Multipath Cloud Federation

Maël Kimmerlin*, Peer Hasselmeyer†, Andreas Ripke†
*School of Electrical Engineering, Aalto University, Finland
†NEC Laboratories Europe, Heidelberg, Germany

***Abstract*—Deploying applications in federated clouds is becoming increasingly important. The ability to place application components in various locations is appealing to a large set of distributed applications. Components spread across a number of clouds still need to communicate with each other. Existing solutions for interconnecting clouds are using only single paths between sites, even though clouds typically have multiple up-links. This paper introduces our work on a MultiPath TCP (MPTCP) proxy suited for cloud interconnection. The proxy transforms TCP streams into multipath connections using MPTCP. By virtue of MPTCP, the bandwidth of multiple links is made available to the proxied connections and resilience can be improved without the need for additional fail-over mechanisms. The proposed solution works transparently for applications, making changes to existing applications unnecessary. Measurements of the implemented scheme confirm the additional throughput achievable and the gain in resilience.**

Keywords—Cloud federation, multipath, MPTCP.

I. INTRODUCTION

With the migration of applications to the cloud, it became apparent that a single cloud location does not always fulfill all the requirements of applications. In particular applications requiring low latency or a high level of resilience can benefit from being installed in multiple locations. Applications being available in multiple locations have a higher chance of being located close to their users. Closeness to users means reduced latency for applications, which is important for many of them, specifically for interactive ones, such as online gaming, industrial internet, and augmented reality. The shift towards Edge and Fog computing is in line with this trend.

Another benefit of a multi-cloud deployment strategy is increased resilience. Different clouds are usually connected to infrastructure providers independently, both in terms of electricity as well as communication infrastructure. The independence of infrastructure connections means that failures of such connections are independent, making their concurrent happening unlikely and thereby increasing availability of services.

Applications running in a multi-cloud environment need to communicate among their distributed components. As such communication is usually internal, non-user-facing traffic, it needs to be protected from eavesdropping and interception. This is typically achieved by interconnecting cloud sites via tunnels or virtual private networks (VPNs), providing confidentiality of information exchange by the use of encryption.

Tunnels and VPNs provide individual paths between the components placed in different cloud locations, solving the interconnection problem from a functional point of view. With current solutions, the use of individual paths directly translates into the use of single connections. Such single connections do not make use of multiple communication uplinks, which cloud data centers usually have. Although communication resilience in case of an outage of an uplink can be achieved by transferring inter-cloud traffic to one of the remaining uplinks, this fail-over mechanism needs to be designed and implemented specifically. Furthermore, the use of only one single uplink means that the bandwidth available on other uplinks goes unused. Transient bottlenecks on the used link can therefore not be mitigated by utilizing other uplinks as well. Even in the case of utilizing all available uplinks by hashing flows to the uplinks, traffic distribution is static and has flow granularity. The ideal uplink selection method, though, would work on a per-packet basis and would also consider the current state of the uplinks.

To reap the benefits of simultaneously utilizing multiple paths for communication between two endpoints, multipath TCP (MPTCP) has been developed and standardized by the IETF as RFC6824 [1]. The MPTCP protocol is an extension to traditional TCP and allows applications to use multiple paths in a network for their data exchange. MPTCP splits up single TCP connections into multiple streams which are transported via (partially) disjoint networking paths between endpoints. MPTCP provides mechanisms that allow its streams (called subflows) to coexist with traditional TCP connections. In particular, multiple MPTCP subflows which happen to be placed on the same link will only consume an aggregated bandwidth share as a single regular TCP connection would. To the existing (MPTCP-unaware) networking infrastructure (including switches, routers and middleboxes) MPTCP subflows look like regular TCP connections and are therefore treated as such, greatly reducing the risk of unexpected packet treatment by such middleboxes.

