VirtualStack: Adaptive Multipath Support through Protocol Stack Virtualization

Jens Heuschkel[1], Alexander Frömmgen[2], Jon Crowcroft[3], Max Mühlhäuser[1]
[1]TK / TU Darmstadt, [2]DVS / TU Darmstadt, [3]SRG / University of Cambridge
{heuschkel, max}@tk.tu-darmstadt.de, froemmge@dvs.tu-darmstadt.de, jon.crowcroft@cl.cam.ac.uk

***Abstract*—More and more network devices, such as servers or smartphones, have multiple network interfaces. Today's commonly used communication protocols do not leverage these interfaces to increase bandwidth and reliability using multiple network paths. Recent approaches, such as Multipath TCP (MPTCP), clearly show these advantages. However, adaptation of MPTCP is slow as it requires a modified kernel and faces compatibility issues inside the network. MPTCP is also inflexible in the sense that all paths must use TCP. The challenge is to support multipathing on any operating system, with any legacy application using any transport layer protocol.**

In this paper, we present *VirtualStack*. VirtualStack manages multiple network stacks per application and decides on the best stack on a per-packet basis. This allows to support multipath using any combination of interfaces and protocols for every application. We evaluate the multipath support by comparing VirtualStack against MPTCP using a combination of TCP and UDP connections. Additionally, we show how rules provide flexible programmings abstractions for VirtualStack.

I. Introduction

Multihoming is a well known technique to increase reliability and bandwidth, by connecting a device to multiple networks. Most devices today are physically multihomed, i.e., they are equipped with multiple network interfaces. Mobile devices, for example, provide cellular and WiFi interfaces.

Multipath TCP (MPTCP) increases bandwidth and robustness by using these multiple network interfaces and hence multiple paths in parallel [4], [16]. For this purpose, MPTCP uses multiple *TCP subflows* covering different paths. The advantages of MPTCP are obvious and have been shown several times [15], [14]. Apple's natural language user interface *Siri*, for example, is known to use MPTCP.

Unfortunately, outside of data-centers and some lighthouse applications, MPTCP penetration is lagging due to slow adoption by operating system vendors and network operators. As of today, Windows does not support MPTCP, and the MPTCP Linux implementation is neither part of the default Linux kernel nor of the standard Android. Additionally, Internet Service Providers (ISPs) must support the MPTCP header, i.e., must not modify MPTCP headers sent as TCP option [8]. However, many middleboxes in the internet modify or filter options [2], severely limiting the adoption of MPTCP.

MPTCP only supports TCP subflows. We argue that multihoming should be agnostic of the used underlying network protocol. In many situations, a multihoming UDP connection might be more beneficial, or even combinations of protocols on different paths. This increases the flexibility and allows to avoid common pitfalls which impact performance on some link types (e.g., a VPN link[1]). Thus, the challenge is to support multipath on any operating system, with any legacy application using different transport layer protocols.

In this paper, we present how VirtualStack (VS) [7], a framework for protocol stack virtualization, provides flexible multipath connections with multiple different network protocols. VS manages several independent network stacks – from the physical to the transport layer – per application flow. It decides on the best stack on a per-packet basis. This allows to leverage multiple paths, and even to apply different transport protocols per path. Our evaluations show that the increased flexibility of VS does not harm the throughput compared with MPTCP. For scenarios with different transport protocol stacks per sub-flow, VS provides even more performance as it can leverage properties of the underlying network links. We show that this benefit comes with a reasonable low CPU overhead. Furthermore, we show how rules provide a flexible programming abstractions for the VirtualStack.

The contribution of VirtualStack in this paper is twofold:

- VirtualStack enables transparent multipath connections for any application using standard protocols.
- VirtualStack enables flexible multipath connections in the sense that each path can be treated differently.

The remainder of this paper is organized as follows. First, we present the architecture of VS in Section II. Next, Section III presents our performance measurements of the different protocols. Section IV discusses the related work. Finally, Section V concludes the paper and discusses future work.

