Efficient Transparent Access to 5G Edge Services

Josef Hammer, Hermann Hellwagner
Institute of Information Technology, Alpen-Adria-Universität Klagenfurt, Austria
{firstname}.{lastname}@aau.at

***Abstract*—Multi-access Edge Computing (MEC) is a central piece of 5G telecommunication systems and is essential to satisfy the challenging low-latency demands of future applications. MEC provides a cloud computing platform at the edge of the radio access network that developers can utilize for their applications. In [1] we argued that edge computing should be transparent to clients and introduced a solution to that end. This paper presents how to efficiently implement such a transparent approach, leveraging Software-Defined Networking. For high performance and scalability, our architecture focuses on three aspects: (i) a modular architecture that can easily be distributed onto multiple switches/controllers, (ii) multiple filter stages to avoid screening traffic not intended for the edge, and (iii) several strategies to keep the number of flows low to make the best use of the precious flow table memory in hardware switches. A performance evaluation is shown, with results from a real edge/fog testbed.**

***Index Terms*—5G, Multi-Access Edge Computing, MEC, Patricia Trie, SDN, Software-Defined Networking**

I. Introduction

MEC services and convenient access to them are important building blocks of 5G networks and applications. In [1] we proposed a solution to *transparently* redirect requests to cloud services to the corresponding services running in local edge hosts. This solution, based on Software-Defined Networking (SDN) capabilities, (i) simplifies the development of applications that use edge computing and (ii) allows existing applications to use edge computing without any modifications.

The services to be redirected to the edge are first registered with a mobile edge platform provider (fig. 1). Each service is identified by its unique combination of domain name/IP address and port number. If a cloud service is not registered with a given edge platform provider, the original cloud service is used as expected. A client's request to a registered service, however, is intercepted by the network (an SDN switch) and automatically redirected to the closest available edge node. The edge service processes the request (potentially having to utilize the cloud service in that process) and responds to the client (UE in 5G terms). To achieve transparency toward the clients, the approach uses the packet filtering and rewriting capabilities of OpenFlow [2] (see fig. 2). We refer to [1] for a description of the fundamental solution, mainly the crucial role of the SDN controller, an initial prototype implementation based on the POX SDN controller, and discussions about the limitations and scaling options of the approach.

This paper presents a considerably improved version of our prototype that realizes many of the performance and scalability improvements suggested in [1]. We also present several new ideas for improving performance, and evaluate our prototype

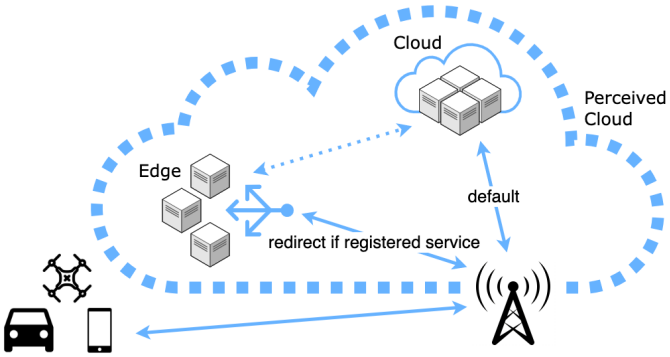

Fig. 1. Perceived cloud vs. real cloud. All requests/responses look like cloud accesses to the client/UE – the redirection to the edge is transparent.

on a real edge testbed. In particular, we focused on minimizing the number of required flows – for the best utilization of the high-speed and expensive ternary content-addressable memory (TCAM) often used for the flow tables in hardware switches [3]. Our new solution uses multiple flow tables – a feature available from OpenFlow 1.1. To future-proof our controller, we chose the version supported by our hardware switch – v1.3 – and thus moved to a different framework – *Ryu*[1].

The contributions of this paper are: (i) an advanced architecture and prototype based on a new SDN controller, deployed in a physical edge/fog computing testbed, (ii) an efficient and scalable implementation using multiple filter stages and proactive flow setup, (iii) several strategies to minimize the number of flows, including matching with unique prefixes and a fast solution to calculate the latter, and (iv) performance evaluation and results.

