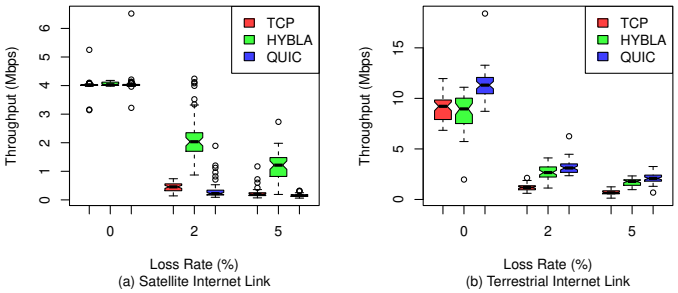


Fig. 2. L4 protocols' performances under various network conditions. Boxes span the 25th to 75th percentiles, with a notch at the median.

performance of the application under TCP. In simple terms, QUIC might perform worse than TCP on the satellite link, when data packet losses occur.

From these first results, we could speculate that it would be enough to use Hybla continually as the alternative to TCP. Nevertheless, let us resume the same experiments on a terrestrial link. Here again, without data packet losses, TCP remains near equivalent to the QUIC and Hybla protocols. However, as soon as the network link suffers its first packet losses, a TCP application that would switch to QUIC will achieve about 1.5˜2x better performance. Further, the results (Fig. 2 (b)) also demonstrated that contrary to the satellite link, QUIC presents better performance than Hybla and seems the best alternative to TCP in the terrestrial network context.

The more we continue the experiments by changing the state of the network and the requirements of the application, the more we observe that the best alternative to TCP changes regularly. This preliminary assessment allows us to validate TCP's performance limitations in specific environments, and demonstrates that replacing TCP may or may not be justified depending on the context. Hence, the interest of having an approach allowing to choose the appropriate protocol *X* because, as we have just seen in the above example, the protocol *X* might not be the same in all network and application contexts and could even perform worse than TCP.

III. Proposed Approach and System Design

A. Transparent and dynamic replacement of TCP

1) Background

VTL relies on the eBPF [11], recently introduced in the Linux OS. eBPF is an extended version of BPF [12] and it allows injecting bytecode at runtime within the OS kernel. Its usage scenarios cover filtering, networking, systems' security, etc. The infrastructure of eBPF is constructed around three major elements: *maps, tail calls,* and *helper functions*.

Maps are data structures storing a set of {*key, value*} pairs used to exchange data either between user-space programs and in-kernel eBPF programs or between eBPF programs running at different points of the kernel. Maps, often attached/pinned to the root file system (i.e. `/sys`), are also useful to ensure data persistence between successive invocations of eBPF programs. In its early versions, eBPF limits each program to a maximum size of 4096 BPF instructions. In order to overcome this

size limitation, eBPF integrates the concept of *tail calls* that could be used to chain up to 32 different eBPF programs; tail calls feature is an enabler of modularization's implementation. Nevertheless, it is worth noting that since Linux version 5.2.0, released in 2020, an eBPF program can contain up to 1M (one million) instructions. Basically, *helper functions* define a list of functions that an eBPF program can call during its execution. Thanks to helper functions (and eBPF verifier), access to kernel by eBPF programs is strictly controlled to prevent OS damage. In other words, helper functions allow eBPF programs to interact directly with the kernel safely.

Each eBPF program that is deployed inside the OS kernel has a specific type and must be attached to a *hook point*, also known as a *kernel event* (incoming packet, system calls, socket operations, etc.). Then, each time the event occurs, the eBPF program attached to it is executed.

2) VTL Hooker component

Hooker goals and requirements. *Hooker* is the component of VTL that achieves the dynamic and transparent replacement of TCP by another protocol *X*. To achieve this objective, the *Hooker* component must interrupt TCP's execution path. At data sending, once the application calls into the `send()/sendmsg()` function, *Hooker* must take control of the packets before the kernel network stack. Therefore, it is necessary to place a hook point on the `tcp_sendmsg()` function so that each time this latter function is invoked, the *Hooker* component executes a dedicated program before the kernel. As for incoming packets, they should be intercepted as soon as they arrive at the network interface card (NIC), here also, to avoid their control by the kernel network stack. The conceptual and technical choices we made to meet these different specifications are described below.

Functional Architecture Overview. Resulting from the above requirements, Fig. 3 depicts the internal structure of *Hooker* and its interactions with the kernel network stack as well as with the legacy applications. Conceptually, *Hooker* is separated in three main subcomponents: *hooker_userspace*, *hooker_egress*, and *hooker_ingress*. As its name suggested, *hooker_userspace* is a normal program running in user-space and that, among other tasks, is in charge of creating and configuring sockets, namely the redirection socket (`redir_sock`) and the socket of the selected Transport protocol *X*. The rest of *Hooker's* subcomponents, i.e. *hooker_ingress* and *hooker_egress*, are eBPF programs and as such, they are executed in the kernel-space and deployed once *Hooker* is activated. The eBPF program composing *hooker_ingress* is an `XDP` program that attaches to the NIC in order to process as early as possible all incoming data packets.

Design choices discussion. In the architecture illustrated in Fig. 3, the choice we made was to pass the *hooker_userspace* in the user-space. This choice, which initially meets a proof-of-concept purpose, opens up in a more global perspective the possibility of using protocols deployed both in kernel-space and in user-space such as QUIC. At the price of higher implementation complexity, it is quite conceivable to bring the *hooker_userspace* subcomponent back into kernel-space.