II. Architecture

In this Section, we present the modular architecture of VirtualStack (VS) and discuss the responsibilities and capabilities of its modules.

A. General Considerations

In principle, the desired functionality could be implemented in several ways, e.g., as (a) programming framework, (b) shim layer, (c) or kernel module. In the following, we discuss these approaches and argue for a fourth, superior solution.

(a) A programming framework which provides a modified network socket could easily manage the network connection for the application. This implies, however, that applications

[1]http://sites.inka.de/bigred/devel/tcp-tcp.html

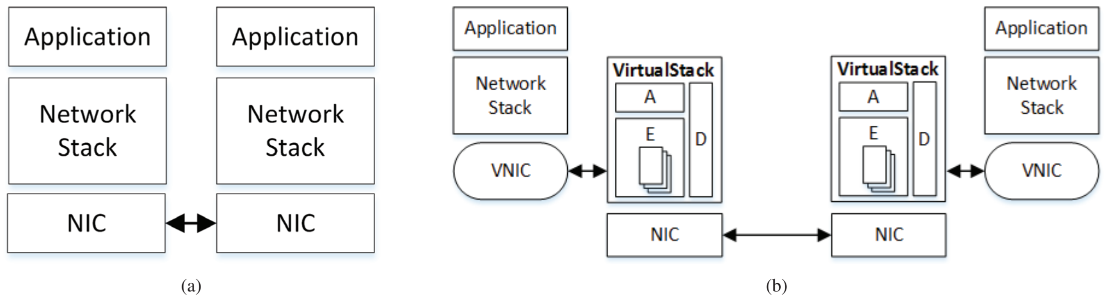


Fig. 1: Illustration of the difference between (a) traditional network operations and (b) the operation with VirtualStack

are prepared to use this framework. Existing legacy applications would not benefit from such a framework. Additionally, developers have to explicitly consider the framework and learn how to use it. Thus, most applications would be written in the same way it is done today without the new functionality.

(b) A shim layer, such as wine [1], is a small library which transparently intercepts API calls. As this implies that the operating systems libraries have to be modified, it is not a optimal solution because operating systems with a closed kernel source wouldn't benefit from the new functionality.

(c) A kernel module suffers from similar problems. Modifying the kernel requires access to the kernel source code. This would work for most Unix/Linux flavors and some BSD systems but not for MAC OSX, iOS or Windows.

In this paper, we tackle these problems and propose a fourth option: Intercepting the network connection with a virtual network interface (VNIC). Using a VNIC – or especially a TUN device – allows us to deploy VS on every operating system that supports VNICs, such as Linux, Windows, Mac OSX, iOS, and Android. Even though we are convinced the proposed VNIC approach is the most suitable, the concepts of VS could inspire the implementation alternatives (a-c).

Figure 1 compares a traditional networking application scenario with VS. Whereas the traditional approach uses a static network stack, VS intercepts the network connection at both hosts. The payload is tunneled through a virtual network interface. VS provides multiple network stacks and adapts the protocol stack for the current network environment. We divide the architecture – illustrated in Figure 2 – in three main modules with different responsibilities: *Analysis* module (Section II-B), *decision* module (Section II-D), and *execution* module (Section II-E). To enable an easy implementation and separate the VS-Core implementation we decide to implement a lightweight interface (Section II-C) to control VS. The decision module is a user-space program which connects to this lightweight interface.

To understand the details of every module, we explain the data-flow – illustrated by the arrows in Figure 2 – inside of VS first. As described above, we use a VNIC (i.e., a TUN device) to get payload (as IP packets) from applications. The received IP packet is processed by the analysis module first. After this step the analysis module passes the packet to the execution module and some meta information to the decision module. The execution module sends the payload over a prepared network connection to its target. Additionally the decision module has the option to send control commands to the execution module.

