VirtualStack: Flexible Cross-layer Optimization via Network Protocol Virtualization

Jens Heuschkel*, Lin Wang*, Erik Fleckstein*, Michael Ofenloch*,
Marcel Blöcher†, Jon Crowcroft‡, Max Mühlhäuser*
*TK / TU Darmstadt, Germany. Email: {heuschkel, wang, fleckstein, ofenloch, max}@tk.tu-darmstadt.de
†DSP / TU Darmstadt, Germany. Email: bloecher@dsp.tu-darmstadt.de
‡University of Cambridge, UK. Email: jon.crowcroft@cl.cam.ac.uk

***Abstract*—The world is driven by the Internet and there is no doubt about its importance in our daily life. However, the Internet has rarely been upgraded since its advent, although the ISO OSI model has already provided the required flexibility. With innovations being blocked, the Internet is suffering from a high-degree of ossification (e.g., the slow progress of IPv6 update), leading to suboptimal efficiency for emerging applications as well as enlarged maintenance cost.**

In this paper, we present VirtualStack, which aims at bringing back the interchangeability of network layers. VirtualStack is based on the idea of protocol virtualization, where the most suitable protocol stack can be dynamically composed and applied on the fly according to the characteristics of both the application and the physical link. Through a comprehensive study, we show that many existing but not widely deployed protocols outperform the omnipresent TCP under various link technologies and network conditions. This provides the necessary insight for dynamic composition of the network protocol stack. We further evaluate VirtualStack under a typical Internet setting with multiple hops under different conditions. The experimental results confirm the benefits as well as the potential of VirtualStack.

***Index Terms*—Protocol virtualization, network virtualization, software-defined networking, network ossification.**

I. Introduction

The Internet has evolved from the 1970s as e-mail network to an important part of our daily life and business today. Research fields and industry trends like Industry 4.0 and the Internet of Things (IoT) keep reinforcing the importance of a well working world wide network. In particular, IoT will be driving us to more mobile and wireless connections than nowadays. As shown by the Cisco Visual Networking Index [1], mobile traffic grew by 52% in Western Europe in the year 2016 and this ratio is projected to increase further. While we were facing 7EB mobile traffic per month in the year 2016, we will probably have to deal with an amount of 49EB in the year 2021.

Unfortunately, this upcoming challenge has not been tackled in an optimized way, resulting in poor cost effectiveness. This is mainly due to the fact that the current Internet is suffering from a high degree of ossification towards TCP/IP. TCP was invented for reliable communication in wired networks without features for real-time communication and encryption, and it does not outbid wireless capabilities. However, as mentioned before, most Internet connections are not simply through wired networks from the link-layer perspective. Instead, the majority of the connections are based on a mixture of different wired and wireless technologies. Since every technology has its own strengths and weaknesses, it is difficult, if not impossible, to address all the physical- and link-layer specifications using one single high-level protocol. This makes the so believed "one-size-fits-all" solution, i.e., TCP/IP, suboptimal in terms of performance in many situations, where cross-layer optimizations are neither considered nor widely adopted.

To address these issues, we present in this paper the following four major contributions: (1) We discuss the vicious cycle in protocol deployment and its important role in the context of Internet ossification. We show in our argumentation how network protocol virtualization could be the key technology for solving this issue, with which we can more flexibly leverage the idea of interchangeable protocol layer as expected in the OSI model. (2) We present a system called VirtualStack for network protocol virtualization, which enables cross-layer optimizations on a hop-by-hop base. We describe in detail its design and implementation for both end-devices and in-network middleboxes. (3) We measure the performance of a large variety of network protocols together with three most significant physical- and link-layer technologies, i.e., LTE, WiFi, and Ethernet, under different link conditions. We observe that TCP is not the best choice for all considered scenarios and we explore the best performing protocol for each of the scenarios. Based on the measurement results we discuss suitable protocol alternatives considering the feature combinations of link- and transport-layer protocols in harmony with application needs. Furthermore, we identify the most influencing factors on network protocol performance based on our measurement experience. (4) We evaluate the benefit of applying end-to-end and in-network protocol virtualization compared to the usual TCP-based solution. To this end, we present a well understandable real-world scenario and discuss our measurements in its context to support our argumentation. The results clearly show that cross-layer optimizations on a hop-by-hop base at technology boundaries are beneficial.

The rest of the paper is structured as follows: Section II presents the necessary background information along with contribution (1). In Section III we present contribution (2), i.e., our system design of VirtualStack, which realizes network protocol virtualization in currently available systems. Section IV presents and discusses our evaluation results, namely contribution (3) and (4). Section V presents related work and Section VI concludes the paper and presents some ideas for future work.

978-1-5386-4413-3/18/$31.00 ©2018 IEEE

II. Background and Motivation

The OSI model is considered as the primary architectural model for network communications. With the proposed layered architecture, the OSI model specifies particular network functions on each of the layers, aiming at providing adaptable service experience. However, the Internet evolved in a different way, resulting in a high degree of ossification due to the fact that network innovations are rarely adopted.

