Beyond the Core: Enabling Software-defined Control at the Network Edge

Jens Heuschkel
TK / TU Darmstadt,
Darmstadt, Germany
heuschkel@tk.tu-darmstadt.de

Michael Stein
TK / TU Darmstadt,
Darmstadt, Germany
stein@tk.tu-darmstadt.de

Lin Wang
TK / TU Darmstadt,
Darmstadt, Germany
wang@tk.tu-darmstadt.de

Max Mühlhäuser
TK / TU Darmstadt,
Darmstadt, Germany
max@tk.tu-darmstadt.de

***Abstract*—Software-defined networking (SDN) changes the structure of core networks from statically planned to flexible and adaptable to the network conditions. The de facto standard for the interaction between the controller and the forwarding devices (e.g., SDN switches) is the OpenFlow protocol. With increasing interest in SDN technologies, researchers found solutions to extend these paradigms to end devices, like it is done with the protocol virtualization framework VirtualStack. However, the current SDN protocols such as OpenFlow are unsuitable for controlling end devices, since they are designed for core network nodes. In this paper, we close this gap by presenting TROLL – a protocol suggestion for realizing software-defined control at the network edge. With the communication between the controller and the VirtualStack-enabled end devices being defined by TROLL, control decisions can be made jointly for both the network core and the edge, improving the flexibility, adaptability, and efficiency of the network system significantly.**

I. Introduction

Today, the Internet has become a natural part of the business world as well as our everyday life. A manifestation of this development is the Internet of Things, where humans are surrounded by inter-connected devices. A major concern on the Internet is its flexibility with respect to two aspects: i) Given the diversity of use cases and requirements, a major challenge is the provisioning and deployment of corresponding suitable network protocols. ii) Due to the inherent dynamics of the Internet, another challenge is to switch between network protocols at runtime (i.e., conduct protocol transitions) with respect to changing requirements.

Both these challenges are being addressed explicitly by the research community: The first challenge is tackled by developing novel network protocols on a regular basis to consider upcoming requirements. Regarding the second challenge, another branch of research is concerned with conducting transitions between network protocols.[1] Despite these efforts, we observe an ossification of used network protocols on the Internet: Only a very small proportion of novel network protocols are actually deployed and used in real-world environments. Consequently, the Internet does not perform to its potential.

With the increasing popularity of software-defined networking (SDN), there is a unique opportunity to finally solve the aforementioned challenges. SDN already finds its way into the market of campus, enterprise, and ISP networks. Using a control protocol, SDN allows an independent controller to configure the network in a flexible way at runtime. The de facto standard as control protocol is OpenFlow [4], introduced by McKeown et al. in 2008.

Overcoming the aforementioned challenges requires a fully software-defined network system, which includes control in the network core as well as at the network edge. Unfortunately, to the present day, the new flexibility introduced by SDN is limited to the core of the network. On the one hand, OpenFlow is originally designed only for controlling flow tables at forwarding nodes, thus providing no support for edge devices. Although OpenFlow has been updated multiple times and various alternatives such as Protocol Oblivious Forwarding (POF) [8] and PindSwitch [10] have been proposed, none of them offer commands for controlling edge devices.

On the other hand, Heuschkel et al. [2], [3] have introduced VirtualStack (VS), a protocol virtualization framework for extending protocol flexibility to edge devices. For this purpose, VS decouples applications from their protocol stacks, and offers a management interface to control the composition, build and switch of protocol stacks. Today, VS is controlled locally. To extend protocol flexibility of SDN to edge devices, VS has to be complemented with a control protocol that allows for managing the protocol stack at the edge devices remotely by the SDN controller.

To close this gap, this paper presents Con**trol** Protoco**l** (TROLL), a protocol to control VS at edge devices. TROLL is designed to serve the adaptation possibilities offered by VS. Like OpenFlow, TROLL enables the controller to install rules on the edge devices to react on conditions or events. Additionally, TROLL provides direct commands for reporting, monitoring and giving direct control over the network stacks. To the best of our knowledge, we are among the first to raise the concept of software-defined control at the network edge, enabling joint control decision making for the network core and the edge in a fully software-defined network system.