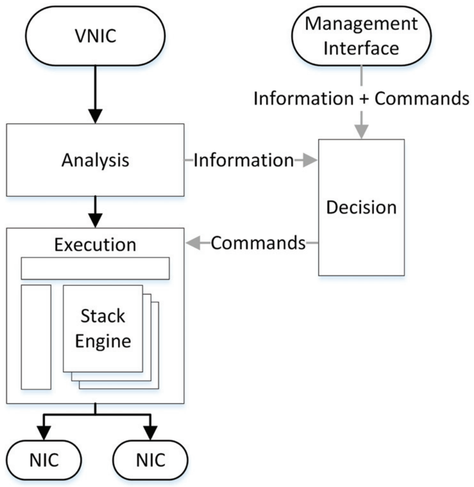


Fig. 2: VirtualStack architecture

B. Analysis Module

The analysis module parses every packet header to identify the corresponding flow. A flow is identified by its *flow id* generated out of three parameters: source port, destination port and destination address. In case the parameters aren't available, the packet is assigned to the flow 0. Every packet from flow 0 is sent over the raw stack (see Section II-E).

Additionally, the analysis module parses the used protocols up to the transport layer (e.g., TCP + IPv4). These meta information are used by the decision module to configure the right endpoint (see Section II-E) and to build an initial stack.

C. Control Protocol

We introduce the control protocol *vs-control* to configure the VS. This protocol decouples the VS from plugable decision modules. The protocol specifies the communication between the decision module and VS in both directions. All registered decision modules are informed about events such as new incoming flows. These events contain additional information, e.g., the *flow id* and the used protocols. The decision module controls VS with the following four basic commands:

- *Build Stack [features] for [FlowId]*: Builds a new network stack with specified features and properties, e.g. a TCP stack with reno congestion control on the LTE link.
- *Set [StackId] for [FlowId] with [quota]*: Sets the specified network stack for the specified flow as active. In case more than one stack is activated, the stacks are used in a weighted round robin manner using the quota.
- *Unset [StackId] for [FlowId]*: Sets the specified network stack for the specified flow as inactive. If every stack is inactive, the default stack – which is the first built stack – is used.
- *Cleanup [FlowId]*: Deletes all stacks to the corresponding flow. After this call, any following packet from this flow is registered as new flow.

These simple but powerful commands allow for plugable decision module which define complex behavior.

D. Decision Module

The decision module is responsible to manage and optimize the flows. Therefore, it requires information about the flows from the analysis module. For every flow it builds a stack engine with an appropriate endpoint and an initial stack. Since the decision module has a management interface, it can communicate with an external optimization instance, e.g., a SDN controller. The optimization instance can send commands or install rules for the best configuration of a connected network path. With these rules the decision engine is capable to build more suitable stacks for a given network environment.

```
agg([flowId].rx.throughput, 5s) < 10Mbps:
  set [TCPoverLTEStack] for [flowId];
```

Listing 1: Example rule which activates an additional LTE flow in case of low WiFi throughput.

We envision decision modules which provide expressive abstractions for complex adaptive behavior. *Event Condition Action* rules for adaptive distributed systems [5], for example, could support the specification of complex adaptation logic. These rules provide concepts to express events and conditions which trigger changes of the protocol stacks. Supposing, for example, an application requires a certain throughput, the rule as shown in Listing 1 turns on LTE in case the WiFi connection is not sufficient. The rules can be evaluation efficiently at the local network device in software and use additional monitoring data and events.

E. Execution Module

The main part of VS is the execution module, illustrated in Figure 3. The module consists of a *flow manager*, multiple *stack engines* and a *raw stack*.

The flow manager assigns every incoming packet to the respective stack engine based on the flow id. If a new flow is registered, the flow manager sets up a new stack engine with the protocols used by the clients. The stack engine is the main component to fulfill the protocol virtualization. It contains the *endpoint* for the application, a *NAT engine*, a *stack manager*, and the corresponding *stacks* of this flow.

After passing the analysis module, every application flow managed by VS is terminated at the associated endpoint. The endpoint acts like a transparent protocol proxy, i.e., it handles the connection to the application and passes the payload to the stack manager. In the case of UDP, for example, the endpoint just sets a pointer to the payload of the packet. TCP is more complex since the endpoint has to handle connection tasks such as handshakes, acknowledgments and retransmissions.