The situation is caused by multiple factors that complicate each other. First, network protocols are usually implemented as part of the operating system kernel and the kernel-level code has a huge influence on the stability and security of the overall operating system. However, operating systems vendors always stay cautious about taking the risk to introduce new protocols which are not popular. As a result, emerging applications may want to use some application-specific protocol optimized for better performance but the protocol cannot be made widely available. Second, network applications are usually developed without knowing much detail about the physical network and thus, developers lean to have the secure choice of TCP since it is well known as an always-working solution. This leads to the fact that new protocols will rarely be used by applications even if they may be partially available. This dilemma eventually leads to a vicious cycle and there is no easy way out. Finally, ISPs usually rely on middleboxes to achieve high network efficiency, which are unfortunately only optimized for TCP traffic and may block traffic using new protocols [15]. Recently, QUIC [18] was proposed, which bypasses this restrictions by sitting on the application layer based on the widely available UDP protocol. However, QUIC is customized for HTTP and currently it is not generally available for none Google applications.

The secure choice of TCP in application development can also lead to suboptimal network performance in many circumstances. Being agnostic to the physical links, TCP may not always perform well for all different physical channels including LTE, WiFi, or Ethernet due to their specific channel properties. One the one hand, workarounds based on intermediate proxies (e.g., Indirect-TCP [4]) have been proposed conforming to the layer separation principle, but they cannot be generalized to different channels. On the other hand, cross-layer optimizations have been shown to be helpful [24], but they are usually highly customized to certain network conditions and lack the required flexibility due to the coupled network layers.

We argue that the culprit of the above situations is the fact that application developers are responsible for choosing network protocols without having enough knowledge of the deployment environment, leading to the de facto (yet undesirable) tight coupling between the application and the protocol stack following the always-working choice of TCP. Therefore, our goal in this paper is to explore how we can take away this responsibility from application developers while enabling flexible selection of protocols in the network stack.

Our proposal is *network protocol virtualization*. The basic idea is to introduce an intermediate layer between the application and the network interface, on which network protocols can be virtualized and the most appropriate protocol stack can be dynamically composed and adaptively adjusted based on real-time system states. In addition to addressing the aforementioned two problems, the virtualization concept will make significant impacts on the following aspects. (1) Full end-to-end network performance optimizations will become possible in the context of software defined networking (SDN). By enabling the communication between the end-device and the SDN controller through a predefined control protocol [13], we are able to extend network optimization to the dimension of protocol selection, complementing traffic steering, based on a (semi-)global view of the network. (2) Additional network functionality such as encryption can be easily introduced as extra layers below the application layer, which can be made widely available for all the applications. (3) A broad deployment of protocol virtualization in SDN environments will contribute to a higher impact of ISP's incentives. Since ISPs have an integral interest in running their networks as efficient as possible, we shall see more optimized protocols for specific physical network components.

III. System Design

Targeting the aforementioned issues in current network protocol stack, we propose VirtualStack (VS), following the idea of network protocol virtualization. In this section, we present its design in detail.

A. Design Overview

To realize protocol virtualization, an intermediate layer between application and the networking hardware is needed. This layer can be implemented in user- or kernel-space, whereby both choices have their own advantages and disadvantage. VS is designed to operate in user-space and this design choice is two-fold: We want to maximally leverage the broadly deployed kernel-space protocols with performance retained. On the other hand, we want to incorporate cutting-edge technologies for novel protocol concepts to demonstrate the extensibility of the protocol virtualization approach. Naturally, this leads to a modular design, which also enables rapid prototyping of new protocols. Furthermore, implementing in user-space is easier and faster, making it more attractive for academic use.

The overall architecture of VS is depicted in Figure 1. VS consists of three basic modules: analysis, execution, and management, and exposes three interfaces: payload source to the application which provides an abstraction for the available strategies to intercept the traffic of applications, packet sink to the physical network interface which provides an abstraction for the actual networking interfaces, and a management interface to the network controller in case of centralized management such as in SDN. We elaborate them separately in the following.

B. Interfaces

Payload source. The payload source provides abstractions for the available strategies for intercepting the traffic from upper-layer applications. To support legacy applications we design and develop two alternatives for this purpose: (1) utilizing a virtual network interface similar to VPN connections, and (2) using library injection or modified shared libraries (e.g., on a specialized Android system) to change the socket behavior

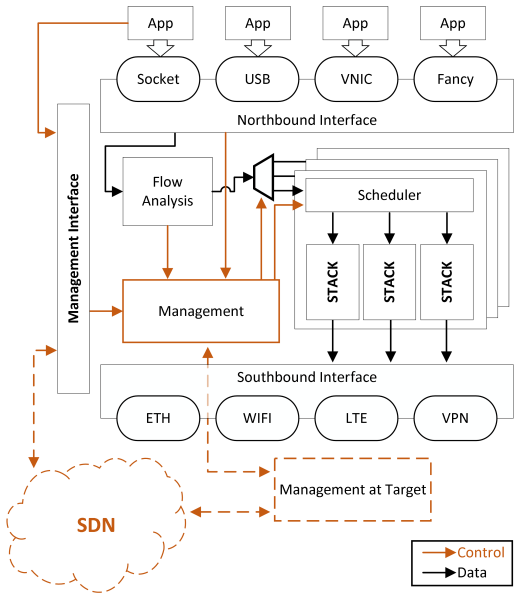


Fig. 1: An overview of VirtualStack's architecture.

to communicate with VS instead of the kernel. For newly developed applications, it's worth to use a state-of-the-art interface over the traditional socket API in order to pass the application needs to VS. Since VS always works on a per packet basis, the provided API to the application has to translate from streams into packets. We will discuss this in Section V.