Applications wanting to use the MPTCP protocol do not need to be modified. They only need to have access to code implementing it. This is usually done by including the MPTCP protocol stack inside the operating system, e.g. as a kernel module on Linux-based systems. As the interface to the networking stack (e.g. Unix sockets) is the same for TCP and MPTCP, applications do not need to be aware of the use of MPTCP. Unfortunately, getting an MPTCP-capable kernel often turns out to be difficult for pre-packaged system images or virtual machines. Furthermore, current Linux distributions do not provide pre-compiled MPTCP packages. Users rather need to patch the kernel sources and compile the kernel on their own, or install pre-built MPTCP-enabled kernels from third-party sites. Use of kernels with MPTCP support is therefore limited.

But even if MPTCP support would be wide-spread, its use in data centers would still be restricted, as users hosting their applications inside data centers are usually unaware of the existence of multiple uplinks. Specifically, kernels wanting to establish multiple MPTCP subflows need to have access

978-1-5090-4026-1/17/$31.00 © 2017 IEEE

to multiple network interface cards (NICs) sending traffic to separate uplinks. Alternatively, a single NIC can be used in case the traffic manager in the data center distributes TCP connections to the available uplinks based on source IP addresses or TCP ports. In that case, though, the MPTCP code must be aware of this traffic distribution method and needs to know how many uplinks there are, in order to create an appropriate number of subflows on different IP addresses or TCP source ports. As this needs to be done for each and every kernel, it quickly becomes unmanageable. Looking at this lack of infrastructure support for MPTCP-aware end-points it is no surprise that MPTCP adoption is lagging behind its promises.

With this in mind, we have developed a solution which enables the utilization of the multiple uplinks of cloud data centers while not requiring applications to use the MPTCP protocol or to even know about MPTCP and multiple uplinks. The basic idea is to intercept regular TCP connections and to transform them into multiple MPTCP subflows. On the other end of the connection, the reverse has to happen – MPTCP subflows need to be reassembled into a single, regular TCP stream. The transformation from TCP to MPTCP and back is done by the MPTCP proxy which is introduced in this paper. Additional actions are required to steer traffic through the proxy and to the appropriate data center uplinks.

An MPTCP proxy has been implemented [2] based on the SOCKS architecture and protocol [3], using Shadowsocks [4]. The performance of that MPTCP proxy for cloud interconnection has been measured within our distributed testbed between Finland and Germany. The measurements show that bandwidth capacity of additional links is automatically being utilized and that fail-over between links happens immediately, automatically, and without interruption of existing traffic.

MPTCP is a multipathing solution for TCP connections only. The solution proposed in this paper is therefore not used for communication protocols other than TCP. Having said that, it has to be noted that TCP traffic currently represents the largest share of all Internet traffic, mainly for Web page and video transmission [5]. The MPTCP proxy is therefore improving performance and resilience of a large portion of the overall traffic. Example workloads for our MPTCP proxy solution are High-Energy Physics jobs requiring large file transfers or distributed in-memory databases such as Hyrise-R. More information on these workloads can be found in [6].

After looking at existing work in Section II, the architecture and the implementation of the MPTCP proxy within the environment of OpenStack-based cloud systems are introduced in Section III and Section IV of this paper, respectively. Measurements and their evaluation are described in Section V. Section VI summarizes the results and presents plans for future work.

II. Related Work

Cloud federation is an active field of research and development. Example implementations include the VPN-as-a-Service (VPNaaS) solution which comes with OpenStack [7] and the interconnection agent introduced by Kimmerlin et al. [8]. Such solutions use tunneling mechanisms to shuttle traffic back and forth between cloud sites. While these implementations solve the functional problem of interconnecting distributed clouds, they are using just single paths between the sites. The work presented in this paper enhances the mentioned interconnection solutions with multipath capabilities utilizing multiple uplinks.

The multipath capability introduced in this paper makes use of the MPTCP protocol. MPTCP is being used in various segments of the network and for a number of applications. For example, Apple uses MPTCP in its Siri application for faster handover between WiFi and mobile cellular networks [9]. Grinnemo and Brunstrom study the impact of MPTCP on different types of cloud traffic and show that the use of MPTCP can reduce latency for high and mid-intensive traffic in many scenarios [10]. Inside the data center, Raiciu et al. [11] have shown that MPTCP can improve network performance as well as infrastructure utilization by spreading traffic across the multiple paths which are typically available in data center core networks.