For every new stack, the target server address is given by the *NAT engine*. Typically this is the target address which is parsed by the analysis module. For the transparent redirection of flows, e.g., after a link failure or for load balancing, the target address can be changed.

The *stack manager* is responsible for the actual stacks, that means it builds and uses stacks to send the payload. When a command to build a new stack is received, the stack manager takes the destination address from the NAT engine. The stack manager treats any stack as abstract container, which leads to some degree of freedom. One option is to take a classical protocol combination and utilize the kernel implementations for that. VS provides the additional flexibility to implement new protocols as user-space stack.

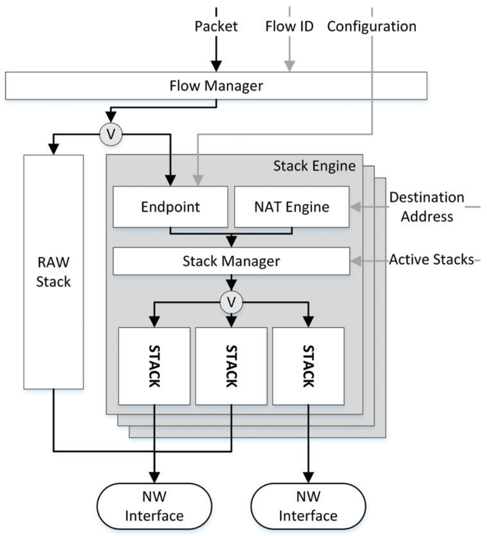
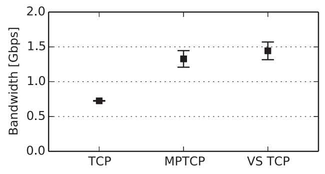
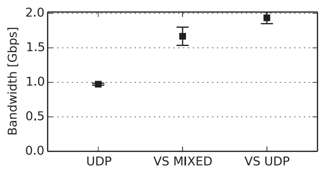


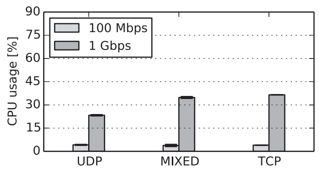
Fig. 3: VirtualStack Execution Module



(a) TCP comparison: Throughput for TCP stacks on two 1 Gbps links.



(b) TCP vs UDP: VS throughput in comparison to traditional protocols on two 1 Gbps links.



(c) Overhead: CPU utilization of VS for two parallel links depending on the link speed.

Fig. 4: Performance evaluation results for VS in comparison to different protocols

As described in Section II-D, multiple stacks can be activate within one stack engine. This is useful for multihoming since the decision engine can activate stacks which are bind to different physical interfaces.

The *raw stack* is a special stack engine for unsupported protocols. Incoming packets that cannot be parsed by the analysis module are mapped to *flow 0*, that is assigned to the raw stack. As packets are sent over the raw stack without any changes, VS supports arbitrary applications.

III. Evaluation

We implemented the VirtualStack (VS) architecture outlined in the previous section and evaluated it with two different setups. First a bare metal setup for performance measurements on physical hardware without any vitualized network parts and second a Mininet [6] setup where we have a controllable environment for demonstrating the operation with ECA rules. Finally, we discuss limitations.

A. General Considerations

VS is implemented as a user-space software in C++. From a performance perspective, it adds another layer of processing to the network interface. Therefore, it is important to mitigate the processing impact as much as possible. VS's implementation is efficiency optimized regarding the CPU, e.g., reduces waiting times for memory operations. Hence, network packets are not copied inside VS. The network packets are read from the TUN device into a kernel buffer. VS then relies on pointers to the respective part of the buffer for any further processing. Therefore, there are only two copy operations of the data in total. First, the copy generated in the kernel-space as the packet is generated. Second, the copy from the kernel-space to the sending buffer of the network device.