Packet sink. The packet sink provides abstractions for the actual networking interfaces and it is used by VS to exchange packets with the physical network. In addition, the packet sink provides communication related services. The most important one is coordination with the VS running at the communication partner either by embedding management packets into the stream or by establishing an addition connection. Note that the introduced overhead is rather small, since the management communication is only needed at connection establishments when VS is not controlled by a SDN controller. Other services of the packet sink include gathering of link layer information and automatic hole punching through firewalls as needed for connections between two host running VS.

Management interface. VS features a management interface that is used to communicate with an external controller, e.g., when running VS in SDN environments. This management interface enables the installation of rules for choosing appropriate network protocols in a given environment by an external controller. Besides, the management interface provides functionality for state monitoring and sharing to the controller for well informed optimization. The external controller is able to communicate with VS through the management interface via a well defined control protocol [13].

C. System Modules

Analysis module. The analysis module takes application traffic as input and assigns it to respective network flows. A network flow represents a connection between the applications. Each network flow is defined by a 5-tuple (sending port, receiving port, sending IP address, receiving IP address, type of the originally used transport protocol). The 5-tuple is either passed from the programming interface of the application or directly extracted from the packet headers. Then, the analysis module characterizes the traffic pattern of each network flow by monitoring the time-varying throughput of the flow. This traffic pattern information, along with the 5-tuple, will be provided to the management module (detailed later in this section) for further actions. The packet payload will be passed to the execution module (detailed below).

Execution module. The execution module is in charge of the composition of network protocol stacks. For each network flow, the execution module maintains a scheduler and a set of network protocol stacks for packet processing. One network flow may consist of multiple sub-flows that will be routed on different network paths for better performance (similar to the idea of MPTCP). Each sub-flow will be handled by one of the network protocol stacks. Thus, it is possible to feature potentially different combinations of network protocols on each sub-flow (in contrast to MPTCP where TCP is used for all the sub-flows). The scheduler takes care of the distribution of packets among the activated network stacks for the network flow. A network protocol stack itself is a composition of virtual network protocols on all the layers in the OSI model. Potentially, additional layers can be introduced for more advanced network functions or services. Therefore, the network protocol stack can implement a completely novel academic networking approach, accelerating network innovations. One example is our UDP Plus implementation, presented in Section IV. Upon the start of a network flow from the upper layer, the execution module creates a set of network protocol stacks and establishes the connections according to the used protocols, respectively. After that, packets from the network flow can be scheduled on the created stacks. To change the used network protocol stack, a new stack can be built on the fly and the scheduler can then schedule the packets to the new stack. The execution module is controlled by the management module regarding creation and composing of stacks and packet scheduling.

Management module. The management module controls the operation of VS. Utilizing a low level API, the management module queries information from the modules and the interfaces as described above. In addition to the information gathered internally, the management module also retrieves monitoring information about the infrastructure from the external controller through the management interface. Based on this information, the management module controls the execution module, i.e., the scheduler as well as the network stacks for the present network flows. Decisions are for each network flow which network stacks should be build by the execution module and which of them should be used by the scheduler. Although network protocol stacks can be deactivated when no packets will be scheduled to be transmitted through the stack, the inbound direction for the connection will be always active. Furthermore the management module can influence the scheduling behavior for achieving different optimization goals [9]. In the current design the management module leverages rules to make decisions. To find rules for Pareto optimal behavior of the

management module in different network environments, we have developed an offline machine learning based approach [10]. Another interesting strategy is to categorize traffic flow patterns and create behavior rules to optimize them [12]. Other decision making strategies like online machine learning approaches are beyond the scope of this paper and are left for future work.

D. Critical Processing Path

In terms of throughput, the performance of network stack processing in mobile devices is typically limited by the network technologies, not by the processor power (see CPU overhead measurement in Section IV) , which shifts the focus to the latency caused by VS. To avoid additional latency, we identify the critical path and develop a high-speed and zero-copy processing pipeline. The critical path for network protocol virtualization is the processing path through payload source, analysis module, and execution module including the scheduler and the network protocol stack, and finally packet sink.

It is beneficial to process the critical path in a single thread to avoid the context switching time. On the other hand, this would block the path until the complete packet is processed and copied to the hardware buffer. This blocking is not an issue when just one stack is used, because following packages would be queued in a buffer anyway, since processing is usually faster than the speed the network interface is able to send packets at. However, the blocking behavior becomes an issue if multiple stacks are used at the same time: When one packet is being processed in a stack, the subsequent packets were processed in other stacks concurrently. Furthermore, if one stack is blocked (e.g., by a full buffer) all other stacks were inherently blocked too, since no following packets can be processed.