With all theses pieces in place, the paper provides the following contributions:

- We propose TROLL and design its protocol structure, available commands and the communication behavior.
- We explore possible usage scenarios for TROLL.
- We provide a publicly available prototype implementation of TROLL to show its feasibility and expressiveness.

[1] https://datatracker.ietf.org/wg/taps/documents/

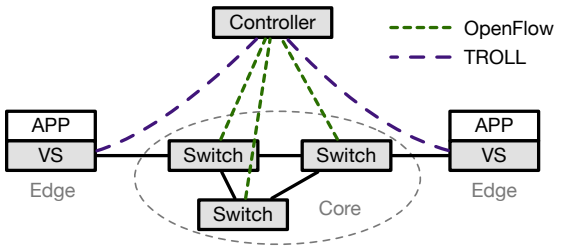

Fig. 1. An overview of a fully software-defined network system where TROLL is envisioned for edge device control, complementing OpenFlow for the network core.

II. TROLL's Design

In this section, we present the protocol structure, the communication behavior and the main design decisions of Con**trol** Protoco**l** (TROLL).

To show the important boundary conditions, we give a quick overview of VirtualStack (VS) [3], which enables SDN control at edge devices. VS is a network protocol stack virtualization framework. It decouples applications from their network stacks, i.e., network stacks are not defined by the application anymore, but built and fully managed by VS. Also, VS allows for switching between network stacks on a per-packet basis. With these two features, building customized network stacks and switching between them, VS enables transitions for every network layer to adapt to the current network environment. To enable SDN control, VS has a management interface, connecting to an SDN controller to receive commands, rules, information or to provide monitoring data.

From our experience with the communication between VS and some controlling instances, we believe it is necessary to provide a generalized protocol for controller $\leftrightarrow$ edge device communication. Therefore, we developed TROLL. Figure 1 shows a typical use case we envision for TROLL: Switches at the core network are typically controlled by OpenFlow. Since OpenFlow is designed for forwarding devices, it is unsuitable to serve the needs of adaptive edge devices. Because VS is currently the only protocol virtualization software for edge devices, we oriented on the feature set VS offers. However, we want to cover the upcoming approaches for adaptive edge device software and contribute our experience with control communication between edge devices and SDN optimization controllers. When we talk about edge networks, we envision use cases where WiFi access points, home routers or cellular base stations are small SDN controllers as part of a distributed controller to manage their sub network. Typically, they will not have a dedicated link just for control commands, like it is solved in SDN core networks. Consequently, we have to use the existing network connection to send commands.

We designed TROLL with four big requirements on the design in mind: i) One of the biggest advantages of SDN is to enable an overall optimization with a global view on the network. To include edge devices into the optimization process, it is necessary to provide tools to influence their behavior. TROLL provides commands to build network stacks, switch between network stacks and influence the scheduling. ii) SDN controllers need information to make optimization decisions. The controllers have some typical monitoring sources to get required information. With TROLL, they can expand their sources to the edge devices by subscribing monitoring channels from VS to get even more monitoring data to achieve a better optimization result. iii) It will not scale if an SDN controller actively has to trigger every adaptation in bigger networks. To relieve the controller, it is necessary to enable some adaptations, e.g., topology adaptations [9], in a self-organized way by the edge devices. This is possible by installing event condition action (ECA) rules [1] trough TROLL. iv) Analogously to the SDN controller, the edge devices need data for their rule based decision making. With TROLL, they can subscribe monitoring channels from the SDN controller to optimize their local behavior to the overall network conditions.

TABLE I
Supported Groups of Control Messages in TROLL

Pattern	Group	Elements
0x0	System	Version, type & packet no.
0x1	Reporting	All
0x2	Monitoring	All
0x3	Actions	All
Other	Reserved	–

According to these usage scenarios, the features of TROLL are structured in four groups (shown in Table I):

A. 0x0 **System** messages are needed for protocol communication (e.g., ACK).

B. 0x1 **Reporting** messages are used to send important control data from VS to the SDN controller. The data are needed to enable correct communication and error free control of VS. Reporting packets never get a response.