II. Related Work

Regarding transparent access, we refer to the broad related work discussion in [1]. Several proposals exist to cope with the limited memory for flow tables in hardware switches. Among them, [4]–[7] try to reduce the number of existing flows. Unlike those, our SDN controller has the necessary domain knowledge to create wildcard-optimized flows right away. *Patricia Tries* [8] are common in routers, and [9] uses these tries to calculate subnets for multiple addresses in the trie. [10] uses Patricia tries to find conflicting flows. To the best of our knowledge, our solution presented in section V is the first to use Patricia tries in the opposite way: to find the largest subnet that does *not* include any key present in the trie.

[1] https://ryu-sdn.org/

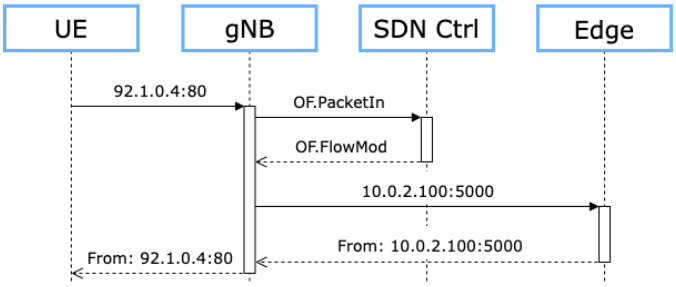

Fig. 2. Routing with a registered service IP: The request is redirected to the closest edge server; transparent to the client (UE). For subsequent requests to the same service, the redirection rule is already known to the gNB; the packet is forwarded directly to the edge host (and not to the controller anymore).

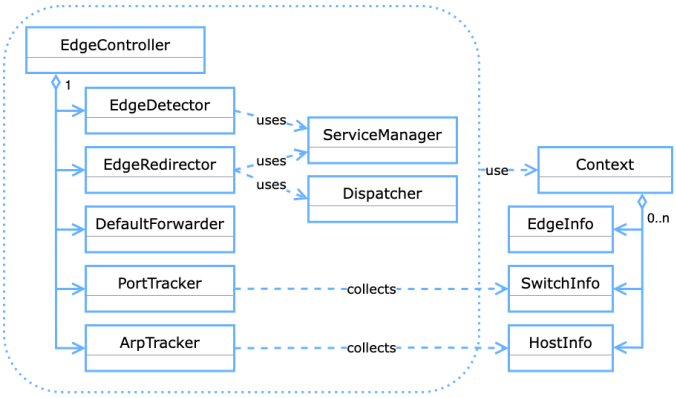

Fig. 3. Architecture of our system.

III. Architecture

Fig. 3 shows the modular architecture of our SDN controller. The core, the *EdgeController*, forwards all incoming requests to the five main modules shown on the left. These five modules work independently of each other. The first three modules represent filter stages (see section IV) and are responsible for their respective flow table(s) in the switch (fig. 5). The fourth module, *ArpTracker*, keeps track of the MAC addresses of our edge server(s). The *PortTracker*, on the other hand, keeps track of the switch output ports for all hosts (in particular for the edge server(s), the clients, and the gateway). These two share their obtained data through the *Context*.

Please note that, for transparent access, we need to know all virtual socket addresses – a reasonable requirement since, in any case, the edge service provider will need an agreement with the edge computing resources provider. The *ServiceManager* maintains this list of virtual sockets. Finally, the *Dispatcher* is responsible for assigning clients to specific edge instances and keeping track of all clients and redirections.

IV. Filter Stages

The main goal of the design of the SDN controller was to optimize the throughput for regular traffic. That is, traffic that shall not be redirected to an edge server (i.e., local traffic and cloud traffic) should be processed by the switch with near-regular speed – as if no filter layer would exist. Furthermore,