To this end, we decided to use a three stage pipeline: (1) Analyze the packet to assign it to a network flow and pass it to the corresponding scheduler. In the execution module the scheduler is still part of the first stage. (2) Process the packet in a protocol stack assigned by the scheduler. This includes beside the protocol stack all additional function layers. Every stack has his own thread independently in order to be able to process packets concurrently. (3) Send out the packet to the network interface to finish the packet processing. As the payload source is usually executed as part of the application, it is not included as part of the pipeline. In times with multi-core processors, this single context switch between the first and second stages can be neglected by using optimized memory management and thread scheduling. Between the second and third stages, there is ideally no context switch but a copy action to the hardware accessible memory.

The management module works concurrently to the critical path. The low level API is designed not to interrupt critical path execution. Thus, operational changes are applied after preparation is done.

E. Receiving Packets

For receiving packets, VS has a passive role. Management decisions are always made by the sending node and communicated to the receiving node via one of the various management mechanisms. The receiving VS management module takes care

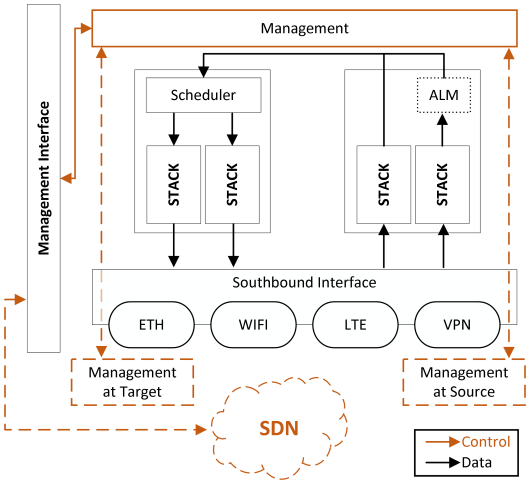


Fig. 2: Architecture of In-Network-VirtualStack.

of fulfilling the required protocol stack configurations. The critical processing path consists only of the execution module, including the network protocol stack and the scheduler.

In VS we consider every connection as a bidirectional connection. If one protocol does not support a bidirectional communication channel, VS automatically builds a stack for a back channel on the receiving side.

F. Operation as In-Network-Process

As motivated in Section II it can be beneficial to change the protocol on technology boundaries (e.g., between wireless and wired links). Therefore we developed a middle-box-version of VS (called In-Network-VirtualStack (INVS)), place-able as in-network process on or near to technology gateways (e.g., on an LTE cellular base station). To change protocols, INVS acts as the intended receiver and builds a respective receiving stack. To forward packets with a new protocol, INVS builds a respective stack and sends packets using that. Applications will still have the transparent transport layer experience, since this action is completely covered by VS itself. As every additional instance of INVS adds overhead, its placement should be restricted to technology boundaries where protocol changes are beneficial.

As illustrated in fig. 2, INVS consists only of the management and the execution module. The analysis module and the packet source interface are not needed since there is no application pushing data into the system. Packets arrive in the respective execution engine and therefore, packets are directly fed into the respective scheduler. As in the end-node VS, the scheduler and optional network functions operate in one thread. Network stacks have their own threads in order to be able to operate concurrently.

Also, the management module is working concurrently in a separate thread. Like in the original VS, it decides based on rules or direct SDN commands what protocol to choose. Management communication is handled by INVS. Since INVS is opening new connections to the communication target, new management communication channels are created on demand.

G. Prototype Implementation

To proof the feasibility and measure performance and overhead, we implemented a prototype of the protocol virtualization

technique VS. The implementation was guided by the design presented in III. To ensure reasonable performance we used the programming languages C and C++. To avoid packet payload copies as much as possible, we use a ring buffer located in the kernel space. We do not create a single copy of payload within the critical path of our processing pipeline.

To access kernel-level implemented protocols, we used the well known *socket* command and the *setsockopt* command to change protocol options. As motivated in Section II, we also support user level implementation of protocols and network functions. With the ability to add code as layer within a stack, VS can be used as rapid prototyping platform. Leveraging the layer interface provided by our implementation allows cascading as many layers as desired. Existing protocol implementations can be wrapped to be included as layer.

IV. Evaluation

In this section we discuss our evaluation scenario, measurement environment as well as the results. We focus on the key components of network protocol virtualization. As stated in Section III, network protocol virtualization also features multipath connections as side effect, since we can build multiple network stacks to enable the protocol transitions. In addition, we can use multiple stacks at the same time. However, the actual benefit is mainly defined by the scheduler strategy, which has been discussed in [9].