C. 0x2 **Monitoring** messages are used to exchange performance data about the network. The SDN controller can use VS as monitoring agent to get data for optimization algorithms. Also, VS can use them to trigger rules based on external events.

D. 0x3 **Actions** are direct commands to control the behavior of VS and install ECA rules, to influence the stack management and the scheduling of VS.

TABLE II
Control Packet Structure in TROLL

Version	Type	Packet No.	Size	Payload
4 bits	16 bits	8 bits	16 bits	up to 65536 bytes

Table II shows the packet structure of TROLL packets. Every packet starts with a 4 bit version field, followed by a 16 bit type field split into 4 bit group according to Table I and 12 bit command. After that, an 8 bit packet number is added. The packet number reflects a continuous packet count; for the case of system response packets, the same number as for the corresponding request packet is used. All fields are interpreted as unsigned integer type. For non system packets, a 16 bit size field and the payload follows. The size field indicates how big the payload is in byte.

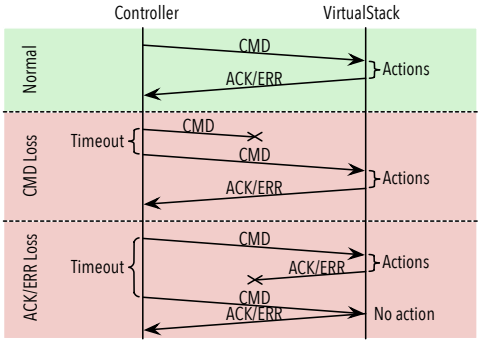

Fig. 2. Error examples of the TROLL communication

As stated above, we cannot rely on a separated physical connection for commands. Especially in wireless links, congestion could occur. To enable a quick adaptation to solve possible network problems, a basic command set is able to be sent within a single UDP packet. This increases the chance for a successful command, since no successful handshake phase is needed. More advanced commands do not fit into a single packet. For such commands, we rely on reliable layers underneath. These design decisions ensure that we can rely on a very slim protocol design and simple communication mechanisms, but still address a wide range of usage scenarios.

Hence, we distinguish between two communication schemes: i) The *low overhead communication*, allowing to send commands within one packet, and ii) the *advanced communication*, allowing to transfer many information pieces.

The low overhead communication is intended to send fast adaptation commands to fix an overloaded network situation. Since it does not use a reliable transport layer, packets could corrupt or get lost, as illustrated in Figure 2. Within this scheme, every command requires a response to indicate if the action is executed or not and whether the packet arrived at the receiver. To detect corruption, the transport layer check sum is an obligation (even in UDP). However, it is also possible to send these commands over a TCP connection.

The advanced communication is intended for organizational communication (e.g., registration), exchanging monitoring data, and installing ECA rules. It should be used on a reliable channel (e.g., with a TCP connection) because most commands do not send a response or do not fit into one packet. Since the command packet is received by VS for sure, we can omit responses when it is not important for the controller to know if the command is executed.

The following commands are defined for TROLL. Packets suitable for the low overhead communication scheme are marked with an asterisk (*).

A. System

The system group includes messages with a type prefix of `0x0`. They are mostly used for protocol communication.

- `0x0001 Ack* ()`
  Acknowledgment for a requested action.
- `0x0002 RuleInstalled* (`*`rule-id`*`)`
  Acknowledgment for a successfully installed rule. The packet contains a 16 bit `rule-id`.
- `0x0010 Error* ()` Unspecific error.
- `0x0011 TransitionError* ()`
  Error during the transition.
- `0x0012 RestrictionError* ()`
  Error because of restrictions.
- `0x0013 NotSupportedError* ()`
  Error because the requested command is not supported.
- `0x0014 UnavailableChannelError* ()`
  Error because the requested channel is not available.
- `0x0015 UnavailableProtocolError* ()`
  Error because the requested protocol is not available.
- `0x0016 RuleInstallationError* ()`
  Error during the installation of the rule.
- `0x0017 RuleNotInstalledError* ()`
  Error because the requested rule is not installed before.
- `0x0020 Warning* ()` Unspecific warning.
- `0x0021 FlowLineRateWarning* ()`
  Warning because the flow line rates exceed the link line rate.