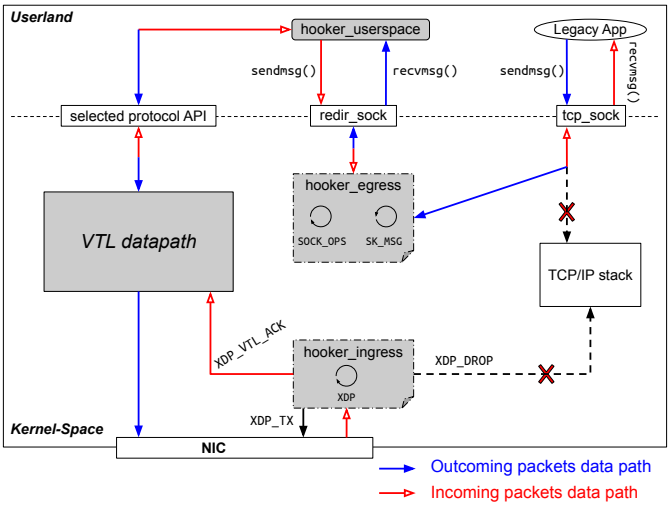


Fig. 3. VTL *Hooker* Component Internal Structure.

Our current hypothesis is that this could improve performance (which however, as we will see later, remains at an acceptable level with the current implementation choice of leaving the *hooker_userspace* in user-space). Though, pushing back the *hooker_userspace* in kernel-space will deprive us of the use of user-space protocols like QUIC for which, ***no*** kernel implementation exists as of this paper writing.

Legacy Application Data Paths. The internal structure of *Hooker* component also depicted the application data packets paths from the `send()/sendmsg()` call to the transmission over the NIC and vice versa. Once it is activated, *Hooker* attaches three different types of eBPF programs at various levels of the network stack: *(1)* a `SOCK_OPS` program attached to the *root* `cgroupv2` [13], *(2)* a `SK_MSG` attached to a `SOCKMAP` at the socket layer, and *(3)* an `XDP` program placed at the NIC to process incoming data packets. By leveraging the hierarchical model of cgroups, *Hooker* is able to process at the socket layer any ingress and egress data packets of all TCP application processes running on the end-system. *Hooker* maintains several maps, especially the `SOCKMAP` that keys are used by the *hooker_egress* programs to identify the right socket towards which the packet data must be forwarded to. Furthermore, the `SOCKMAP` is helpful to keep a trace of applications whose packet data should be intercepted and redirected by *Hooker*. Each time a connection is established or closed by one process, the map is updated by *hooker_egress* thanks to the `SOCKS_OPS` bpf program attached to `cgroupv2`. In addition to `SOCKMAP` updating at the connection opening, the `SOCK_OPS` bpf program is used to add to the SYN packet a VTL_COMPLIANT option that, as its name suggested, is useful to advertise to the receiver that the sender is VTL compliant. Every time the TCP application process sends data by the invocation of the `sendmsg()` function upon the TCP socket, the `SK_MSG` bpf program running by *hooker_egress* intercepts the data packet and rewrites it if necessary thanks to the helper function `bpf_msg_push_data()`. Then, to redirect the egress

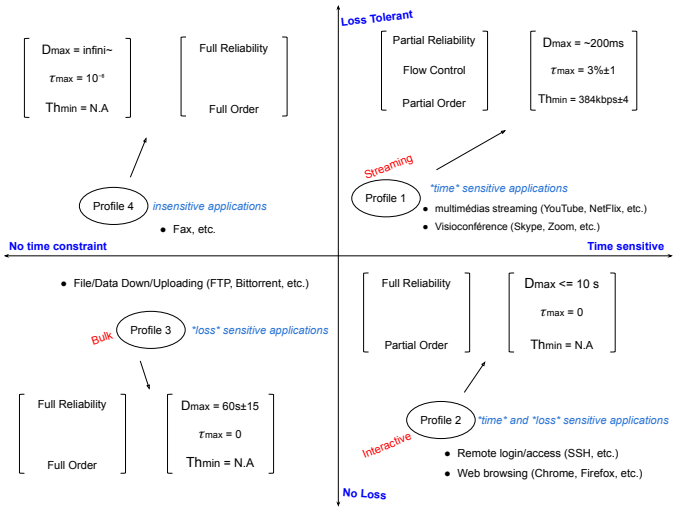


Fig. 4. Applications profiles based on ITU recommendations. The associated requirements of each profile are also shown.

data packets to the redirection socket, *hooker_egress* program leverages the `bpf_msg_redirect_map()` helper. At the incoming of data packets, *hooker_egress* uses the same helper to redirect the packets to the TCP socket. The redirection socket is created and maintained by *hooker_userspace* which will use the `recvmsg()` operation to get the redirected data packet and send it to the VTL datapath that should emulate the selected Transport protocol functioning. At the receipt of a data, as soon as the NIC receives the data packet, the `XDP` bpf program running by *hooker_ingress* intercepts the data packet and processes it by issuing the right verdict. The *hooker_ingress* program can drop the packet data (`XDP_DROP`), redirect it to the same network interface card (`XDP_TX`), or, as currently done, pass the packet to the ingress VTL datapath (`XDP_VTL_ACK`) for further processing.

