VirtualStack: A Framework for Protocol Stack Virtualization at the Edge

Jens Heuschkel, Immanuel Schweizer, Max Mühlhäuser

Telecooperation, Computer Science, TU Darmstadt, Hochschulstr. 10, D-64289 Darmstadt, Germany

{jens.heuschkel, schweizer, max}@tk.informatik.tu-darmstadt.de

***Abstract*—Applications require connectivity and the network is treated as a black box providing it. Striving to improve this black box, recent research efforts attempt to make the core network more adaptive leveraging new technologies (e.g., software-defined networking). Obviously, any attempt to improve the core network misses out on the first and last hop. Hence, we require solutions that decouple the applications' requirements from the network at the edge.**

In this paper, we present VirtualStack, as a framework addressing this issue. For each individual application, VirtualStack supports multiple optimized network stacks and dynamically chooses the current optimal network stack, possibly taking into account measurements and commands received from the core network. As a proof of concept, our paper demonstrates protocol transformations between UDP and DCCP without any changes to the application, switching delay, or loss in throughput. While the prototype introduces some overhead, it provides a maximal throughput of 4.36 GBit/s.

***Index Terms*—protocol virtualization, application decoupling, network virtualization, software-defined networking**

I. Introduction

Applications today are built on the assumption of connectivity. This increases the strain on the network. One solution to cope with increasing application demands is a move towards more adaptivity in the core network. This is accelerated through trends such as software-defined networking (SDN) and network function virtualization (NFV).

However, these solutions cannot influence the application demands themselves. Unfortunately, applications and operating systems are not adaptive in their request for network resources. Instead, they mostly rely on a limited set of standard protocols (e.g., TCP/IP, and static request patterns). Regardless of the state of the network, an application will request the same protocols for a given connection. Thus, we would like to always adapt the network to best suit the application demands, instead of shaping these demands to the network. Ideally, we mediate between application requirements and the optimal network configuration at the edge node – before the first network hop.

To this end, we introduce VirtualStack (VS), a framework that decouples the application's demands for connectivity from the network stack. VS manages several independent network stacks – from the physical to the transport layer – per application flow. Applications request a connection from the provided virtual network interface (VNIC), enabling transparent network adaptivity without any application changes.

Hidden behind the VNIC are VS's decision and execution engine. These engines will analyze incoming packets, create and manage different network stacks, and decide on the optimal stack per application per packet. VS also provides network address translation per application in case connections have to be re-routed transparently. With these pieces in place, VS enables seamless adaption of the network stack, from the physical interface up to the transport layer. By acting as a virtual network interface, any application developed today can benefit from the adaptive network stack management.

To further motivate VS, we shortly discuss three possible use cases: (i) Protocol transformation, (ii) Multipath routing, and (iii) Flow migration.

(i) Protocol Transformation: In fully managed networks, providers will implement special protocols or cross-layer stacks to achieve better quality-of-service or performance, e.g. link utilization [1]. Across domains, developer default to standard protocols to achieve interoperability at the cost of network performance. VS enables transparent protocol transformation at the edge. Applications can request a connection using standard protocols. However, new protocols can be implemented and applied on-the-fly. This allows for better performance and much faster adaption rates for new protocols. In fact, our evaluation will focus on a simple transformation from UDP to DCCP [2] as a proof-of-concept. Other more elaborated transformations could, add security primitives to connections on the fly for example.

(ii) Multipath Routing: Many devices today feature multiple network interfaces. Laptops are equipped with Ethernet and WiFi. Mobile devices offer cell and WiFi connectivity. Protocols such as Multipath TCP exploit this opportunity by utilizing multiple paths [3]. However, these are inflexible in the sense that each path is a TCP connection. VS can decide on the optimal network interface per packet. Additionally, VS can adapt the remainder of the network stack, e.g., UDP can be used to reduce overhead for the reliable Ethernet connection, while TCP is used for the unreliable WiFi connection.