B. Reporting

The reporting group includes messages with a type prefix of `0x1`. They are mostly used for the edge device to report the status to the controller.

- `0x1001 RegisterHost (`*`host-id`*`,`
  `protocols-capabilities``, [`*`public-key`*`])`
  Registers a new edge device at the controller. The `host-id` is a 48 bit node identifier described by RFC 4122[2] v1. The `protocol-capabilities` field contains the available protocols as a list out of 20 bit values. Every value has 4 bits for indicating the OSI layer and 16 bits to indicate the protocol. The protocol number is defined by IANA online database[3] and the public ethertype list[4]. Optionally, it can add a public RSA key for encryption.
- `0x1002 NewLink (host-id, link-id, config)`
  Registers a new link available for VS at the controller. Every link gets a randomly picked 8 bit `link-id`, which is unique in combination with the `host-id`. The config field contains the link configuration like physical type, IP version, and addresses as string key value pairs.
- `0x1003 LinkLost (host-id, link-id)`
  Reports a lost link to the controller. If a link was bound to the specific link, VS assigns it automatically until the controller sends an assignment again. This prevents the flow from dying.
- `0x1004 NewFlow (host-id, flow-id, stack)`
  Registers a new flow with a 64 bit `flow-id` generated from the source port, the destination port, and the destination address. The specific used hash function is up to the edge device, but must be distinct in combination with the

[2]https://tools.ietf.org/html/rfc4122
[3]https://goo.gl/PuaUYn
[4]http://standards-oui.ieee.org/ethertype/eth.txt

host-id. The stack field contains the network protocol stack configuration, specified in the same manner as in the RegisterHost command.

- 0x1005 FlowCompleted (host-id, flow-id, state)
  Reports a completed flow. The state field contains the reason for finishing the flow. Allowed values are zero for normal completion or a string for an error message.
- 0x1006 StackOptions (host-id, options)
  Reports the available options for stacks and specific protocols. The packet is only sent if the edge device supports stack options. The options field contains the available socket options as a list of 8 it unsigned integers.

C. Monitoring

The monitoring group includes messages with a type prefix of 0x2. The commands are used by the controller to proactively monitor the status of the edge device, and by the edge device to monitor global network parameters.

- 0x2001 RequestChannels ()
  Requests the available channels for monitoring at this node.
- 0x2002 ResponseChannels (channel-list)
  Respons for RequestChannels. The channel-list is a string representing a list of the available channel names separated by commas. A channel name is a string structured as ⟨L|F⟩:⟨link-id|flow-id⟩:⟨parameter⟩. The channel provides one parameter, which is link or flow specific.
- 0x2003 SubscribeChannel (channel-name)
  Subscribes a channel for monitoring data. If the channel is available, VS sends the last known value and sends an update when a new value becomes available. Otherwise, VS sends an UnavailableChannelError as response.
- 0x2004 UnsubscribeChannel (channel-name)
  Unsubscribes a monitoring channel. There is no response.
- 0x2005 DataUpdate (type, value, channel-name)
  Initiates a data update for the specific channel. The 4 bit type field defines the type of the value according to Table III, whereby strings are '\0' terminated. The value field contains the new value for the channel.

TABLE III
SUPPORTED DATA TYPES FOR MONITORING DATA

Pattern	Data Type	Size
0x0	boolean	1 bit
0x1	char	8 bits
0x2	uint64	64 bits
0x3	int64	64 bits
0x4	float	64 bits
0x5	string	char[]
Other	Reserved	–

D. Actions

This group of features includes messages with a type prefix of 0x3, which define the actions that can be carried out by the controller to the edge device.