In access networks, MPTCP is being used commercially to improve bandwidth and availability for end users. Mostly, a hybrid access scheme is being used with one link being a fixed line (typically DSL) and the other being a wireless link, usually LTE. By bundling these two links via MPTCP and with MPTCP's automatic distribution of traffic, the two links can be used simultaneously and the packets are automatically distributed to the links, even in case of varying bandwidth. Commercial deployments of hybrid access solutions, including those by Tessares, Korean Telecom, OVH and Swisscom, use MPTCP proxies to convert traffic from TCP to MPTCP on the client gateway and back to TCP on an aggregation server gateway located in the cloud or the provider network. These deployments have many similarities in terms of components used with the architecture proposed in this paper. As access network solutions, they are strictly client-server deployments, while cloud federation is a peer-to-peer application requiring different setups.

This short overview shows that MPTCP has been applied in different scenarios and has shown its benefits in many of them. To the best of our knowledge, MPTCP has not been used for cloud federation so far. This scenario is being analyzed and evaluated in this paper.

III. Architecture

The main aim of our MPTCP proxy is to make the power of multiple network paths available to regular applications. As most applications do not support MPTCP by themselves, the proxy needs to intercept regular TCP connections, transform them into MPTCP ones, and recreate regular TCP connections at the remote end. The MPTCP proxy therefore consists of two components, each one terminating regular TCP connections on its "endpoint" side and talking MPTCP to its peer proxy. Ideally, a large portion of a connection between two endpoints is handled as MPTCP traffic, in order to be able to reap the most benefits from multiple paths in the network. The two parts of the MPTCP proxy should therefore be placed close to the endpoints that they are proxying. For federating clouds, each proxy will typically be placed between the virtual machines and the connection to the external world, usually the uplinks to the Internet. Figure 1 depicts the overall architecture and protocols used.

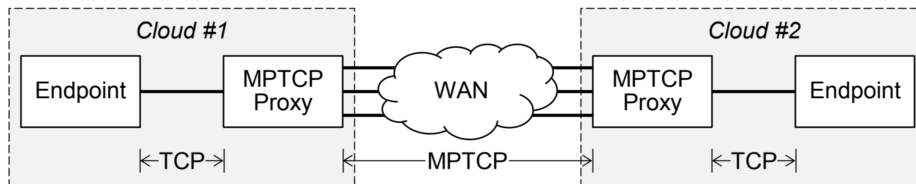

Figure 1. Architecture overview

Multiple solutions to the TCP-MPTCP translation problem exist. One solution is to intercept TCP packets and transform each packet into an MPTCP one. With this approach, there is a direct coupling of individual TCP packets to their corresponding MPTCP incarnations. The approach requires the proxy to implement the MPTCP protocol on its own and to tie its behavior to the TCP connections being proxied, including packet reordering and handling packet loss.

A potential issue is the size of MPTCP packets. MPTCP packets have an additional TCP option included in their header carrying the sequence number for their specific subflow. This option requires additional space in the TCP packet header and therefore reduces the space available for data payload. This issue can be addressed by using MSS clamping, by telling the TCP endpoints to send packets smaller than the maximum segment size (MSS) available on the MPTCP link. The proxy then has enough room for adding the required MPTCP option.

A different approach to translating MPTCP to TCP is to avoid the direct coupling of connection legs, and instead treat each part of the end-to-end connection as a connection in its own right. This approach therefore splits the end-to-end connection into three separate connections, two regular TCP connections "at the ends", i.e. between the MPTCP proxy instances and the endpoints, plus one MPTCP connection between the proxies. The only coupling between the connections is the data that is exchanged end-to-end. In particular, there is no direct coupling at the protocol layer. Congestion control mechanisms are handled individually for all three connections and a loose coupling model is therefore realized. There still is indirect coupling via buffers at the MPTCP proxy which bind the connections together and which can affect protocol behavior on both sides of the proxy. For the implementation of our MPTCP proxy we chose to use this loose coupling model.