A. Scenario

As the client server pattern is one of the most common connection pattern in the internet we used it for the evaluation. We set it up as following: A mobile client wants to access content or a service from a data center (DC) (e.g., a website, or a REST-API access). This involves different participating networks to fulfill the described scenario: i) The network inside the DC where the server is located. We consider this as an arbitrarily fast wired network. ii) The core network required as delivery network. This is modeled as fast wired network too. We rely here on measurements from related work [1], [5], [6]. iii) The client network is the last hop to the client (e.g., LTE connection or the home ADSL connection). Usually this introduces a big amount of limitations. There is a variety of access technologies, but in our scenario we focus on LTE as the most common one [1]. For the performance parameter we rely on measurements from related work [16], [2]. Figure 3 illustrates the described network. For all of our measurements we use a mouse flow and an elephant flow to represent the traffic pattern. Likewise in [11], we define the mouse flow as a low-throughput short-term flow and the elephant flow as a high-throughput long-term flow.

The considered scenario reveals a big problem of current end-user connections. Most network connections do not consist out of a single physical layer type, but a set of link layer types including wired and wireless links with different properties. Typically, wired network connections are full-duplex and are very reliable. On wireless links we deal with half duplex and high packet loss rates over a shared medium. A major problem is the resulting latency jitter of medium access technologies

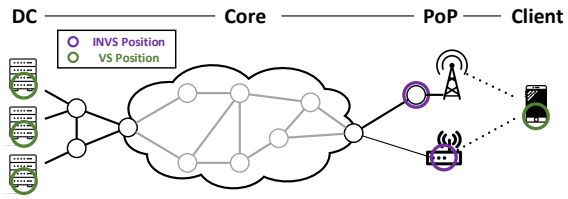


Fig. 3: Full evaluation scenario including the network partitions.

or to get access at all in crowded scenarios. With these contradicting link layer properties, it is impossible to optimize for all their properties at once. Potential optimizations have to work over the whole path, whereby the optimization is a big compromise for all used links and therefore not optimal at all.

We evaluated both wireless and wired connections in different flavors regarding reliability, throughput, and latency. Ideally, every link technology utilizes an optimized transport layer technology to solicit the best performance for the link.

B. Evaluation Environment

For the evaluation we executed VS on bare metal hardware. VS used existing kernel- and user-level implementations of the evaluated protocols. Therefore, we used a Intel Xeon E5-2687W v4 CPU together with Ubuntu 16.04.4 LTS, kernel version 4.4.0-83. To realize the physical connection between the different VS-instances, we used the network simulator NS3 in version 3.27. Leveraging a TAP interface, VS fed the generated network packets into a ghost node inside the simulation. Hence, VS and the network protocols were executed on real hardware, and thus, behaved like in real networks, since the tap device is treated as actual network hardware.

For simulating wired Ethernet and wireless LTE connections we used the built-in models available in NS3. Due to the shared medium LTE enables sending and receiving in slots, which was, unfortunately, not simulated by the NS3 LTE model. Since we expect a big influence of the resulting jitter on the congestion control, we came up with a medium access emulation: We added a node between the sender and receiver nodes. To the sender, the intercalated node was connected with an ultra fast (100Gbps throughput and 0ms delay) CSMA connection, and to the receiver the intercalated node was connected with a point-to-point LTE connection reflecting the respective test parameters. The CSMA connection was capable of forbidding the sending, which was used to simulate the LTE slots. According to the scenario presented in Section IV-A we evaluated the abstracted topologies shown in Figure 4.

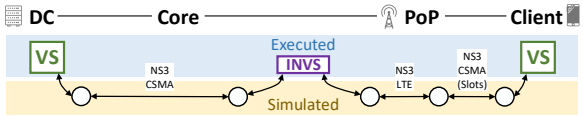
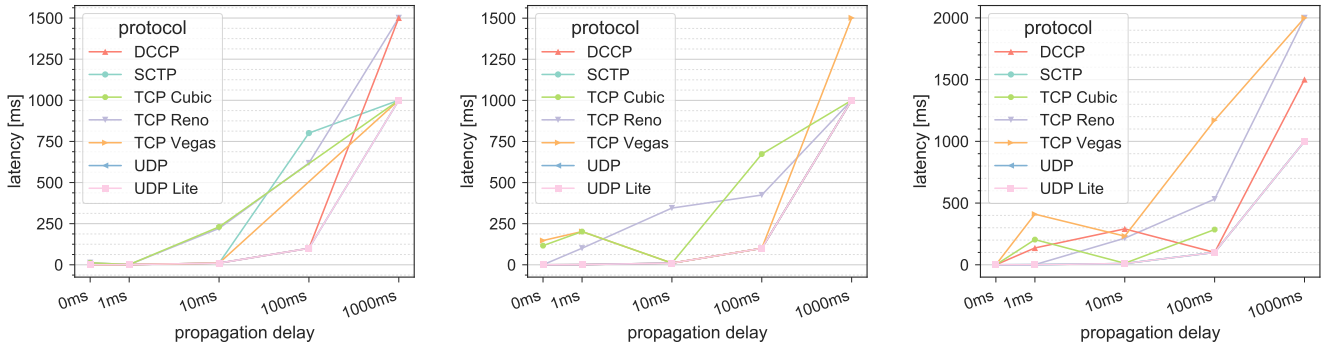


Fig. 4: Abstracted form evaluation scenario.