For our performance measurement evaluation, we used two machines – a server and a client – each with two 1 Gbps Ethernet network interfaces. They are connected through two physically separated path with CAT6 Ethernet cables. The CPU used for the CPU usage measurements was a mid range commodity CPU (Intel Core i5-4690k) with 3.5 GHz.

For the Mininet evaluation we used one machine with two virtual hosts connected through two separated path, one main channel with 50 Mbit/s (WiFi speed for IEEE 802.11g) and one offloading channel with 20 Mbit/s (typical LTE speed [9]). The operating system used was Ubuntu 14.04 LTS x64 with a 3.13.0-45-generic kernel for both setups.

To generate the workload, we used a Python-based packet generator and counted the transferred packets. The packet generator creates payloads for packets and sends them through a TCP or a UDP socket. The packet counter counts the number of received packets and calculates the throughput. We verified our results with IPerf [17] measurements for TCP, UDP, and MPTCP. Since the server side implementation of VS – to back-transform the used protocols – is still ongoing we couldn't use IPerf for the measurements with VS.

B. Performance Measurements

We start with building a baseline through measuring the performance of the traditional protocols UDP, TCP, and MPTCP. Since VS is a framework which enables automatic network management and optimization we focus on the standard configurations of the single protocols. This reflects the performance when the application does not tweak the network protocol to optimize the connection (which is in fact the most common situation). The maximum aggregated performance of a multipath connection is limited by the combined performance of the single connections. Our measurements show that VS performance is even closer to this maximum as MPTCP. Further, we show that it is beneficial for performance to use lighter protocols like UDP on reliable links.

To demonstrate its capabilities, we used three different modes: (i) VS TCP, which uses TCP on both channels, (ii) VS UDP, which uses UDP on both channels, and (iii) VS MIXED, which uses an UDP and a TCP channel. We discuss our results in three groups: (a) a TCP comparison, (b) TCP vs UDP and mode comparison, and (c) the induced overhead.

(a) To show that VS is competitive to established methods, we measured TCP and MPTCP as baseline and compared it to VS. Figure 4a illustrates our measurement results regarding the considered TCP flavors. As expected, a single TCP connection has the lowest throughput with 725 Mbps. With a variability ($throughput_{max} - throughput_{min}$) of 1 Mbps, it is the most stable link in our measurements. MPTCP enables an average throughput of 1327 Mbps, which is quite near the assumed optimum of 1450 (two times the TCP connection). The variability over was 119 Mbps. VS in (i) TCP mode creates two traditional TCP connections and distributes the packet load over these two. It enables a throughput of 1433

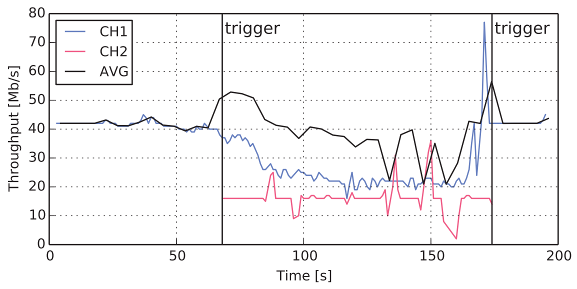


Fig. 5: Throughput for an offloading scenario: main channel (Ch1), offloading channel (CH2), combined (AVG).

Mbps, which is very close to the theoretical optimum and slightly better than MPTCP but has also a slightly higher variability of 127 Mbps. In fact, the measurements show that VS is competitive to MPTCP in terms of throughput, but brings a number of additional features and more flexibility with it. However, VS is able to use MPTCP as transport protocol but opens the possibility to use different protocols on other channels.

(b) As a second common protocol, which is fairly lightweight and therefore promises a higher throughput, we measured UDP (Figure 4b). With regard to theoretical optimum bandwidth, UDP has better performance than TCP since it is a much lighter protocol without the overhead of acknowledgments, retransmissions or congestion control. This restricts the use cases to reliable links or applications which are able to cope with the weaknesses of UDP (e.g., video streaming [13]). Thus, the measured throughput of 973 Mbps is near to the theoretical optimum bandwidth of the link. With VS in (ii) UDP mode, we reach a throughput of 1934 Mbps, which this is nearly the sum of two UDP channels. The variability of 84 Mbps is slightly lower than in TCP mode, which is owed to the missing congestion control.