- 0x3001 SetProtocol* (flow-id, stack)
  Forces VS to change the protocol stack for a specific flow. The stack field specifies the requested protocols as a list out of 20 bit values. Every value has 4 bits for indicating the OSI layer and 16 bits to indicate the protocol. The protocol number is defined by IANA online database and the public ethertype list as we have already discussed in command RegisterHost. Layers not specified are automatically chosen by VS. This command gets an ACK response when it is successfully done, while it gets a TransitionError or RestrictionError if something went wrong. If the requested protocol is not supported, VS responds with an UnavailableProtocolError.
- 0x3002 SetFlowLineRate* (flow-id, line-rate)
  Defines the available line rate for a specified flow. The line-rate is encoded in $x = 16$ bit number and $y = 8$ bit exponent of 10; resulting in $x * 10^y$. If the sum of all flow line rates exceeds the available line rate of the link, it is up to VS to manage the flow line rates. Additionally, VS will answer with a FlowLineRateWarning packet. To switch to automatic mode, the line rate should be set to zero. In this mode, VS disables the flow control and commits the control to the transport protocol. That also means, if a transport protocol without flow control is used, there is no flow control at all.
- 0x3003 SetLinkLineRate* (link-id, line-rate)
  Defines the available line rate for a specified link. The command behaves analogously to 0x3002, but it sets the line rate for links instead of for flows.
- 0x3004 InstallFlowRule (flow-id, rule)
  Installs a rule bound to a specific flow. The rule is encoded as a string. After installing the rule, VS responds with a RuleInstalled packet to notify the rule-id, or with a RuleInstallationError.
- 0x3005 InstallLinkRule (link-id, rule)
  Installs a rule bound to a specific link. The rule is encoded as a string. After installing the rule, VS responds with a RuleInstalled packet to notify the rule-id, or with a RuleInstallationError.
- 0x3006 InstallRule (rule)
  Installs a rule not bound to a specific flow or link. The rule is encoded as string. After installing the rule, VS responds with a RuleInstalled packet to notify the rule-id, or with a RuleInstallationError.
- 0x3007 ActivateRule* (rule-id)
  Activates a rule by its rule-id. The rule must be installed before. If the rule is successfully activated or already active, VS responds with an ACK; otherwise VS responds with a RuleNotInstalled error, when the rule is not installed.
- 0x3008 DeactivateRule* (rule-id)
  Deactivates a rule by its rule-id. The rule must be installed before. If the rule is successfully deactivated or already inactive, VS responds with an ACK; otherwise VS responds with a RuleNotInstalled error if the rule is not installed.
- 0x3009 RequestStackOptions (flow-id)
  Requests the currently set stack options for the specified flow-id. If stack options are not supported, VS will answer with a NotSupportedError.
- 0x300A ResponseStackOptions (stack-options)
  Responses on the request for currently set stack options.

A stack-option is a list of key value pairs, containing an 8 bit unsigned integer key as reported with the `StackOptions` message and a 16 bit unsigned integer value. If stack options are not supported, VS will answer with a `NotSupportedError`.

- `0x300B SetStackOption* (stack-option, value)`
  Sets a stack option specified by the `stack-option` field, a 8 bit unsigned integer key as reported with the `StackOptions` message. The `value` field contains a 16 bit unsigned integer value for the stack option. VS will respond with an `ACK` when the option is set or with a `TransitionError` if something went wrong. VS will answer with a `NotSupportedError`, if stack options are not supported.
- `0x3010 SetScheduler* (flow-id, behavior)`
  Sets the scheduler for a respective flow. Supported scheduling policies are represented as 8 bit type. We distinguish between northbound (packets incoming to VS from applications) and southbound (packets being scheduled to a link) scheduling. For northbound scheduling we envision the following scheduling policies:
  - *0x0 Automatic (TM)*: Scheduling is up to VS. This is the default for every flow.
  - *0x1 FlowPriority (FP)*: Each flow gets a customized priority according to Table IV. Incoming packets of a flow are processed according to their priority. By default, every flow has a normal priority. VS is not in charge to handle starvation issues.