Connection splitting as just described is the main purpose of the SOCKS protocol [3]. It can therefore be used to terminate regular TCP connections and establish MPTCP connections to the remote proxy, as long as it has access to an MPTCP-capable networking stack, which can be done by using, for example, an MPTCP-enabled Linux kernel.

In order to make the connection interception transparent to clients, TCP connections from endpoints need to be redirected away from their regular path to the MPTCP proxy. The MPTCP proxy then needs to act as if it were the remote endpoint of that TCP connection. In particular, it needs to use the remote endpoint's IP address for all its responses. On the other end of the connection, the MPTCP proxy can use its own IP address to communicate with its local endpoint. To the endpoint, it therefore looks like it is communicating with the MPTCP proxy as its peer rather than with the application on the other end of the connection. Although this is in most cases not an issue, it might be problematic in case the peer's IP address affects the behavior of the endpoint. Consider, for example, the case where access decisions are made based on specific IP addresses. Allowed access to resources might be denied or access to protected resources might incorrectly be granted, as the client IP is hidden.

In order to solve this issue, we extended the SOCKS protocol to also carry the source information upon connection setup. We added two new SOCKS address types, one for IPv4 and one for IPv6. Each new type contains the destination address, the source address and the source port. Usually, the server proxy parses the packet sent by the client to obtain the destination IP address and port to connect to the server. With our extension, the server proxy also parses the source IP address and port from the payload. For the sake of implementation ease, the server proxy sets Conntrack Source NAT rules, matching on the destination IP address, destination port and source port, to translate the source IP address and port to the address and port used by the client. As the response is being redirected to the proxy automatically, we do not have routing issues. This approach makes the proxy fully transparent to both endpoints.

Cloud federation is a peer-to-peer application in the sense that all cloud sites can initiate connections to their peers. The SOCKS protocol, on the other hand, is designed for client-server communication and is therefore not a direct match to cloud federation. For our MPTCP proxy, we chose to deploy both client and server parts of the SOCKS protocol within each proxy. The appropriate role (client or server) is selected depending on where the connection is initiated from. In case of the connection being started from the local cloud site, the proxy assumes the client role. For connections from other clouds, the proxy acts in the server role. In that way, the MPTCP proxy realizes the peer-to-peer communication model required for cloud federation.

IV. Implementation

The MPTCP proxy has been implemented using a SOCKS5 proxy software, Shadowsocks [4]. It has been deployed inside a virtual machine running a minimal Linux operating system. The Linux kernel has been compiled with the MPTCP patches applied. The whole setup is independent of the IP version used. The proxies can even use a different version of IP between them than in the rest of the setup. IPv6 is natively supported by OpenStack, hence the full setup is IPv6-compliant. The proxy software automatically uses MPTCP in case the remote endpoint also supports MPTCP. The tests reported on in Section V have been executed using Linux kernel version 4.1 and MPTCP version 0.91.

The MPTCP proxy has been integrated into two solutions for federating OpenStack-based clouds: VPNaaS [7] and the