We further measured this scenario in MIXED mode with different protocols on each of the two channels. This feature is useful, if the connectivity of the device is heterogeneous, e.g., a laptop with an reliable VPN link build on TCP and a unreliable WiFi link. VS in MIXED mode enables a throughput of 1667 Mpbps with a variability of 132 Mbps (Figure 4b). Theoretically, the combination of a TCP and a UDP channel enables 1688 Mbps, which is again just a little more than VS delivers. The shown protocols are the most common protocols on the Internet but it would be possible to use the whole range of available protocols. E.g., in data-centers, where reliable links are available, instead of UDP, DCCP could be used if a congestion control is required.

(c) The higher flexibility comes with a cost. As we described above, we do not copy packets inside VS to reduce any unnecessary waiting time for memory operations and to enable efficient CPU utilization. In Figure 4c, the CPU usage is illustrated. For the used 1 Gbps link, VS needs between 22% and 37% on a single core of our test machine. As we limit the network speed of both interfaces to 100 Mbps, VS uses around 4% of one CPU core. This indicates that VS scales very well, since a 10 times faster link leads just to around 5-9 times higher CPU usage. In fact, this is not much overhead over the TCP encumbered CPU usage. A generally accepted rule of thumb [3] is that 1 Hertz of CPU processing is required to send or receive 1 bit/s of TCP/IP, which would lead to 29% of CPU usage with a 1 Gbpslink. We also observe that the CPU usage does not increase with more flows. That's because the load comes from the number of processed packets, which is limited by the available bandwidth.

We want to create a solution which works, in principle, with every operating system. For this reason, we refrained to use TCP Offload Engine (TOE), which potentially enables significant lower CPU usage [10]. We didn't re-implement all known improvements and techniques from MPTCP. As congestion control we rely on the techniques from TCP on every single path. With UDP we have no congestion control at all. Also the packets need to be reordered on the server side and the round robin scheduling is not convincing on unequal links. In Section V we present some ideas how to solve this.

Our measurements show that multiple paths on devices with multiple network interfaces enable higher throughput. We further demonstrated that the increased flexibility of VS does not harm the performance. VS delivers a wider range of applications since VS is able to freely use available protocols for every network layer up to transport layer.

C. Rule-based Programming

To demonstrate the ease of the rule-based programming abstraction, we executed VS with two simple rules for an offloading scenario (Listing 1). Imagine a live video transfer, which requires at least an average throughput of 30Mbit. With two simple rules, VS can activate a second channel, e.g., a LTE connection, in case the throughput of the primary channel drops suddenly due to congestion. Figure 5 shows an example run in Mininet, where cross traffic leads decreasing throughput on the first channel and therefore triggers the activation of the second channel. A second rule deactivates this channel later, as it detects that the primary channel is sufficient again.

IV. Related Work

In [7], *Heuschkel* et al. presents rough sketch of VirtualStack (VS) and discusses the use-cases (i) protocol transformation, (ii) multipath routing, and (iii) flow migration. The authors evaluated the use-case (i) protocol transformation between UDP and DCCP with a very low switching delay and almost no throughput degradation. The prototype provided a maximal throughput of 4.36 Gbps.

Multipath TCP (MPTCP) is the most related protocol for the discussed features of VS. In [14] and [15], it is shown that MPTCP is capable to improve the performance and robustness of network connections. MPTCP has some limitations as noted previously: the end-to-end path must permit the MPTCP flag, and it is not yet implemented for all operating systems.

Additionally, MPTCP does not provide the flexibility to use different protocols for its subflows.

In [12], *Martins* et al. presents ClickOS, which is a lightweight operating system uses click modular router [11] to enable network flow manipulation. It is also possible to split network flows to multiple path, which could lead to a better performance. It suits perfect the needs for applications regarding network function virtualization. However, it is not intended to run on edge devices, such as mobile devices.