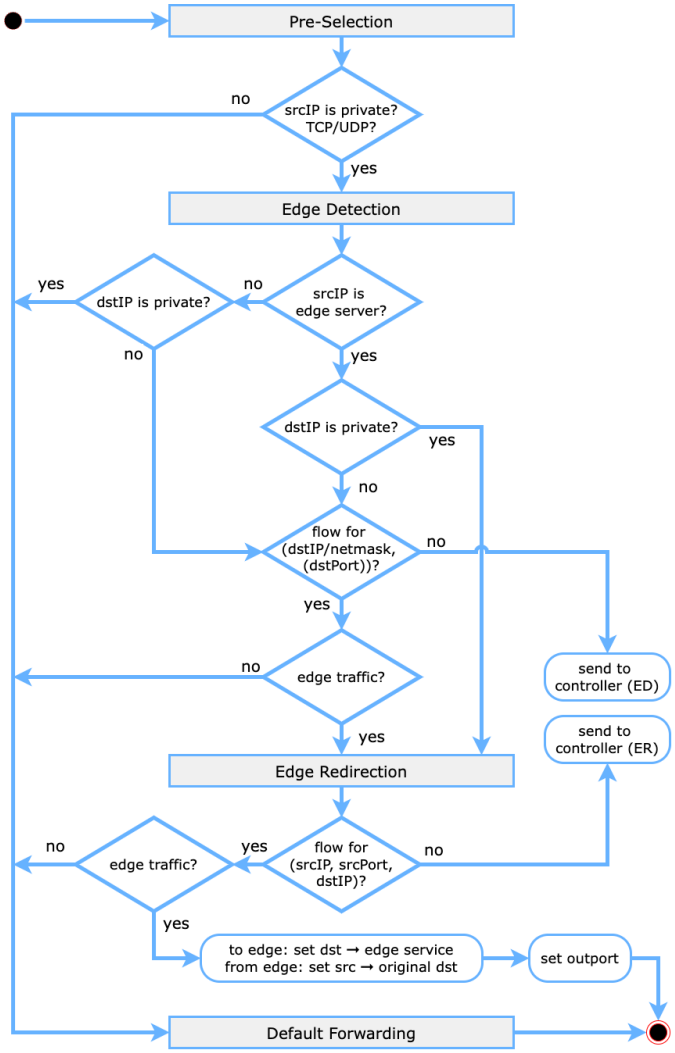

Fig. 4. Four filter stages to improve scalability and to keep the load for the SDN controller at a minimum.

we aimed at a scalable design that allows for thousands of edge services. These goals led to a layered design with four filter stages: (1) *Pre-Selection*, (2) *Edge Detection*, (3) *Edge Redirection*, and (4) *Default Forwarding* (see fig. 4). Please note that we use the terms *rule* and *flow* interchangeably.

A. Pre-Selection

The first stage is a coarse filter to avoid sending traffic to the controller that cannot be edge traffic. Firstly, we examine only protocols relevant to our edge services. Currently, we include both TCP and UDP; however, this choice is arbitrary and could include other protocols. Secondly, we are interested only in traffic coming from a private IP address – traffic coming from the cloud cannot require any intervention. Any other traffic, including, e.g., ARP or ICMP, is forwarded immediately to the *Default Forwarding* stage. No traffic goes to the controller at the first stage – these rules are added permanently, i.e., without any timeout. Whenever an edge switch connects to the controller, the controller adds these permanent flows to

Filter Stage	Table	Priority	Match Fields	Actions
Pre-Selection 1	0	3	arp, eth_dst=02:00:00:00:00:03	Controller
		1	tcp, ipv4_src=10.0.0.0/8	→ 1
		1	udp, ipv4_src=10.0.0.0/8	→ 1
		0		→ 3
Edge Detection 2	1	Max	ip, ipv4_src=10.0.2.100, ipv4_dst=10.0.0.0/8	→ 2
		Max-1	ip, ipv4_dst=10.0.0.0/8	→ 3
		0		Controller
Edge Redirection 3	2	0		Controller
Default Forwarding 4	3	0		Controller, → 4
	4	0		Flood

Fig. 5. Permanent flows added to each edge switch. In this example, `10.0.2.100` is the edge server, and ARP messages to `02:00:00:00:00:03` are responses for our *ArpTracker*.

the switch. In theory, we could include these rules as negated (i.e., exclusion) rules with the highest priority in the second stage. However, as OpenFlow does not allow negation rules, a separate stage makes things a lot easier. The stages themselves are implemented using multiple flow tables in the switch, using the OpenFlow `Goto-Table` instruction to forward packets from one table to another. OpenFlow requires the destination table-id to be greater than the current table-id [2]; therefore, the *Default Forwarding* stage must be last. Moreover, incoming packets are delivered to table-id `0`; thus, we use this table for the *Pre-Selection* stage. See fig. 5 for a list of all permanent flows our controller sets up in the edge switches. Note that this list does not include flows for requests sent to the controller – those *temporary flows* are added on-demand and contain an `idle_timeout`, after which the switch will remove them again.