C. Single Hop Measurements

We start our evaluation by finding the baseline for a variety of protocols (TCP Cubic, TCP Reno, TCP Vegas, UDP, UDP



(a) 0% packet loss (b) 0.1% packet loss (c) 1.0% packet loss

Fig. 5: Latency in relation to propagation delay for different protocols. Measured on a LTE channel with slotted medium access.

Lite, SCTP[1], DCCP[2]) on a set of common link technologies, namely Ethernet, LTE and WiFi. For variable link layer performance parameters we used 1) bandwidth {1, 10, 100, 1000} Mbps, 2) propagation delay {0, 1, 10, 100, 1000} ms, and 3) packet-loss {0, 0.01, 0.1, 1, 10} % . We measured every combination of the network parameters to research how different protocols perform in different network environments and how the parameter influences each other.

In Figure 5 we illustrate the measured latency in relation to propagation delay on an LTE link. The three diagrams show the results for loss rates of 0%, 0.1% and 1%. Surprisingly, the raw link speed does not seem to have any influence on the performance of the transport layer protocol, when the host provides enough computing power for the packet generation. The propagation delay and loss rate have a big influence on the performance of protocols with congestion control. In particular, TCP seems to perform very well in the typical Ethernet configurations till with smaller than 100ms propagation delays and low loss rates ($< 0.1\%$). But with the wireless connections, where the medium access is within slots, TCP seems to have problems to work with higher propagation delays and high loss rates, resulting in a latency of several times of the physical propagation delay. We believe this is a problem with the congestion control algorithm, explaining also the heterogeneous results for the TCP measurements with different congestion control algorithms. We observe similar results in our throughput measurements.

Fortunately, the UDP-based protocols perform very well since they are very lightweight and do not include congestion control. UDP is worth to be use in applications that can deal with light packet loss (e.g., video streaming codecs can deal with packet losses[20]), or on links which can prevent packet losses. However, applications often need a reliable communication channel, which is not provided by UDP.

The above results demonstrate that, depending on the link conditions transport layer protocols perform very differently and there is no silver bullet in all cases. Even the configuration of specific functions like the congestion control has a huge influence on performance, while being rarely used in reality.

[1]https://tools.ietf.org/html/rfc4960
[2]https://tools.ietf.org/html/rfc4340

D. Composed-Connection-Measurements

For the measurement we also composed a typical internet connection as depicted in Figure 4. We show the raw link configuration as the first bar in Figure 6, to present the best possible result. As a baseline we measured the default TCP configuration of the current Linux kernel (TCP Cubic) with underwhelming results. As we already realized in the single hop measurements, TCP Cubic handles slotted medium access not optimal.

Therefore we leverage the concept of additional stack layer for adding network functionality. We implemented a packet ordering layer, a flow control layer and a NACK-based reliability layer and used them together with UDP. This combination is shown as UDP-Plus in our measurements. UDP-Plus fits the needs of single-hop LTE connections perfectly, since it is still very lightweight without congestion control but features the reliability of TCP. Still, it is not optimal for core networks, because on none-exclusive channels congestion control is needed.

Following the baseline, we measured every network protocol of our selection to find out which performs the best. With these results we can configure VS to use the best performing protocol to establish an optimized end-to-end link. In Figure 6 we show TCP Vegas and SCTP as better performing alternatives for the specific link configuration. UDP-Plus can be an alternative for mouse flows. In a real-world deployment this is very hard to realize, since we cannot measure every supported protocol for every network connection. We would need sophisticated heuristics and constantly exchange monitoring data with the ISPs to obtain a best guess. However, it could be a starting point to leverage lightweight protocols for applications, which can handle a certain degree of unreliability [20].

As last case, we used INVS on technology boundaries to select and deploy the optimal protocol for each hop as shown in our single hop measurements. In this case the decision is as easy as consulting a look-up table, since measurement results can be produced for every possible environment beforehand. Thus, this solution is suitable for real-world deployment. The results are convincing although INVS introduces more overhead on delay. As motivated above this could be a good solution

just for the first hop of the clients within the LTE network, since the first hop turns out to have the biggest influence on the overall performance. A desirable configuration would be UDP-Plus for LTE and TCP for the rest of the internet, as we used it in this evaluation.

In terms of throughput, the slowest hop dominates the overall performance of the connection, since it literally builds a bottleneck. Therefore, the optimization comes down to optimize this bottleneck hop. INVS switches the protocol from default TCP to a better performing one on this link, and thus, improving the overall performance. Typically, this is the first hop of the access network. That makes this approach even more attractive, since ISPs are typically in full charge over the LTE connections and often even the phone itself (through customized firmware as part of their contracts).