B. Protocols selection

1) Receiver-driven application profiling

The purpose of profiling is to identify the nature of the TCP application. It permits to infer the requirements of the application. We have established the profiles on the basis of data from the ITU recommendations [14]. To each profile, we associate requirements expressed in terms of Transport services and QoS parameters. The profiling, driven by the server (receiver of the connection), is initiated as soon as the first TCP packet (SYN) is received and continues over the following nine[3] packets. When profiling is successfully completed, the application is classified into one of the following profiles (see Fig. 4).

Profile 1: time-sensitive applications; e.g., multimedia streaming applications (YouTube, NetFlix, etc.) or videoconferencing applications (skype, zoom, etc.). Transport service requirements associated with this application profile are: partial reliability, partial order, and flow control to contribute into the jitter management. The multimedia streaming applications

[3]Profiling is attempted on the first ten packages. This number is arbitrary but higher than the recommendations in [15].

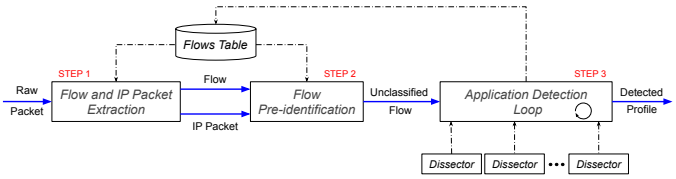


Fig. 5. Application profiling pipeline.

might tolerate a maximum delay of 10 seconds whereas the delay allowed by videoconferencing applications fluctuates between 10 milliseconds and several hundred milliseconds. The time-sensitive applications could experience a loss rate between 2 and 4%. However, it is worth noting that this profile of applications rarely uses TCP.

Profile 2: time and loss-sensitive applications; e.g., interactive applications such as remote login based on Telnet or SSH, web browsers (Chrome, Firefox, etc.), or online games (Call of Duty, Fortnite, etc.). The Transport services required by this application profile are: total reliability and partial order. The associated QoS parameters in terms of delay are a fraction of a second for remote login and online games whereas web browsing could accept delaying up to 10 seconds. The applications of this profile do not allow any loss of data.

Profile 3: loss-sensitive applications; e.g., (large) file transfer applications based on FTP/HTTP or BitTorrent, and text messaging (Facebook Messenger, WhatsApp, etc.). For these applications, Transport services with total reliability and order are required. They do not tolerate any loss of data. On the other hand, these applications are less constraining with regard to the delay, which can go beyond 60 seconds.

Profile 4: insensitive applications; e.g., Fax. These applications are the least constraining in terms of packet loss and data transit delay.

The pipeline of application profiling by packet classification is shown in Fig. 5. The identification of the TCP application takes place in three main stages.

(1) Flow and IP packet extraction. In this work context, a flow is basically defined by the tuple $\{ip_{\text{src}}, ip_{\text{dst}}, port_{\text{src}}, port_{\text{dst}}\}$. The L4 protocol type information is not "necessary" because only TCP packets are processed, so it is impossible to differentiate flows based on this information. During this first step, matching between the intercepted raw packet and the flow table enables the extraction of the flow to which the packet belongs. If the packet does not belong to any stream in the table, a new stream is created. An IP packet is then extracted from the

TABLE I
NETWORK PROFILES BASED ON THE LINK QUALITY PARAMETERS.

	RTT	*Bandwidth (B.W)*
Long-delay Networks (LDN, e.g. Satellite)	>= 500 ms	4 Mbps
Terrestrial Internet Links	50 ms to 500 ms	100 Mbps
LAN (e.g. Internal D.C, home network)	<50 ms	100 Mbps to 100 Gbps

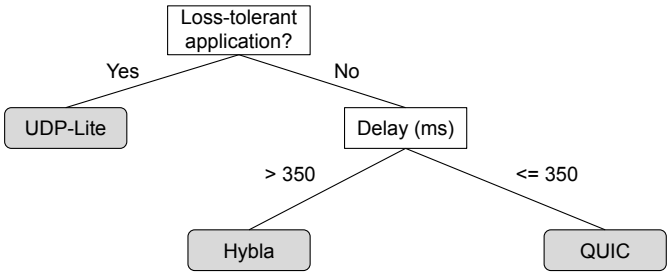


Fig. 6. Simplified decision tree to select the most appropriate protocol. Leaf nodes (gray box) represent classes, whereas internal nodes (white box) represent the attributes.

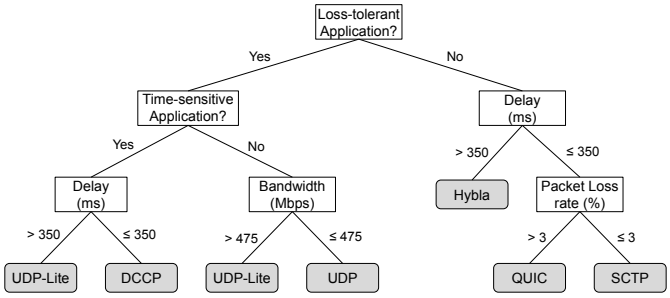


Fig. 7. Extended decision tree to select the most appropriate protocol. Leaf nodes (gray box) represent classes, whereas internal nodes (white box) represent the attributes.

raw packet by removing the L2 header of the packet. This phase's final result is a flow and an IP packet ready to be used in the second phase.