TABLE IV
PRIORITY VALUES FOR PRIORITY-BASED SCHEDULING

Pattern	Priority
B00	low
B01	normal
B10	high
B11	reserved

Southbound scheduling only matters when multiple links are used for one flow, since VS has a separate scheduler for every flow. The prioritization of a single flow can be done by the northbound scheduling because VS can only send packets that are already scheduled through the northbound scheduler. We envision the following scheduling policies:

- *0x0 Automatic (TM)*: Scheduling is up to VS and is default for every flow.
- *0x2 RoundRobin (RR)*: Schedules the packets in a round robin manner to the activated links. By now, VS uses this in automatic mode by default.
- *0x3 LinkPriority (LP)*: Assigns a priority to a link, according to Table IV. By default every link has a normal priority. By changing the priority of a link, the packets are scheduled in priority order until the buffers of the respective links are full. Links with the same priority are scheduled in a round robin manner.
- *0x4 LowLatencyFirst (LLF)*: Schedules to the link with the lowest latency until its buffer is full.
- *0x5 HighLineRateFirst (HLRF)*: Schedules to the link with the highest line rate until its buffer is full.

VS responds with an ACK packet when the scheduling policy has been successfully changed.

III. USAGE SCENARIOS

In this section, we explore the usage of TROLL and its communication behavior for multiple scenarios. As our experience is based on VS as edge device management software, we use VS's behavior in our examples.

Standard patterns: On start up of a device or when entering a new network, VS registers itself at the controller. After that, VS reports all its available links and the available stack options. As an application initiates a new flow, VS reports it to the controller with the flow properties. When applications finish their communication, VS reports the network flow as completed. In contrast to OpenFlow devices, VS starts immediately with the communication with the protocols requested by the application (if no rule is installed) and does not wait for a decision of the controller.

Use monitoring: The controller can subscribe monitoring information from VS to enrich its data for network optimization. The controller has to request the available channels first. VS answers with the available channels representing monitoring data. The controller then subscribes the relevant channels. After that, VS sends the last available channel value, and sends an update every time the value changes. Also, VS can subscribe monitoring data from the controller. This is intended for situations when VS has to execute rules that are based on the network status.

Direct commands: For registered flows, the controller is able to send direct commands to VS. With the current command set, it is possible to set network protocols on every OSI layer. This includes the physical layer by setting a linkID as physical layer "protocol". VS manages the build and transition between stacks automatically, including the protocol choice for not specified layers. When stacks are bound to an interface, it is possible to create multiple stacks for one flow. This enables multipath connections. VS would manage the transport qualities (e.g., packet ordering) like the requested protocol would do it. For example, when the application requests TCP, VS would use MPTCP [6] or act like it [2]. If the application requests UDP, VS would do nothing. The command set also supports to specify a line rate for a link or a flow. This could be helpful to support congestion control algorithms and to solve problems when a network is overloaded.

Working with rules: To minimize the communication overhead and to relieve the controller, the protocol supports the installation of ECA rules. For example, if the controller wants to change protocols on a link by default, it is possible to install a rule bound to the respective link. Another example would be to offload traffic to a cellular link when the current WiFi connection is not sufficient for the needed throughput. A rule has to be installed first. After the installation, a rule can be activated or deactivated as needed. VS could subscribe monitoring channels if this is specified in the rule.

For better illustration, we present a practical usage scenario: On a big fair where many people attend, network problems are well known. Internet Service Providers would have experience from the last years, that congestion occurs very likely. To tackle that, they would provide WiFi hotspots in the pavilions as offloading channels and use their cellular base stations as SDN controller to install offloading rules for this area. If congestion occurs in the cellular network, the base station SDN controller could limit the line rate for the cellular network to enable messaging, but switch the bigger flows (traffic that is probably not urgent) to the offloading channel.

IV. Implementation

We implemented a subset of TROLL to show the feasibility and expressiveness of our protocol design. It is publicly available[5] The implementation includes five commands:

- `0x1001 RegisterHost`: On start up, the host registers itself to the controller.
- `0x1002 NewLink`: On start up, VS sends the available network links with a randomly generated `link-id`. Currently, changes on the link availability is not implemented.
- `0x1004 NewFlow`: When VS registers a new network flow, it will be reported to the controller with a `flow-id`. It is generated out of source and destination port.
- `0x1005 FlowCompleted`: When a flow is completed, VS reports this to the controller with the `flow-id` and the state 0. Other states are currently not implemented.
- `0x3001 SetProtocol`: With this action, the controller can set the used protocol stack. Since we did not implement the protocol support exchange, the support of TCP/IPv4 and UDP/IPv4 is assumed.