(iii) Flow Migration: The third use case is motivated by the rise of network function virtualization or in-network processing. In this approach, lightweight virtual machines (VMs) are deployed into the network infrastructure to provide services. With user and VM mobility, connections between applications and VMs get lost. VS is capable of seamlessly re-establishing lost connections and provide address translation capabilities. Hence, from the application's perspective the connection will

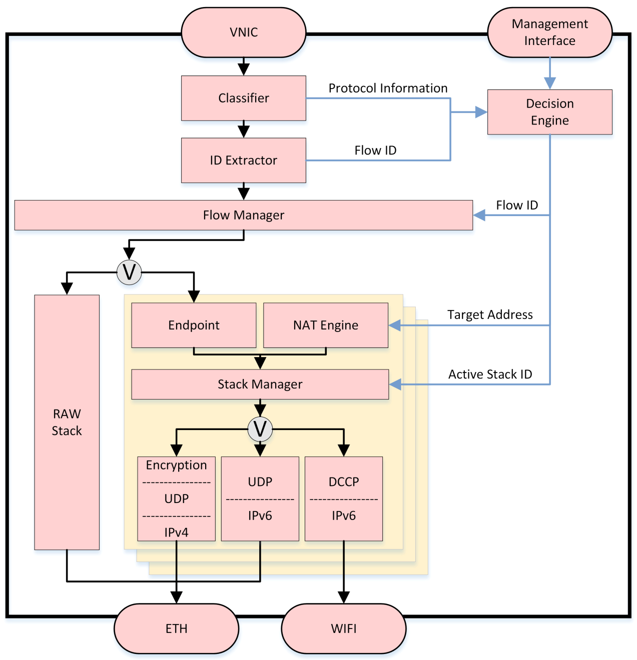

Fig. 2: The VirtualStack Architecture

not be lost, reducing service interruptions.

Our contribution in this paper is VS. VS provides the means to decouple applications from the network. Using VS will increase performance, security, and protocol diversity as motivated in the three use cases. In our evaluation, we focus on one usecase: (i) protocol transformation. We switch between UDP and DCCP during runtime without any reduction in performance, illustrating the feasibility of the VS approach. Even though VS is an early prototype, the overhead is reasonable and we achieve a maximum throughput of 4.36 GBit/s.

The remainder of this paper is organized as follows. First, we will present the detailed architecture of VS in Section II. Next, Section III presents the results with a short discussion on the overhead imposed by VS . Then Section IV introduces the related work. Finally, Section V concludes the paper and presents some ideas for future work.

II. Architecture

VirtualStack (VS) decouples applications and their requirement for connectivity from the instantiated network stack. Applications open a connection to the VNIC provided by VS. Thus, we avoid the need for reprogramming, as applications will send standard IP packets to provide the payload. In the following, we describe the VS architecture and discuss different implementation challenges and how we solve them.

Figure 2 illustrates the architecture of VS with two flows: The data flow of the network packets is illustrated by black arrows on the left side. The corresponding management data flow is shown on the right by blue arrows.

Obviously, the data flow starts with an application initializing a connection. The application will generate a payload and send it as IP packets through the VNIC over a standard socket using the tools of the respective programming language. Then the packets are processed by VS.

VS can be divided into three main parts required to process these incoming packets: (1) Analysis, (2) Decision, and (3) Execution. Each of them consists of several subsystems.

The analysis component (1) will parse incoming packets to extract required information. It is further divided into the *classifier* and the *ID extractor*. The classifier extracts meta-information about the packet and the requested connection, e.g., the protocol or set flags. Identification of the respective flow id is done in the ID extractor.

This information are fed into the decision part (2), where the *decision engine* will decide on the optimal network stack. The decision is based on the information provided by the analysis. It is also based on commands coming in from the *management interface*, where monitoring or control packets from the core networks facilitate better decisions at the edge.