(2) Flow pre-identification. The flows in the table might be already classified or not. A flow is classified when the application to which it belongs has already been detected. Therefore, the purpose of this phase is to directly retrieve this information from the table rather than systematically and blindly launch the application detection loop.

(3) Application detection loop. If the flow is not yet classified, either it is a new one or the flow's first packets have not been sufficient to identify it. The detection loop is based on a hybrid approach to identify the application whose flow it receives: it integrates the signature-oriented approach based on protocol dissectors and the standard method based on port number mapping. It first attempts to identify the application by contrasting the flow to a set of predefined protocol dissectors. Dissectors are snippets of code that identify a specific protocol by reading/processing the entire IP packet (headers and payload included). For instance, an HTTP protocol dissector might fetch the "GET" string in the IP packet to determine whether the flow is an HTTP flow. Once a dissector correctly identifies the flow, the loop stops. If the application is not identified, the next packets of the stream (up to the 10th packet)[4] are used to attempt a new detection of the application and its classification in one of the four profiles described above.

2) On-request network monitoring

In addition to the application's requirements, the network state is used to drive the selection of the best protocol *X* to replace TCP. To do this, we associate to each network link a state or profile characterized by three main parameters: $\{[RTT_{\text{min}}, RTT_{\text{max}}], BW_{\text{max}}, loss_{\text{moy}}\}$. The RTT_{min} (resp. RTT_{max}) denotes the minimum (resp. maximum) round-trip-time experienced under the network. The $loss_{\text{moy}}$ is the average rate of packet loss, and BW_{max} is the maximum bandwidth available within the network link. In Table I, we can see that typical LDN networks such as satellite networks have profile P1 = $\{[500ms, \infty[, -, 1Mbps\}$ [16]. Note that the loss parameter is neither static nor closely bound to a specific network profile but depends more on the network's congestion state. Therefore, it is possible (probably the fact) to experience more data packet losses under congested wired-LAN than non-congested wireless-LAN.

Since the bandwidth (the incoming data rate, in fact) estimation does not require any packet injection into the network, the monitoring component *continuously* captures a copy of the incoming packets to deduce the network link's bandwidth. However, we estimate the RTT and loss rate values by injecting out-of-band, albeit lightweight, ICMP ECHO/REPLY packets on the network. To minimize the impact of monitoring on the network traffic load, monitoring these two parameters is only triggered on demand through a set of functions exposed by the internal API of the monitoring component. The caller of the monitoring component has the possibility to specify the periodicity of the monitoring and its duration. The period defines the time interval between packets injection for calculation of RTT and loss rate. The larger the interval, the less expensive the monitoring is at the price of the estimation's accuracy.

3) Construction of Decision Tree Models

Application profiling and network monitoring are prerequisites to the selection of the most suitable alternative L4 protocol to TCP. They provide two information: the *application profile* (i.e., its requirements) and the *network state*. This information is the attributes (i.e., the inputs) of decision tree models on which are based the selection rules of the most appropriate protocol to replace TCP. These decision trees feed and represent VTL's knowledge base that dictates the selection rules based on the above two information attributes.

Dataset. For the models' training, we generated a labeled *dataset* of more than a hundred cases. The labels or classes are the L4 protocols and the attributes, as stated above, are the application requirements and the network conditions. Following the classical approach, we separated the dataset into two main parts: the *training dataset* ($\simeq$ 66% of the initial dataset) and the *test dataset* ($\simeq$ 33% of the initial dataset). As its name suggests, the training dataset is the part of the dataset used to train the models. Additionally, it allows evaluating the trained model's ability to classify correctly the already seen cases. What about the unseen cases? The answer to the latter concern is the task of the test dataset. It permits us to evaluate the trained model's prediction quality, i.e., the precision at which the model can classify unseen cases. The dataset is generated from extensive evaluations of several IETF

[4]See footnote 3

L4 protocols for diverse application requirements and network conditions. We attribute the classes (protocols) to each based on these experiments and the literature's recommendations. For instance, it is commonly accepted that UDP and UDP-Lite are appropriate to loss-tolerant applications. Suppose two or more protocols satisfy application requirements and feet the network characteristics. In that situation, the performance criterion used to assign a label to the case is the throughput experienced during data transfer.

Fig. 6 and Fig. 7 illustrates examples of the outcomes of the training stage. In the instance of Fig. 6, the application is considered to be either loss-tolerant (profiles 1 and 4, Fig. 4) or not (profiles 2 and 3, Fig. 4). This assumption leads to a more simplified decision tree that, as we will see later, could provide satisfactory classification and prediction quality compared to a more extended decision tree. The complete evaluation of these models' quality and their use benefits are extensively evaluated and presented in Section IV.

TABLE III
CONFUSION MATRICES OF THE DECISION TREE *model1*.