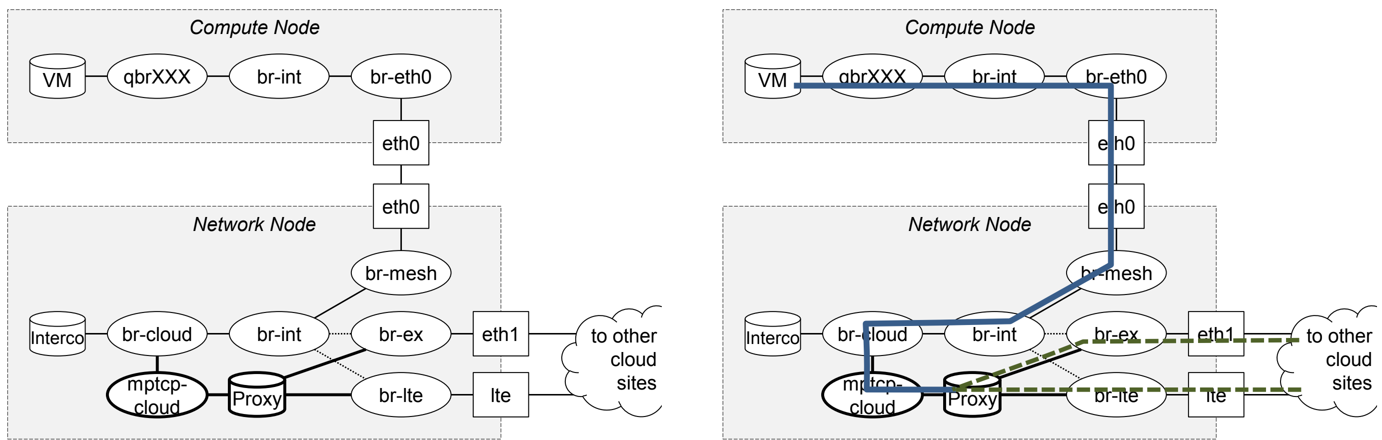

Figure 2. Integration of the MPTCP proxy with OpenStack and the Interconnection Agent

interconnection agent we previously developed [8]. VPNaaS is the legacy project of OpenStack to provide remote connections. The VPN is realized as a layer-3 tunnel between sites. This feature is offered by running an IPsec daemon in the network namespace of the virtual routers in OpenStack. The traffic from one site to the other is routed through the IPsec tunnel where it gets encrypted.

The interconnection agent is a cloud federation mechanism for OpenStack-managed cloud installations aimed at interconnecting cloud sites. Such interconnection is managed by the cloud administrators, but can be used by all tenants inside the clouds. An interconnection agent is run on each cloud site, establishing encrypted tunnels to all clouds it has direct interconnection agreements with. Traffic of all tenants is carried over the tunnels, identified using GRE, VXLAN or GENEVE key identifiers to guarantee isolation of tenant networks.

The integration of the interconnection agent and the MPTCP proxy is shown on the left hand side of Figure 2. One MPTCP proxy VM is deployed for each of the virtual networks that are extended to other cloud sites. The figure shows only one such network with a single tenant VM on it. The VM is located on a compute node, while the MPTCP proxy resides on a node with external connectivity, usually the network node. Inside the network node, the MPTCP proxy is placed on its own bridge, called `mptcp-cloud`. This bridge connects to the `br-cloud` bridge, which integrates the interconnection agent (abbreviated as "Interco" in the figure) into the OpenStack networking system. The bridge `br-cloud` is connected to the internal bridge of OpenStack, `br-int`, from which it receives all the traffic to be sent to the federated cloud sites. `br-cloud` is separating traffic depending on the IP protocol used. TCP traffic is redirected to the MPTCP proxy, while all other traffic is sent to the interconnection agent, which tunnels that traffic to the appropriate peer cloud. The architecture is similar for the VPNaaS implementation, except that the uplinks of the proxies are in the virtual networks. Hence the proxy is only connected to `mptcp-cloud`

Each uplink on the network node has an IP address dedicated to proxying. Connected to each physical uplink, a bridge is created, to which all the proxies for the interconnection agent are connected. Each such bridge has its own private IP subnetwork, on which every proxy gets its own private IP address. Each proxy is also attributed its own TCP port on the public IP addresses of the uplinks. The traffic from the proxies is then translated from the proxy's private IP address to the associated TCP port on the uplink's public IP address. In that way, separation and isolation of different tenants' traffic via a single IP address is achieved.

When exiting the network node, the MPTCP traffic is encrypted using IPsec in transport mode, or in tunnel mode for the legacy VPNaaS. Using transport mode rather than tunneling mode provides several advantages. While the level of security is the same as for VPNaaS or the interconnection agent, the overhead is reduced compared to VPNaaS, as no encapsulation and decapsulation are necessary. Compared to the interconnection agent, the overhead is also highly reduced since the traffic is not tunneled in GRE, VXLAN or GENEVE tunnels.

The TCP traffic is redirected to the proxy VM using OpenFlow rules and to the SOCKS5 client port using an iptables redirect rule. The isolation of the traffic between the SOCKS5 client and server is guaranteed by using different destination ports, one per virtual network.