The main part of VS is the execution of the decisions. This component (3) consists of the *flow manager*, the *stack engine*, and the physical network interfaces. The flow manager assigns every incoming packet to the respective stack engine. This is done based on the flow id extracted during analysis. If a new flow is registered, the flow manager will set up a new stack engine. The stack engine is the main component. It contains the application *endpoint*, a *NAT engine*, a *stack manager*, and the corresponding *stacks* of this application.

Every application flow managed by VS will be terminated at the associated endpoint. The endpoint will handle the connection to the application and the payload is passed to the stack manager. The stack manager will send the payload over the respective stacks picked by the decision engine, given the target server address from the NAT engine. If a new protocol is required, a new stack will be built and activated by the decision engine.

Additionally, VS offers the *raw stack*, which is a special stack engine for unsupported protocols. Incoming packets that cannot be parsed will be sent over the raw socket without any changes. Hence, any existing application will work with VS.

VS was implemented as a user-space program in C++. From a performance perspective, it adds another layer of processing to the network interface. Thus, it was important to mitigate the impact as much as possible. To enable high CPU utilization, we tried to reduce any unnecessary waiting time for memory operations. Hence, network packets are not copied inside VS. They are read from the VNIC into an internal buffer. VS then relies on pointers to the respective part of the buffer for any further processing. Therefore, there are only three 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 VNIC and into the VS buffer. Third, the copy from the VS buffer back to the kernel-space to send the packet over the respective network interface.

III. Evaluation

We implemented a prototype of the VirtualStack (VS) architecture outlined in the previous section. We know that

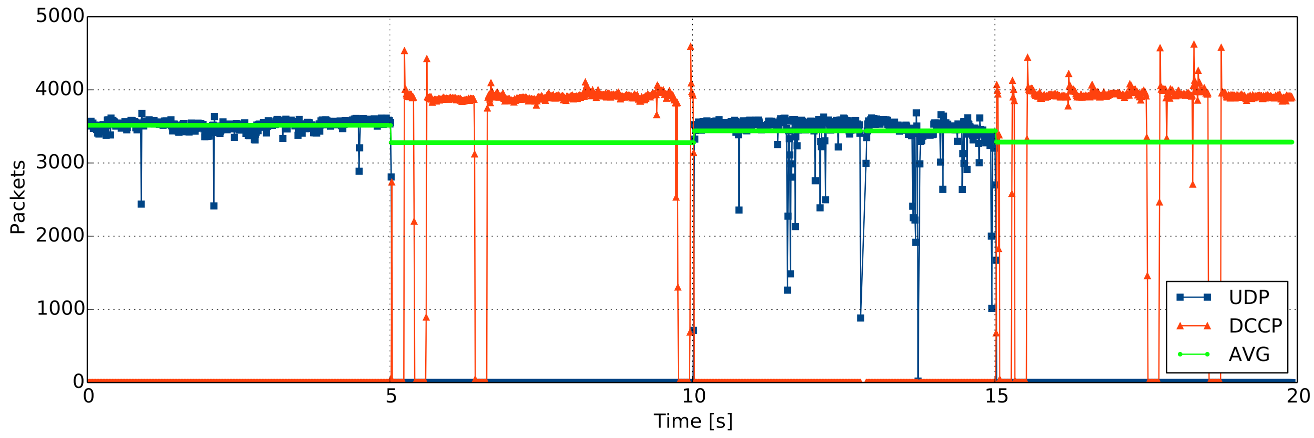

Fig. 1: Number of processed packets per 10 ms for both UDP and DCCP

under certain conditions using DCCP is beneficial compared to UDP (e.g., in congested networks with lots of TCP traffic). Hence, we evaluate a client requesting a UDP connection and VS seamlessly switching between UDP and DCCP connections (cf. use case (i) protocol transformation). To study feasibility, we evaluate the switching delay and performance impact. Since the stacks are build once and then cached, we expect almost no impact on the performance and instant switching time. We also study the overhead of the prototype, by evaluating maximal throughput.