(a) training dataset

		Predicted						
		Hybla	UDP	UDPLite	SCTP	QUIC	**Precision**	**Recall**
Actual	Hybla	18	0	0	0	0	100%	100%
	UDP	0	0	6	0	0		0%
	UDPLite	0	0	42	0	0	78%	100%
	SCTP	0	0	0	0	12		0%
	QUIC	0	0	6	0	24	67%	80%
	Weighted Average						**78.7%**	**93.3%**

(b) test dataset

		Predicted						
		Hybla	UDP	UDPLite	SCTP	QUIC	**Precision**	**Recall**
Actual	Hybla	6	0	0	0	0	100%	100%
	UDP	0	0	2	0	0		0%
	UDPLite	0	0	15	0	0	79%	100%
	SCTP	0	0	0	0	4		0%
	QUIC	0	0	2	0	9	69%	82%
	Weighted Average						**79.5%**	**93.8%**

IV. EXPERIMENTS AND SYSTEM EVALUATIONS

The carried experiments' goals were to *(i)* show VTL's ability to effectively replace TCP with another L4 protocol during data transfer, *(ii)* measure the cost in terms of delay of the data redirection operations, and the delay of the dynamic deployment of the VTL's programs, *(iii)* evaluate the VTL's benefits on TCP applications' performance by using decision tree models and *(iv)* assess the precisions and recalls of the trained models used to drive the best L4 protocol selection.

A. Testbed setup and Methodology

The experiments have been performed under a testbed constituted by two hosts linked by one router (Fig. 1). Each host was equipped with Intel Core i7-7500U CPUs, 3.8GiB RAM, and Qualcomm Atheros QCA6174 NIC driver. In addition to TCP and its extension Hybla, we evaluated the following IETF protocols: UDP, UDP-Lite, SCTP, DCCP2, DCCP3, and the QUIC protocol. For each protocol, we implemented a distributed application (one server and one client). The server part can stream several kinds of files with different sizes. The network link parameters are still emulated thanks to `netem` tool [17]. The network parameters used during experimentations are reported in Table I. For each emulated link, the random loss rate is variable between 0 and 5%.

Satellite links emulation. Often used as backup Internet links, satellite Internet is useful for critical missions such as SAR (search and rescue) operations as well as to provide Internet access in rural areas. The main characteristic of satellite links is their long delay that can cause severe performance degradation. Based on [16], we used the following parameters to emulate a satellite link between the client and the server during experiments: RTT to 600 ms, and 4 Mbps of bandwidth.

Terrestrial Internet links emulation. To emulate a classical Internet link between the server and the client, we set the bandwidth to the arbitrary value of 100 Mbps and fix the RTT to 100 ms. To estimate the average RTT value on the classical Internet, we used the WonderNetwork [18] tool to find out the mean RTT between different locations all over the world within the Internet.

Local Network links emulation. The third emulated network profile is a local network (LAN), such as a home network. The RTT is set up to the highest value 50 ms whereas the available bandwidth is 850 Mbps.

Evaluation scenarios and validation approach The experiments were carried out in 2 stages: *(1)* First, we compared performances of the application data transfer under each protocol, i.e., TCP and all other protocols (UDP, UDP-Lite, SCTP, DCCP2, DCCP3, QUIC). In this first step, the application had an API allowing it to directly access each of the protocols evaluated (SCTP API, DCCP API, etc.). This stage allowed us to assess the maximum benefits achievable by using the protocol selected as the most suitable alternative to TCP according to the target application and network contexts and to generate the dataset we used to train the decision tree models. *(2)* Secondly, we repeated the same experiments by comparing TCP with each of the protocols identified by the trained models as the best alternatives to TCP. But this time, the application accesses the service of the selected protocol indirectly thanks to VTL. The application invokes the socket API of TCP, but, thanks to the redirection mechanisms (implemented by the Hooker component of VTL), it will transparently use the services of the selected protocol as an alternative to TCP. With the Wireshark analyzer [19], we validate the correctness of the data redirection by checking the Transport protocol used on the wire during data transfer.

TABLE II
DATA REDIRECTION COST AND *Hooke*R ACTIVATION DELAY.

	Compilation	***Deployment***	***Redirection ops***
`SK_MSG`	0.06 s	0.062 s	11 μs
`SOCK_OPS`	0.09 s	0.064 s	N.A
`XDP`	0.08 s	0.057 s	N.A
Hooker User	0.456 s	N.A	2019 μs
Total	**0.686 s**	**0.183 s**	**2030 μs**

TABLE IV
CONFUSION MATRICES OF THE DECISION TREE *model2*.

(a) training dataset

		Predicted							
		Hybla	UDP	UDPLite	SCTP	DCCP	QUIC	**Precision**	**Recall**
Actual	Hybla	20	0	0	0	0	0	100%	100%
	UDP	0	12	3	0	0	0	67%	80%
	UDPLite	0	3	15	0	3	0	83%	71%
	SCTP	0	0	0	12	0	0	100%	100%
	DCCP	0	0	0	0	15	0	83%	100%
	QUIC	0	3	0	0	0	24	100%	89%
	Weighted Average							**90%**	**89%**

(b) test dataset

		Predicted							
		Hybla	UDP	UDPLite	SCTP	DCCP	QUIC	**Precision**	**Recall**
Actual	Hybla	6	0	0	0	0	0	100%	100%
	UDP	0	4	1	0	0	0	67%	80%
	UDPLite	0	1	5	0	1	0	83%	71%
	SCTP	0	0	0	4	0	0	67%	100%
	DCCP	0	0	0	0	5	0	83%	100%
	QUIC	0	1	0	2	0	6	100%	67%
	Weighted Average							**86%**	**83%**

B. Microbenchmarks

1) Data redirection operations' cost

As reported in Table II, it takes less than one second to activate the *Hooker* component. Further, we could also note that when the *Hooker* is precompiled, its activation delay is reduced to less than 200 ms. Besides the activation delay, once the *Hooker* is activated, its operations namely data redirection introduce additional overheads. We computed these overheads in terms of the average delay required for data redirection operations that are achieved in *hooker_egress* (i.e. `SK_MSG`) and *hooker_userspace* subcomponents. The results are reported in Table II and showed that it takes approximately 2 ms to redirect packets during data transfer.