In the example shown in Figure 2, the network node has two independent uplinks to the Internet (and, therefore, to other cloud sites) via two physical network interfaces called `eth1` and `lte`.

An example traffic flow is depicted on the right hand side of Figure 2 for the interconnection agent. The traffic originating from the VM on the compute node flows into the network node, through `br-int` and from there to `br-cloud`. In `br-cloud`, the traffic is redirected to `mptcp-cloud` from where it goes into the MPTCP proxy (just labelled "Proxy" in the figure). The MPTCP proxy terminates the TCP connection and creates an MPTCP connection to its peer proxy on the destination cloud. The MPTCP proxy creates two subflows, one for each of the two external interfaces (`eth1` and `lte`). These are indicated by the dashed lines in Figure 2. Source routing rules inside the host's operating system steer the flows from the MPTCP proxy to their respective interfaces via the bridges `br-ex` and `br-lte`. Before exiting through the physical network interfaces, `iptables` rules are applied to the traffic which subject it to IPsec encryption

and tunnel encapsulation. On the remote site, the packets flow through the same structure, just in the opposite direction. The MPTCP proxy takes care of properly reassembling the MPTCP subflows into a single regular TCP flow, which is passed on to the destination VM.

For creating connections to its peer proxies in remote cloud locations, the MPTCP proxy needs to know the IP addresses the peer proxies. Such information is currently configured manually. In the future, it will be retrieved from the interconnection agent, which already keeps some knowledge of peer clouds and which could be extended to also know about peer MPTCP proxies.

Initially, the MPTCP proxy knows only a single IP address of the peer proxy and it therefore establishes subflows to that single destination address only. In case the remote MPTCP proxy has multiple uplinks, it will use the MPTCP add address option (`ADD_ADDR`) to inform the proxy of the availability of the IP addresses it is reachable at on the additional uplinks. As per the behavior of the MPTCP protocol, the proxy will then try to establish another subflow to the new addresses using the `MP_JOIN` option. As soon as an additional subflow has been established, traffic will start flowing through it in addition to the previously established flows.

V. Evaluation

Our MPTCP proxy has been implemented as described in the architecture section: the proxy is running inside a virtual machine on top of an MPTCP-capable Linux operating system. It has been deployed at two sites, both running OpenStack Newton. One site is at NEC Laboratories Europe, located in Heidelberg, Germany. The other site is at Aalto University, located in Helsinki, Finland. Although the uplink speeds of the sites are 300 Mbps and 1 Gbps, respectively, experimental results showed that only around 40 Mbps can be achieved between the two sites. The OpenStack configurations of the two sites are similar, with the main difference being the use of VLAN tenant separation at NEC and VXLAN separation at Aalto. Both sites have a single wired Internet uplink each. For our multipath tests, we added an LTE connection as a second uplink to the NEC site. Measurements showed varying speeds on that link, with the maximum being around 10 Mbps. The Aalto site features a single uplink, but two different IP addresses were used for the tests in order to increase path diversity.

For our measurements, we deployed a tenant virtual machine on each of the two sites, running `iperf` for throughput measurements (with 5 parallel streams), using the two federation mechanisms mentioned before. Round-trip-time (RTT) measurements show http transfer times executed with an apache web server and `httping` on the client side, executing an http HEAD request. Since the proxy is diverting only TCP traffic, latency cannot be measured by ICMP as such packets take a different path. Tools like echoping or sending unsolicited ACKs are not suitable measurement methods either as the proxy is terminating the connection and immediately responds to such requests. The measurements would show the round-trip-time of the first leg instead of the whole path. End-to-end measurements were only possible with generating application traffic. Hence, we used a web server and an `httping` client to transfer a minimal request/response.