B. Edge Detection

The second stage tries to find out two things: (1) Is this connection intended for a virtual edge socket, or (2) is this traffic going from one of our edge servers to an internal IP address (which might be a client)? The second question is easy to answer: Since we know all our edge servers' IP addresses, the corresponding flow(s) can be added permanently. As a result, traffic from the edge for a private IP will never go to the controller in this stage but immediately to the *Edge Redirection* stage. Furthermore, we can immediately forward any other traffic for a private IP to the *Default Forwarder*. To achieve an `elseif`-flow, we set a lower priority than for the previous flow (see fig. 5). Note that these permanent flows do not match traffic from the edge towards a public IP – an edge service might well use another transparent edge service.

This leaves us only with traffic coming from other private IP addresses going to a public IP address – potential edge traffic – which is forwarded to the controller to answer the first question: Is the request intended for a virtual edge socket? The controller looks up the destination IP and port in its list of virtual edge sockets. If found, the controller sets up a temporary flow that forwards all requests for this service (independent of the client) to the *Edge Redirection* stage. Note that only a single temporary flow is required – all responses from the edge are redirected permanently to *Edge Redirection*.

If the request is not intended for a virtual edge socket, a temporary flow is set up to send matching traffic to the *Default Forwarder*. Again, only a single temporary flow is required – all responses from the cloud are redirected permanently to the *Default Forwarder*. To further reduce the number of flows and requests that need to be sent to the controller, the controller tries to set up this single flow to match as many destination IP addresses as possible (see section V).

C. Edge Redirection

The third stage finally handles traffic going to virtual edge sockets and coming from local edge servers. There are no default rules in this stage – each new request that does not already have a dedicated temporary flow installed in the switch is forwarded to the controller. The *Edge Redirection* module of the controller first checks whether the request is indeed intended for a virtual socket. If so, the redirection module sends a request to the dispatcher module to select – *for this specific client IP address* – an appropriate edge service running on one of the available edge servers. With the response, the redirection module installs a temporary flow in the switch. The new flow will match subsequent packets from this client and rewrite their destination fields – MAC address, IP address, and TCP/UDP port – to forward them to the selected edge service instead of the cloud. The same modifications are applied to the current packet, which is then sent out on the appropriate switch port. Note that it is necessary to have these flows *per client* instead of *per service* to detect client migration.

If the request was not intended for a virtual socket, the redirection module then checks if it is coming from a local edge server. If so, again, a flow is installed – rewriting the destination fields back to their original values. However, we include this part only to be on the safe side – to speed things up, we proactively add the return flow together with the flow redirecting packets to the edge. With this optimization, the responses from the edge usually do not hit the controller.

Note that there might still be some by-catch in this stage, i.e., regular traffic that could not be filtered out in the previous stages – such traffic is forwarded to the *Default Forwarder* with request-specific temporary flows. Please also note that to speed things up further, we could split this stage into two: one for traffic *to* the edge, and one for traffic *from* the edge. To simplify the concept, we use a single stage in this paper.

In total, a single edge service requires up to three temporary flows – one *per service* in the *Edge Detection* table and two *per client* in the *Edge Redirection* table. At first sight, this might look inefficient. We could achieve the same result with only two flows, which would better utilize the switches' precious memory. However, apart from keeping the number of flows per stage/table low, this approach allows for further optimizations we suggested in [1]: offloading *Edge Redirection* to another switch or utilizing a different controller for *Edge Redirection*.

D. Default Forwarding

The final stage is for all regular traffic – any traffic not redirected to or from the edge. Since this stage is completely decoupled from the first three stages, it can use any forwarding algorithm of choice. The only (slight) requirement is that the *Default Forwarder* must be able to use a specific flow table on the switch. In other words, it must include the table-id with every flow it creates – otherwise, the switch would add the flows to the default table 0. For our prototype, we use a simple layer-2 forwarder that uses *two* flow tables: The first (switch table 3) to detect source MAC addresses it does not know yet, the second (switch table 4) to set the output port for a matching destination MAC address. Whenever a packet for an unknown source MAC address arrives at the controller, the controller adds the source MAC address to both tables. The flow for the first table redirects packets *from* this MAC address to the second table (so they will not be sent to the controller again). The flow for the second table sends any traffic *for* this MAC address to the switch port where the current packet entered the switch. Parallel to being sent to the controller, the packet is sent to all switch ports (`flood`) to speed things up until a flow exists (see fig. 5).