2) Decision tree models benchmarking

Second, we evaluated the precision and the recall of the trained decision tree models provided in Fig. 6 and Fig. 7. We constructed the models from exactly 146 instances/cases by using an open-source C implementation [20] of the supervised machine learning algorithm C5.0 (described previously in Section 5.2). For the rest of this section, we will call the simplified decision tree illustrated in Fig. 6 *model1* and the extended one shown in Fig. 7 *model2*.

The confusion matrices of *model1* and *model2* are shown in Table III and Table IV, respectively. The reported results show that *model2* achieves more precision (around 10%) than *model1* when it comes to select the appropriate protocol if the pair {*application requirements / network context*} is already encountered. The trend is reversed for the recall's values where on weight-average, model1 presents 93% recall, whereas *model2* achieves 89% recall. As stated previously, the ability to classify correctly already seen cases is not sufficient to assess a model's quality. Its prediction quality, i.e., its ability to classify accurately new and never seen instances, gives more insights. Therefore, we apply the trained models *model1* and *model2* on a *test dataset* containing around forty cases. We observed that *model1* classify almost with the same precision (79.5%) seen as well as unseen cases. The trend is slightly different for *model2*, where the achieved precision (86%) on the unseen instances is not so better as the precision of the classification of seen cases.

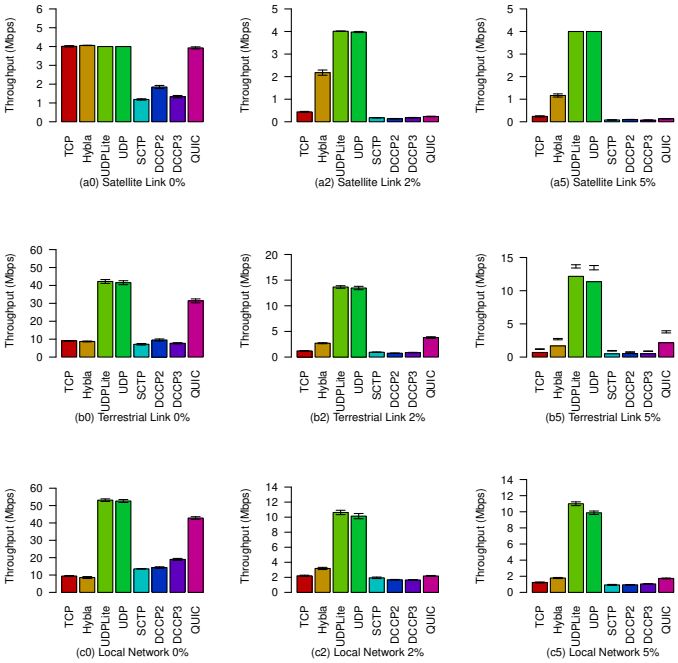
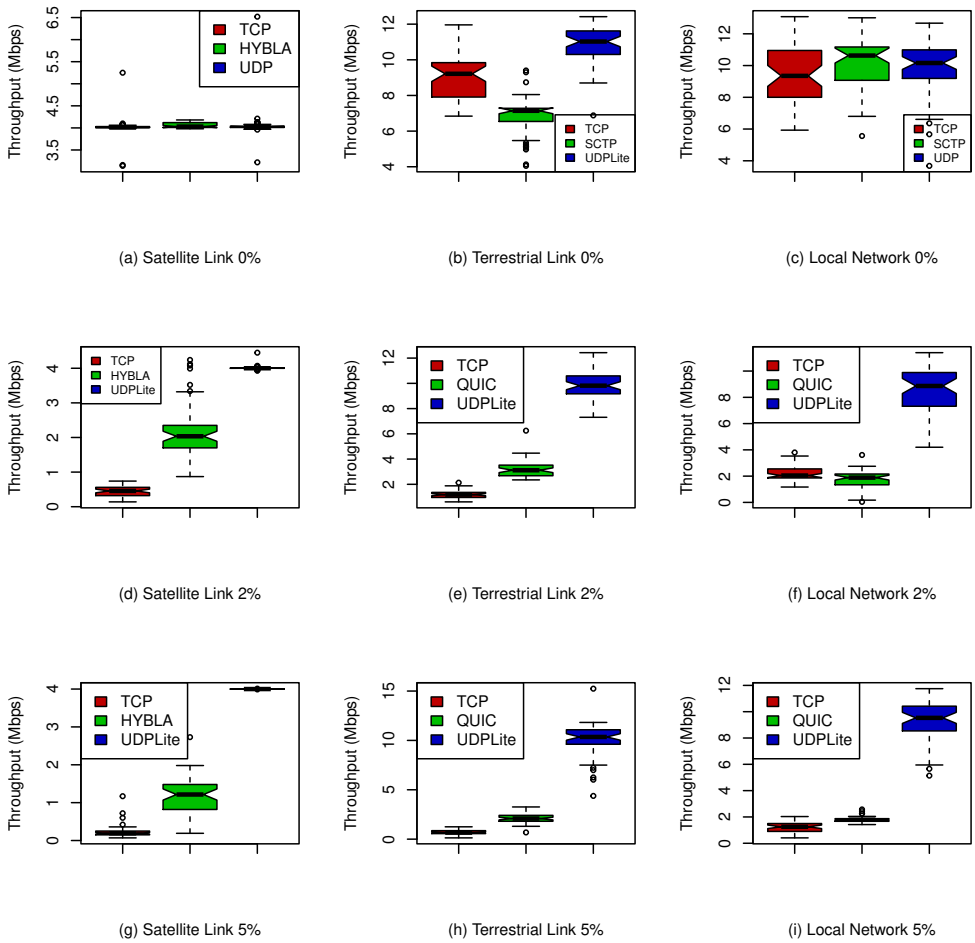