A. Evaluation Environment

For our evaluation, we used a Intel Core i5-4690k 4x3,5 GHz with 16 GB DDR3-1800 RAM. However, VS uses one core with a high utilization of nearly 100% and less than 2 MB of RAM. We used the operating system Ubuntu 14.04 LTS x64 with a 3.13.0-45-generic kernel. Our measurement software and VS are running on bare metal hardware without additional virtualization layers.

In our scenario, we implemented a packet generator application sending packets to two local servers, one for UDP and one for DCCP. A simple python script sends UDP/IPv4 packets with $s = 1500$ byte (20 byte IP header + 8 byte UDP header + 16 byte timestamp string + 1458 byte random string) through the virtual network interface provided by VS. The number of packets c are counted per 10ms, due to the timestamp accuracy in Python on this system.

With this setup, we evaluate the following scenario: VS and the corresponding VNIC are initialized, before the application is started. Any packets sent through the virtual network interface are, by default, sent to the first server using a stack transmitting via UDP/IPv4. Every five seconds, VS will switch the stack. The second stack transmits to the second server over DCCP/IPv4. When the first switch occurs, VS will build the new stack. Afterward, the stack is cached for further use.

B. Results

Figure 1 illustrates the number of received packets at the UDP and the DCCP server per timestamp. It also shows the average performance of the respective UDP or DCCP block. We observe that, with UDP, we constantly receive about 3500 packets per timestamp. With DCCP we get bursts of about 4000 packets per timestamp, with short occasional breaks of about 200 ms. This is probably due to the congestion control. On average the performance of DCCP is almost equal to UDP not considering any packet loss. In summary, we already stated that DCCP can provide benefits compared to UDP especially in congested networks. Applications can now continue to request UDP connections, but VS may provide DCCP, if it is beneficial to the overall network performance.

However, VS adds another software layer to the operating system possibly impacting performance. Fortunately, we observe 3700-3900 packets over more than 3 seconds as peak performance. This amounts to a peak network speed of: $c_{max} * s * 800 = 3900 * 1500 * 800 = 4.36 GBit/s$ On average we get about 3.78 GBit/s, which is excellent for a first prototype and more than enough for most real-world applications.

Figure 1 also shows the real time switch from the UDP stack to the DCCP stack. At the first switch, VS builds the new DCCP stack (at second 5). After this point in time, there is a 200 ms gap needed to conduct the DCCP handshakes and establish the connection. We observe no performance loss, because the stack is built very fast.

VS can now reuse the cached DCCP stack for the second switch (at $15s$). In comparison to the first switching point, we observe no time needed to create a stack and establish a new connection. Hence, switching time is practically zero and we have absolutely no performance drop during the switch. However, after about 50 ms, we observe a drop which is most likely caused by congestion control.

The evaluation shows that it is feasible to perform real time protocol transformation using VS. Also, the change of a stack with an active connection is possible and introduces no additional overhead. Even building an entirely new stack at run time introduces only a one-time time penalty of 200 ms.

IV. Related Work

In [4], Fish et al. present DRoPS, a reconfigurable stack to adapt protocols at runtime. To realize dynamic reconfiguration, protocol stacks are built as a composition of microprotocols, offering simple functions like a checksum up to more complex functionality like TCP. Therefore, the interconnection of these microprotocols is enforced by a configuration, which could either be some initial default configuration or specified by the application through a provided API. Adaptation agents execute policies, to define the information that needs to be monitored by microprotocols. This information then serves as a basis for optimization of the current configuration. However, the presented solution requires close interaction of the application and the actual stack for reconfiguration. For automatic adaptation, Fish et al. sketch a design consisting of a heuristic control mechanism, but do not implement this solution.