E. Overhead

Naturally, network protocol virtualization introduces some overhead. We measured the introduced latency by sending packets from a packet generator through VS / INVS to a receiver and taking respective timestamps. Figure 7 shows the time VS needs to process the payload and create a network packet out of it. Obviously, our prototype VS introduces very low latency due to its optimized design as described in Section III. Even the slowest measured protocol TCP takes just around $9\mu s$ to get a packet processed. Also INVS (see Figure 8) has a low latency footprint. Most protocol transformations are processed within $18\mu s$, except our self-built UDP-Plus, which is obviously a problem with our implementation. However, the processing is finished within $80\mu s$ at most. Thus, VS's and INVS's influence on latency is in network dimensions negligible.

The time to open a new connection is dominated by the used protocol. For stateless protocols (e.g., UDP) there is no extra delay. For connection-oriented protocols (e.g., TCP) there is a delay for the respective handshake. Again, processing times are negligible. In cases where VS and INVS are not pre-configured with rules (e.g., by a SDN controller), VS opens management connections to distribute the protocol decisions. This introduces an additional delay of 3 RTTs.

The maximum throughput per network flow is limited by the single threading performance of the CPU. In our test environment we observed a maximum throughput of roughly 260 kpps, which translates with an MTU of 1400 bytes to roughly 2.9 Gbit/s. We observed no influence of the packet size on the throughput performance. Note that these numbers reflect the performance of our prototype, which aims only for academic demonstration and evaluation purposes.

F. Limitations

We are aware that transport layer protocols are not arbitrarily exchangeable. It heavily depends on applications needs. In one case, a robust video stream codec can deal with high losses and prefers better latency over reliability [20]. In another case, certain functions like reliability or in-order delivery are heavily needed. On the other hand, even if an application needs all these functions, it is still worth to optimize all layers. E.g., LTE supports both functions in the link layer, which make them pure processing overhead at the transport layer. Additionally, our measurements show that, function equivalent protocols (like SCTP) perform better in certain cases.

Not to forget, most developers choose the TCP/IP stack as a common practice or for compatibility doubts. There is a big likelihood that many applications do not need all the functions provided by TCP. As motivated above, network protocol virtualization provides the necessary environment to harmonize application needs, network capabilities, and used network protocols.

V. Related Work

There is a huge amount of concepts tackling the internet ossification problem. It would go beyond the scope of this paper to mention them all. For a very good overview we recommend the survey by G. Papastergiou et al. [21]. Here we discuss related work to network protocol virtualization at end- and router-nodes.

In [19], Martins et al. presents ClickOS, which is a lightweight operating system using click modular router [17] to enable network flow manipulation. It suits the needs for applications regarding network function virtualization perfectly well. However, it is neither intended to run on end-devices (e.g., laptops or cellphones) nor designed to work on networking protocols itself like VS.

In the IETF Draft [23], You presents 3RED TAPS. Like VS it wants to provide a decoupling of applications and the network transport layer. Another common goal is to achieve that decoupling without a customization or reimplementation of the applications. Thus, TAPS need a kernel modification to insert their services before the network packets are processed by the kernel. However, VS is intended to decouple the applications from all network matters and not just from the transport layer. Also, the goals of VS include cross layer optimization of application, transport and link layer.

In the last couple years, there were some new approaches to replace the socket interface by a more high level interface, offering a service oriented specification instead of specifying a protocol. The approaches either define a new socket interface to allow the application to label flows [14], to express communication preferences by policies [22], optimize for application objectives, e.g., throughput or delay [7], or provide a resource oriented view to the application [8], [3]. Schmidt et al. [22] and Higgins et al. [14] focus on selecting an interface for the whole data flow. Deng et al. [7] also aim at switching the interface during the data flow if the used transport protocol supports roaming. All these approaches show the benefit of choosing an appropriate interface.

VI. Conclusion and Outlook

In this paper we presented our approach of network protocol virtualization. It decouples applications and network protocol stacks, while enabling cross-layer optimization for network protocol stack. This addresses not only the network environment agnosticism of applications, but also the application agnosticism of network environments and protocols. Based on a real world

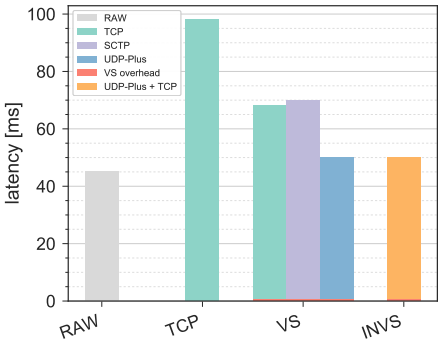


Fig. 6: Results for composed connection.