Fig. 8. Application's absolute performance (in terms of throughput) on top of various Transport protocols.

All in all, we note that the simplicity of a model is not necessarily a restriction to its usage. The quality achieved by a simplified model (for instance, *model1* in our work) could be good enough for its use. A model could classify correctly all seen cases but perform worst on new and unseen instances. The trained models *model1* and *model2* are able to make accurate selection of the appropriate protocol 8 times out of 10.

3) Application performances ***Absolute throughput evaluations***. In a first step, we assessed all protocols' absolute performance, i.e., without VTL operations and use of trained models. The results reported in Fig. 8 show the throughput of the evaluated protocols. These results are those used to generate and construct the dataset used to train the decision tree models. Furthermore, they provide us insights into what significant benefits might be achieved by using on the wire another protocol instead of TCP.

TCP applications performance improvement. Then, we evaluated VTL impacts on the performance enhancement of TCP applications. For each considered scenario, we show only the protocols that the decision tree model selects for the considered context. For instance, when the application is loss-tolerant and the network state is {600 ms, 4 Mbps, 0%}, the selected protocol by *model1* to replace TCP is UDP-Lite. In the same network context, when the application is sensitive to data packet losses, the protocol selected by the decision tree *model1* and *model2* is Hybla. Then, Fig. 9 (a) compares the TCP application's performance without redirection and its performance when it is redirected to UDP-Lite or Hybla. The



(a) Satellite Link 0% (b) Terrestrial Link 0% (c) Local Network 0%

(d) Satellite Link 2% (e) Terrestrial Link 2% (f) Local Network 2%

(g) Satellite Link 5% (h) Terrestrial Link 5% (i) Local Network 5%

Fig. 9. Hooked TCP Application's performance (in terms of throughput) under VTL.

evaluations reported from Fig. 9 (b) to Fig. 9 (i) follow the same logic in order to alleviate the figures. The results show that VTL allows TCP applications to achieve at average ˜5x better performances in most scenarios.

V. Related Work And Discussion

A. Related work

Several works have aimed to stimulate the use of Transport protocols other than TCP. In order to gradually enable the deployment and the use of SCTP on the Internet, the authors of [21] and [22] propose TCP-SCTP mapping system for transparent redirection of TCP connections to SCTP. In [21], the mapping tool acts like a transparent proxy called CMG (Connection Manager Gateway) that merges multiple TCP connections into a single SCTP association whereas in [22] the mapping tool is a shim layer designed to be directly integrated into the end-system OS. Similar to CMG and Shim Layer, MiMBox [23] is a protocol converter that ensures the translation between the regular TCP and its multipath extension i.e. MPTCP.

However, the above-mentioned solutions present two main drawbacks. First, they address only the adoption issues of only one Transport protocol: for instance, SCTP in the case of CMG and Shim Layer mapping tools, and MPTCP in the case of MiMBox protocol converter. They are what we could call a *one-to-one* protocol translator and therefore do not provide a comprehensive way for mapping TCP to multiple protocols. Second, even if there is no need to alter the application itself, most of those solutions require the change of the socket API thanks either to kernel patches such as done by Shim Layer, or to the preloading technique like in CMG. Further, MiMBox is developed as a Linux kernel module and as such, it inherits the drawbacks associated with kernel modules namely the lack of security and safety of the end-system.

The approach we explored in this contribution allows the invocation, during the execution of the legacy application, of the alternative protocol *X*, without any modification of the application's code. This approach, which we introduced and implemented, leads at the level of the host machines to intercept the system calls related to the socket API (i.e., `connect()`, `sendmsg()`, etc.) to ultimately invoke the protocol *X*. Therefore, the *Hooker* component does not act as a simple proxy insofar as *(i)* TCP is not activated but rather replaced by the protocol *X* and *(ii)* there is ***no*** one-to-one static and permanent mapping of TCP to a single Transport protocol.

As introduced in section I, more recent works [6], [7], [24]

propose to rethink the entire Transport layer architecture in order to delegate to the Transport layer the choice of the protocol to be used; let us recall that this choice is currently let to the application developer. In [24], the authors assert that the main cause explaining the lack of new Transport protocols deployment and adoption comes from architectural limitations of the Transport layer, hence their proposal for a new architecture. Although this approach is promising, it requires rewriting existing applications and as we previously mentioned, this can slow down its adoption.