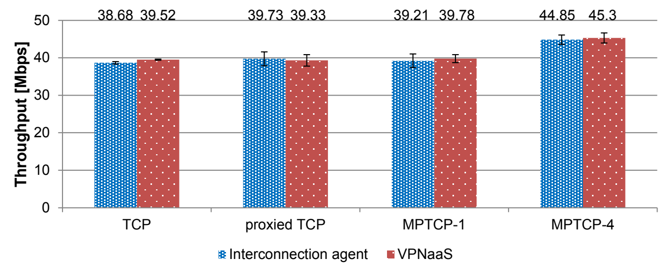

Figure 3. Throughput measurements

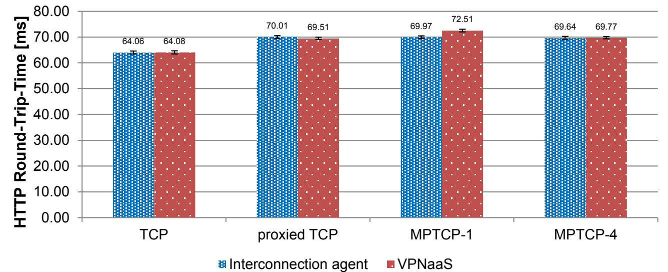

Figure 4. HTTP round-trip-time measurements

We performed a set of measurements with different configuration options. We looked at four scenarios, each one measured with the interconnection agent and the VPNaaS solution. The MPTCP-capable Linux kernel on the proxy was set to use the full-mesh scheduler, meaning it will try to create one connection per local/remote IP address pair. The four scenarios are (a) Interconnection agent or VPNaaS only, no MPTCP proxy (called "TCP" in the figures); (b) Interconnection agent or VPNaaS plus TCP proxy (MPTCP is disabled, called "proxied TCP" in the figures); (c) Interconnection agent or VPNaaS plus MPTCP proxy with a single flow (called "MPTCP-1" in the figures); and (d) Interconnection agent or VPNaaS plus MPTCP proxy with four subflows (2 IP addresses on each side with 2 physical uplinks on NEC's side; called "MPTCP-4" in the figures). The measurements without the proxy ("TCP") are considered to be the baseline measurements, while the measurements with the proxy in the traffic path show the impact of the proxy and MPTCP on the traffic. For all scenarios, average numbers including standard deviation are presented. The throughput tests were each measured 10 times; the latency tests were done 100 times each. Despite running the experiments over the public Internet, throughput and latency measurements showed consistent numbers across all runs, giving us confidence in reporting reliable figures here.

Throughput and HTTP query time figures are shown in Figure 3 and Figure 4, respectively. The increase of around 6 Mbps in the "MPTCP-4" case shows that the MPTCP proxy can make use of the additional bandwidth offered by the second uplink. The measurements also show that there is no significant change in bandwidth when using the MPTCP proxy with one subflow or with MPTCP disabled compared to plain TCP. This is because the bottleneck is on the network throughput.

The round-trip-times as shown in Figure 4 indicate that RTT is only slightly increased with the insertion of the proxy into the traffic path. It has to be noted that the presented

times are for requests on a persistent HTTP connection. Connection establishment time is not accounted for in the figure. We additionally measured the same HTTP transaction with establishing a new TCP connection. These measurements showed a clear increase in the HTTP transaction time, from an average of 138ms to an average of 278ms – basically a doubling of the RTT. This increase is due to the additional data packets which the SOCKS protocol is exchanging before starting the actual data forwarding. As most data transfers are larger than a single packet exchange and the connection establishment time is amortized over all exchanged packets, the increased RTT for connection establishment is tolerable in most cases and in fact more than made up for by the increase in throughput.

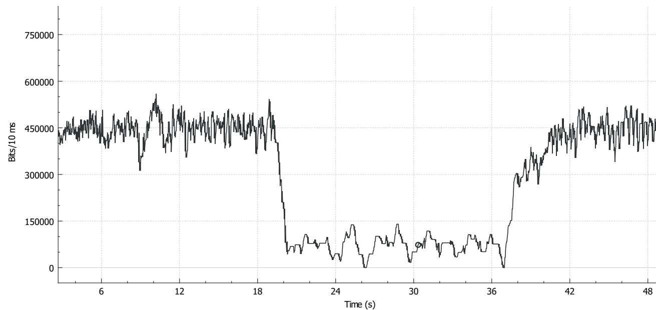

Figure 5. Throughput when cutting and re-establishing a link

Figure 5 shows the connection behavior when cutting and re-establishing one of the links. At around second 19, the wired link is cut and data transfer can continue only through the LTE link. At around second 36, the wired link is re-established and both links are available for data transfer. Figure 5 shows throughput of a data transfer running over the two bundled links, smoothed over 1-second intervals. The drop and the later increase in bandwidth can be clearly seen.