In the IETF Draf [18], *You* presents 3RED TAPS. Like VS it provides 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. For that, TAPS need a kernel modification to insert their services before the network packets are processed in the kernel. However, VS is intended to decouple the applications from all network matters and not just from the transport layer. Also, VS is build to run on every operating system without modifying the kernel.

In contrast to the related work, this paper presented the first approach to use protocol stack virtualization for combining the benefits of multiple paths and the flexibility of a virtual protocol stack which decouples the application and the network.

V. Conclusion and Outlook

In this paper, we presented VirtualStack (VS), a protocol virtualization framework which decouples the application and the network stack. We discussed how this increases the flexibility, and showed on a concrete example how it enables optimizations. We further showed how protocol virtualization enables multipath communication increasing reliability and throughput. We demonstrated the flexibility of the protocol virtualization concept combine different protocols on different network paths transparently for the application. The decoupled architecture of VS enables existing applications to benefit from multiple paths in the network and the flexibility of different protocols without changes.

We implemented VS as user-space program. Our measurements show that VS delivers at least the same throughput as MPTCP. We further presented measurements to show the performance of a multipath UDP network stack and mixed stack with UDP and TCP on the different paths. Thereby, the overhead of VS in terms of RAM is less than 2 MB in all tested scenarios. The CPU usage depends on the achieved throughput and the underlying links. For a 1 Gbps link VS uses just between 22% and 37% (depending on protocol) of CPU power on a single core. On a 100 Mbps link VS uses just about 4% of CPU power on a single core. Note that it is just the utilization of one core and competes with the common rule of thumb for TCP connections.

In this paper, we discussed how the decision engine can implement adaptive behavior to control the usage of multiple paths in the network. In the future, we will connect existing SDN controllers in the network with the control infrastructure of VS to fully decouple the data plane from the control plane not only *inside* the network, but also on the *end-devices*.

Acknowledgment

This work has been funded by the German Research Foundation (DFG) as part of the projects A02 and B02 within the Collaborative Research Center (CRC) 1053 – MAKI.

References

[1] Winehq - run windows applications on linux, bsd, solaris and mac os x. *https://www.winehq.org/*, 2015.

[2] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 1–8. ACM, 2013.

[3] A. P. Foong, T. R. Huff, H. H. Hum, J. P. Patwardhan, and G. J. Regnier. Tcp performance re-visited. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 70–79. IEEE, 2003.

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

[5] A. Frömmgen, R. Rehner, M. Lehn, and A. Buchmann. Fossa: Learning eca rules for adaptive distributed systems. In *International Conference on Autonomic Computing*, 2015. in press.

[6] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[7] J. Heuschkel, I. Schweizer, and M. Mühlhäuser. Virtualstack: A framework for protocol stack virtualization at the edge. In *Proceedings of IEEE Local Computer Networks Conference*, 2015.

[8] 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, 2011.

[9] 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*, MobiSys '12, pages 225–238, New York, NY, USA, 2012. ACM.

[10] K. Kant. Tcp offload performance for front-end servers. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 6, pages 3242–3247 vol.6, Dec 2003.

[11] 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.

[12] 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.

[13] 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.

[14] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. Exploring mobile/wifi handover with multipath tcp. In *Proceedings of the 2012 ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design*, CellNet '12, pages 31–36, New York, NY, USA, 2012. ACM.

[15] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. *SIGCOMM Comput. Commun. Rev.*, 41(4):266–277, Aug. 2011.

[16] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, M. Handley, et al. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *NSDI Vol. 12*, pages 29–29, 2012.

[17] A. Tirumala, L. Cottrell, and T. Dunigan. Measuring end-to-end bandwidth with iperf using web100. In *Web100, Proc. of Passive and Active Measurement Workshop*, page 2003, 2003.

[18] J. You. 3red model for taps. Internet-Draft draft-you-taps-3red-model-00, IETF Secretariat, June 2015. http://www.ietf.org/internet-drafts/draft-you-taps-3red-model-00.txt.