B. Perspectives

Although conceptually robust, eBPF technology presents some limitations related to the current implementation choices of some of its components, notably `SOCKMAP`. In the implementation of the *Hooker* component of VTL, these limitations have led us to not being able to bypass TCP socket calls without "going back" from the kernel to the user-space. At the cost of an implementation effort (and potentially higher complexity), we could initially consider replacing `SOCKMAP` with a `DATAMAP` which would allow *Hooker* to share data with application directly in the kernel without the need to open and manage additional sockets from the user-space. A contribution to the eBPF community (more broadly to Linux) to address this limitation is a possible technical area of future work.

During our work, the machine learning models used to select the most appropriate Transport protocol have been trained *offline* beforehand of the deployment of the VTL system. A future direction could be to enhance this approach with *online* learning. That is to say, VTL should be able to learn and update alone the initially trained models. This will permit to limit the risk of inaccurate models when the network environments radically changed or integrated new characteristics not considered in the initial training.

VI. Conclusion

In this paper, we presented and discussed two main contributions. First, we design and implemented a technique that enables the replacement of TCP with another L4 protocol during data transfer. We performed TCP's replacement *transparently* for legacy applications, i.e., there is no need to modify these applications. We believe that fulfilling this requirement is a key factor in promoting the use of Transport protocols other than TCP. We performed extensive evaluations to show the effectiveness of the proposed solution, i.e., its ability to replace at *runtime* TCP by an alternative L4 protocol without any modification of the legacy application. We also assessed the proposed solution impact on TCP application's performance. Further, the results showed that the most appropriate alternative protocol to TCP varies depending on the network conditions and the legacy application's requirements. Even worse, the selected alternative protocol's performance could be worse than TCP's one if the alternative is chosen blindly.

Therefore, we described our second contribution that addresses the problem of the selection of the "best" Transport protocol to use in replacement to TCP based on the application's requirements and the network conditions. To perceive this latter information, we proposed and implemented *dynamic* identification algorithms of the TCP applications' needs and the underlying network characteristics. We used a set of machine learning models, namely decision trees, that we trained to guide the best protocol selection. Finally, we carried out thorough assessments of our proposed algorithms and models. The evaluations showed that leveraging the trained models feeding its knowledge base, we accurately select the most appropriate Transport protocol for diverse application profiles on different network conditions. The outcome is the improvement of the hooked TCP applications' performance.

References

[1] E. Dubois *et al.*, "Enhancing tcp based communications in mobile satellite scenarios: Tcp peps issues and solutions," in *2010 5th advanced satellite multimedia systems conference and the 11th signal processing for space communications workshop*. IEEE, 2010, pp. 476–483.

[2] E. Kohler, M. Handley, S. Floyd, and J. Padhye, "Datagram congestion control protocol (dccp)," 2006.

[3] R. Stewart *et al.*, "Stream control transmission protocol," 2007.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM 2010 conference*, 2010, pp. 63–74.

[5] A. Langley *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the SIGCOMM Conference*, 2017.

[6] T. Pauly *et al.*, "An architecture for transport services," *Internet-Draft draft-ietf-taps-arch-00, IETF*, 2018.

[7] N. Khademi *et al.*, "Neat: a platform-and protocol-independent internet transport api," *IEEE Communications Magazine*, vol. 55, no. 6, 2017.

[8] D. Murray *et al.*, "An analysis of changing enterprise network traffic characteristics," in *2017 23rd APCC*. IEEE, 2017, pp. 1–6.

[9] C. Caini and R. Firrincieli, "Tcp hybla: a tcp enhancement for heterogeneous networks," *International journal of satellite communications and networking*, vol. 22, no. 5, pp. 547–566, 2004.

[10] D. Coffield and D. Shepherd, "Tutorial guide to unix sockets for network communications," *Computer Communications*, 1987.

[11] M. Fleming, "A thorough introduction to ebpf," *Linux Weekly News*.

[12] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[13] T. Heo, "Control group v2. 2015, https://www.kernel.org/doc," *Documentation/cgroup-v2.txt*.

[14] I. Union, "Itu-t g. 1010: End-user multimedia qos categories," *G SERIES: Transmission Systems and Media, Digital System and Networks-Multimedia Quality of Service and Performance Generic and User-Related Aspects*, 2001.

[15] L. Deri *et al.*, "ndpi: Open-source high-speed deep packet inspection," in *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2014, pp. 617–622.

[16] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 329–342.

[17] S. Hemminger *et al.*, "Network emulation with netem," in *Linux conf au*, 2005, pp. 18–23.

[18] "Wondernetwork," https://bit.ly/33TWfVf, accessed 2020-10-11.

[19] L. Chappell, *Wireshark network analysis*. Podbooks. com, Llc, 2012.

[20] J. R. Quinlan, "Data mining tools see5 and c5. 0," *http://www. rulequest. com/see5-info. html*, 2004.

[21] M. Welzl, F. Niederbacher, and S. Gjessing, "Beneficial transparent deployment of sctp: the missing pieces," in *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011*. IEEE, 2011, pp. 1–5.

[22] R. W. Bickhart, "Transparent tcp-to-sctp translation shim layer," DELAWARE Univ., Tech. Rep., 2005.

[23] G. Detal, C. Paasch, and O. Bonaventure, "Multipath in the middle (box)," in *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*, 2013, pp. 1–6.

[24] M. Oulmahdi, C. Chassot, and N. Van Wambeke, "Transport protocols: limitations, evolution obstacles and solutions for an actual deployment in the internet," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 30, no. 6, pp. 515–535, 2015.