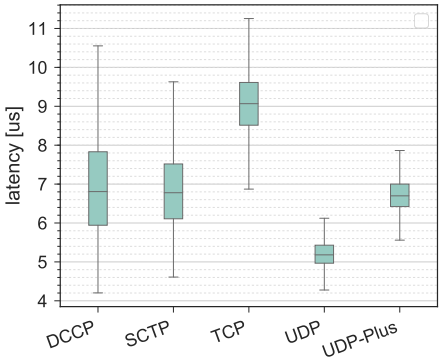


Fig. 7: Critical path latency

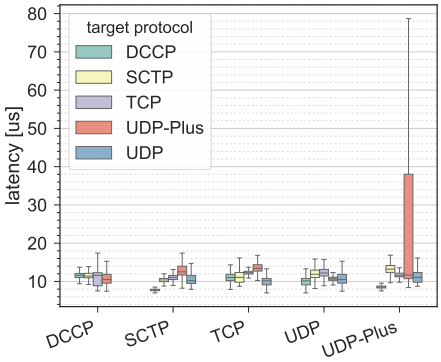


Fig. 8: Protocol transformation latency

scenario we demonstrated how network protocol virtualization can improve the overall network performance for complex network connections build on many different network link technologies. Thereby we showed, depending on application requirements reasonable performance improvements in terms of latency and throughput are possible.

In future, we want to evaluate the presented approach in a real world deployment. We showed the important basic features in the described scenario, but in fact, we see some more application fields in dynamic mobile communication scenarios (e.g., car or IoT communication). Furthermore, we want to go beyond the current performance by utilizing specialized hardware for network protocols and management tasks.

Acknowledgment

This work has been funded by the German Research Foundation (DFG) as part of the projects B2 within the Collaborative Research Center (CRC) 1053 – MAKI.

References

[1] Cisco visual networking index: Global mobile data traffic forecast update, 2016 - 2021. Feb. 2017.

[2] The state of lte (february 2018). *http://opensignal.com/reports-data/global/data-2018-02/report.pdf*, February 2018.

[3] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik, and R. West. A quality-of-service enhanced socket api in gnu/linux. In *Proceedings of the 4th Real-Time Linux Workshop, Boston, Massachusetts*, 2002.

[4] A. V. Bakre and B. R. Badrinath. I-TCP: indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, British Columbia, Canada, May 30 - June 2, 1995*, pages 136–143, 1995.

[5] D. Belson, J. Thompson, M. McKeay, B. Brenner, R. Möller, M. Sintorn, and G. Huston. Akamais state of the internet. *Recuperado de https://www. akamai. com/de/de/multimedia/documents/state-of-the-internet/akamai-state-of-the-internet-report-q2-2014. pdf*, 2014.

[6] M. Chowdhury, R. Agarwal, V. Sekar, and I. Stoica. A longitudinal and cross-dataset study of internet latency and path stability. *Dept. EECS, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-172*, 2014.

[7] S. Deng, A. Sivaraman, and H. Balakrishnan. All Your Network Are Belong to Us: A Transport Framework for Mobile Network Selection. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, 2014.

[8] P. G. S. Florissi, Y. Yemini, and D. Florissi. Qosockets: a new extension to the sockets api for end-to-end application qos management. *Computer Networks*, 35(1):57–76, 2001.

[9] A. Frömmgen, J. Heuschkel, and B. Koldehofe. Multipath tcp scheduling for thin streams: Active probing and one-way delay-awareness. ICC, 2018.

[10] A. Frömmgen, R. Rehner, M. Lehn, and A. Buchmann. Fossa: Using genetic programming to learn eca rules for adaptive networking applications. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on*, pages 197–200. IEEE, 2015.

[11] L. Guo and I. Matta. The war between mice and elephants. Technical report, Boston, MA, USA, 2001.

[12] J. Hchst, L. Baumgrtner, M. Hollick, and B. Freisleben. Unsupervised traffic flow classification using a neural autoencoder. In *2017 IEEE 42nd Conference on Local Computer Networks (LCN)*, pages 523–526, Oct 2017.

[13] J. Heuschkel, M. Stein, L. Wang, and M. Mühlhäuser. Beyond the core: Enabling software-defined control at the network edge. In *2017 International Conference on Networked Systems, NetSys 2017, Göttingen, Germany, March 13-16, 2017*, pages 1–6, 2017.

[14] B. D. Higgins, A. Reda, T. Alperovich, J. Flinn, T. J. Giuli, B. Noble, and D. Watson. Intentional networking: opportunistic exploitation of mobile network diversity. In *Proceedings of the 16th Annual International Conference on Mobile Computing and Networking*, pages 73–84, 2010.

[15] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 181–194. ACM, 2011.

[16] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[18] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. R. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W. Chang, and Z. Shi. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 183–196, 2017.

[19] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473. USENIX Association, 2014.

[20] J. Nightingale, Q. Wang, C. Grecos, and S. Goma. The impact of network impairment on quality of experience (qoe) in h.265/hevc video streaming. *Consumer Electronics, IEEE Transactions on*, 60(2):242–250, May 2014.

[21] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, et al. De-ossifying the internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2017.

[22] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann. Socket intents: leveraging application awareness for multi-access connectivity. In *CoNEXT*, pages 295–300, 2013.

[23] J. You. 3red model for taps. Internet-Draft draft-you-taps-3red-model-00, IETF Secretariat, June 2015. https://tools.ietf.org/html/draft-you-taps-3red-model-00.

[24] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. H. Katz. Detail: reducing the flow completion time tail in datacenter networks. In *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, pages 139–150, 2012.