For testing the implementation, we built up a virtual topology within the ToMaTo testbed [7]. The topology contains one SDN controller node and two VS nodes. All nodes are connected together through a network switch. A snapshot from the topology is shown in Figure 3.

When VS starts, it connects to the SDN controller and reports the host id and the available links. Then, we start a packet generator to simulate a network flow. After VS recognizes the flow, it reports it to the SDN controller. Then, we switch the used protocol by setting a new protocol stack (change from UDP/IPv4 to TCP/IPv4). After that, we stop the packet generator and VS reports the flow completion to the SDN controller. We survey the process with the log files of VS and the SDN controller tool. We also observe that there is no delay measured for all tasks.

On controller side, we implement an SDN southbound API library to enable scripting and manual control over a GUI. Additionally, we started to implement a plugin for the SDN controller OpenDaylight [5].

[5]The implementation is based on a bachelor's thesis, available under https://github.com/tklab-tud/troll. Also, we have an ongoing implementation of a framework for using TROLL in our project (available by fall 2017).

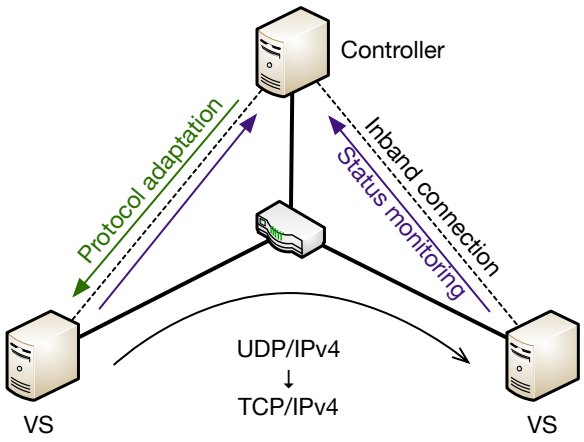

Fig. 3. The test topology for using the protocol with VirtualStack

V. Conclusion and Outlook

In this paper, we presented TROLL, a protocol to control protocol virtualization software (e.g., VS) at edge devices. We showed the protocol structure, the available commands and the communication behavior, also illustrated by some usage scenarios and a short implementation. TROLL brings significant improvement to the flexibility, adaptability, and efficiency of future fully software-defined network systems.

In the future, we will investigate whether TROLL fits to upcoming features of edge device SDN software. We have designed TROLL to fit to the capabilities of VS. Since VS is still evolving, maybe we need to evolve TROLL too. Additionally, we want to integrate some security features to ensure a secure operation within the shared links.

Acknowledgment

This work has been funded by the German Research Foundation (DFG) as part of the project A1 and B2 within the Collaborative Research Center (CRC) 1053 – MAKI.

References

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

[2] J. Heuschkel, A. Frömmgen, J. Crowcroft, and M. Mühlhäuser, "VirtualStack: Adaptive Multipath Support through Protocol Stack Virtualization," in *INC*, 2016.

[3] J. Heuschkel, I. Schweizer, and M. Mühlhäuser, "VirtualStack: A Framework for Protocol Stack Virtualization at the Edge," in *LCN*, 2015.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM*, 2008.

[5] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *WoWMoM*, 2014.

[6] 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.*, 2011.

[7] D. Schwerdel, D. Hock, D. Günther, B. Reuther, P. Müller, and P. Tran-Gia, "Tomato-a network experimentation tool," in *TridentCom*, 2011.

[8] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *HotSDN*, 2013.

[9] M. Stein, A. Frömmgen, R. Kluge, F. Löffler, A. Schürr, A. Buchmann, and M. Mühlhäuser, "TARL: Modeling Topology Adaptations for Networking Applications," in *SEAMS*, 2016.

[10] T. Zhou, G. Xiangyang, Y. Hu, X. Que, and W. Wendong, "PindSwitch: A SDN-based protocol-independent autonomic flow processing platform," in *Globecom Workshops*, 2013.