VI. Conclusion and Future Work

This paper shows how MPTCP can be used to increase throughput and resilience for cloud interconnection. We introduced the architecture of a MPTCP proxy which intercepts regular TCP traffic, splits it up into multiple MPTCP subflows, and then reassembles them to again form a regular TCP stream. With this approach, application endpoints do not need to be MPTCP-capable themselves, but can rely on this feature in the MPTCP proxy. Furthermore, applications do not need to know about the multiple uplinks of the cloud site they are running in. It is enough if the MPTCP proxy is configured with such information.

The proxy has been implemented using open source software, including the Linux MPTCP stack. We have measured the proxy's performance in a real-life testbed, connecting clouds in Germany and Finland. Our measurements show that the architecture and implementation are capable of utilizing the bandwidth of multiple links. Latency is increasing due to the proxy added. However, whether it is TCP or MPTCP used by the proxies, and with any number of subflows, the latency remains the same. Resilience is increased, as cutting a link showed only short-term impact on existing flows, besides the obvious reduction in bandwidth.

While the MPTCP proxy is working properly, our tests have revealed a number of issues that could be improved and which we are planning to work on in the future. The two main focus points are scalability and configuration. First, one MPTCP proxy VM is currently run per tenant per site. Such VMs are rather heavy-weight, in particular in terms of memory consumption. We will be looking at alternative solutions, including deployment in containers and using one proxy instance for multiple tenants. Then, the IP addresses of the MPTCP proxy have so far been configured manually. For real-life deployments, an automated configuration mechanism needs to be devised. We will be working on a scheme which acquires information about available uplinks (e.g., from the interconnection agent or another management system) and configures interfaces accordingly.

Acknowledgement & Disclaimer

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866. This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch, "TCP Extensions for Multipath Operation with Multiple Addresses," Internet Engineering Task Force, Internet-Draft draft-ietf-mptcp-rfc6824bis-07, Oct. 2016, work in progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-mptcp-rfc6824bis-07

[2] SSICLOPS Project, "Transparent MPTCP-Proxy Implementation," Aug. 2017. [Online]. Available: https://github.com/SSICLOPS/shadowsocks-libev/tree/transparent

[3] M. D. Leech, "SOCKS Protocol Version 5," RFC 1928, Mar. 1996. [Online]. Available: https://rfc-editor.org/rfc/rfc1928.txt

[4] Shadowsocks. Shadowsocks Homepage. [Online]. Available: https://shadowsocks.org/

[5] Cisco Inc., "Cisco VNI Forecast and Methodology, 2015-2020," Jun. 2016. [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html

[6] M. Kimmerlin, M. Plauth, S. Heikkilä, and T. Niemi, "A Practical Evaluation of a Network Expansion Mechanism in an OpenStack Cloud Federation," in *IEEE International Conference on Cloud Networking (CloudNet 2017)*, September 2017.

[7] OpenStack Foundation, "Virtual Private Network as a Service," Jan. 2017. [Online]. Available: https://wiki.openstack.org/wiki/Neutron/VPNaaS

[8] M. Kimmerlin, P. Hasselmeyer, S. Heikkilä, M. Plauth, P. Parol, and P. Sarolahti, "Network Expansion in OpenStack Cloud Federations," in *2017 European Conference on Networks and Communications (EuCNC)*, June 2017, pp. 1–5.

[9] O. Bonaventure and S. H. Seo, "Multipath TCP Deployments," in *IETF Journal*, Nov. 2016. [Online]. Available: https://www.ietfjournal.org/multipath-tcp-deployments/

[10] K. J. Grinnemo and A. Brunstrom, "A first study on using mptcp to reduce latency for cloud based mobile applications," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, July 2015, pp. 64–69.

[11] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," in *ACM SIGCOMM 2011*, 2011. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018467