The x-Kernel [5] by Hutchinson et al. is an operating system that is optimized to construct and compose network protocols. It provides different protocol objects, which can form a protocol graph. Instances of theses protocol objects are called session objects. They support pushing and popping messages. This offers a convenient way of testing and implementing novel protocol compositions. However, it is not meant to interface with real-world applications.

More recent approaches [6], [7] move essential parts of the communication stack to the user space. While Jeong et al. [7] focus on TCP performance optimizations by limiting system calls or leveraging multi-core functionality, Honda et al. [6] present an approach called MultiStack to support protocol innovation. MultiStack creates dedicated protocol stacks to support MultiStack-aware applications. The authors also executed their stacks in parallel to the operating system stack to support legacy applications. MultiStack itself is built on top of netmap [8], to access the NIC, and VALE [9], a well performing software switch used to separate the single user-space stacks. This solution provides powerful insight in the potential of dedicated protocol stacks. However, the authors require application programmers to change their applications.

Meeting application requirements by selecting the appropriate network interface on multi-access hosts is presented in [10], [11], [12]. The presented approaches either define a new socket interface to allow the application to label flows [11], to express communication preferences by policies [10] or optimize for application objectives, e.g., throughput or delay [12]. Schmidt et al. [10] and Higgins et al. [11] focus on selecting an interface for the whole data flow. Deng et al. [12] also aim at switching the interface during the data flow, if the used transport protocol supports roaming. These approaches show the benefit of choosing an appropriate interface. However, they do not support the adaption of all layers of the stack, limiting the impact.

VirtualStack is the first approach to support adaptation and seamless switching of network protocols up to the transport layer without any changes to the application or the operating system.

V. Conclusion

We present VirtualStack (VS), a powerful framework to decouple applications from the network.

VS receives payload as packets comparable to a socket implementation. By building a specific network stack and binding it to a specific network interface, it enables to choose transport, network, data link, and physical layer. VS provides a virtual network interface to redirect network packets to our system, and with that, being compatible to legacy applications.

Switching between DCCP and UDP in the evaluation showed the feasibility, with no performance impact. Even though VS adds another layer of processing to the network interface, we achieved a maximum throughput of 4.36 GBit/s.

To cope with increasing application requirements, network management and adaptation has to start before the first network hop. VS provides management and optimization at the edge, while also being able to interface with the core network. Future work includes applying VS to all three scenarios discussed in the introduction and has yielded very promising early results. We also work on a rule based decision engine using the Fossa framework[13].

Acknowledgment

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

References

[1] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: a platform for high-performance internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[2] S. Floyd, M. Handley, and E. Kohler, "Datagram congestion control protocol (DCCP)," *RFC4340*, 2006.

[3] 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*, 2012.

[4] R. Fish, J. Graham, and R. J. Loader, "DRoPS: kernel support for runtime adaptable protocols," in *Proceedings of the 24th Euromicro Conference*, 1998, pp. 1029–1036.

[5] N. C. Hutchinson and L. L. Peterson, "The X-Kernel: An Architecture for Implementing Network Protocols," *IEEE Trans. Softw. Eng.*, vol. 17, no. 1, pp. 64–76, 1991.

[6] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo, "Rekindling Network Protocol Innovation with User-level Stacks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 52–58, 2014.

[7] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems," in *NSDI*, 2014.

[8] L. Rizzo, "Netmap: A novel framework for fast packet i/o," in *USENIX Annual Technical Conference*, 2012.

[9] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *ACM CoNEXT*, 2012.

[10] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann, "Socket intents: leveraging application awareness for multi-access connectivity," in *ACM CoNEXT*, 2013, pp. 295–300.

[11] 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 *ACM MobiCom*, 2010, pp. 73–84.

[12] 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.

[13] A. Frömmgen, R. Rehner, M. Lehn, and A. Buchmann, "Fossa: Learning ECA Rules for Adaptive Distributed Systems," in *ICAC*, 2015.