V. Unique Prefix Calculation

The various filter stages already are very effective in optimizing the throughput for regular traffic. Still, we can further reduce both the number of flows and the number of requests to the SDN controller at the *Edge Detection* stage. With switching at OSI layer 3, the controller would set up a separate flow for each public IP address requested by one of the clients. However, since we know all virtual socket addresses in advance, we can create flows that not only match a single public IP address but as many as possible – without matching one of our registered virtual IP addresses, of course.

Our approach is to add a prefix to each public IP address that shall match regular traffic. Consider a request coming in, headed towards `100.0.0.1`. A typical flow would match only this single destination IP address. However, if we would add, e.g., the prefix 25 to the address – `100.0.0.1/25` – this would create a netmask that matches not only one but many potential destination IPs. With a single flow we would not only match `100.0.0.1`, but also any other IP address with the same first 25 bits, i.e., all addresses in the range `100.0.0.0` – `100.0.0.127`. Should a client request another IP address within that range, it will not bother our controller anymore and go straight to the default forwarder. Of course, we cannot choose an arbitrary prefix – we need one that matches as many regular IP addresses as possible without matching any of our registered virtual IP addresses. This prefix we call the *Unique Prefix*. So how do we calculate this unique prefix?

A *Patricia tree* (or *Patricia trie*) is a variant of a binary radix trie [8] where the internal nodes only store the position of the first bit ("prefix") that differentiates two sub-trees. An internal node only exists if at least two of the keys differ at that specific bit of the key. Therefore, for a key with k bits, we need at most k comparisons to find any element in the tree (O(k)). This structure is ideal for storing/retrieving IP addresses where many keys share the same prefix. We use it to efficiently search through all virtual socket addresses, with the keys being a concatenation of IP address and port number. Therefore, the key length is 48 bits (32+16) for IPv4 and 144 (128+16) for IPv6. Since we could not find a suitable Patricia trie module for our requirements, we developed *TinyTricia*, a space-efficient implementation for keys up to 256 bits. With *TinyTricia*, we can keep track of up to half a billion (2^{29}-2; 2^{28}-2 for IPv6) virtual socket addresses with tiny memory requirements. Unlike some other space-efficient implementations, *TinyTricia* allows adding and removing keys at runtime. For 16.78 million (2^{24}) 48-bit keys (IPv4), we would need at most 268 MB of memory – ideal for tracking a vast number of registered edge services from developers worldwide.

However, the main reason for using a Patricia tree is a different property. Patricia trees are frequently used in routers to match an incoming IP address against subnets registered in the tree. Our goal is opposite: We want to *calculate* the largest subnet that includes *only* the search address – neither of them present in the tree. When searching for a key (i.e., IP address + port number) that is *not* included in our search tree, the leaf node will contain the key where the search took all the right branches considering the existing prefix (= internal) nodes. Thus, we only need to find the first bit of the new key that differs from the existing key – this position would be the prefix node to insert the new key. The newly created subtree would contain only a single key – the address we were looking for – thus, we successfully calculated our unique prefix. *Note:* When talking about the position of bits, in this paper we count from left to right, i.e., the leftmost bit is bit number 1.

To illustrate this, let us assume our tree contains only two IPv4 addresses without a port: `100.0.0.255` and `101.1.0.42`. These two addresses would lead to a quite simple Patricia tree with only a single branching, i.e., prefix node, at bit 8 (see fig. 6). Now consider again our request coming in, headed towards `100.0.0.1`. When we look up this IP address in the search tree, we take the left branch since bit 8 of `100.0.0.1` is zero (see fig. 6). In this example, we would not find the search key; however, at the end of the search tree, we would end up with a registered (virtual) IP address, `100.0.0.255` in this case.

Next, we compare these two keys to find the first bit (from the left) that is different. We achieve this by `XOR`ing both keys and searching for the first bit set – bit 25 in this case. This position is the prefix node to add the new key to the tree – our unique prefix. Finally, we add this unique prefix to the IP address we were looking for and create a new flow using the resulting value for the `ipv4_dst` match field (fig. 6).

In our actual controller, we search not only for an IP address but also for the port number. It could happen that we do find the requested IP address in our tree, but the port number does not match. In such a case, our flow needs to be more restrictive, not less – it must match not only the exact IP address but also the exact port number. For requests to virtual socket addresses, the port number is always included in the flow.

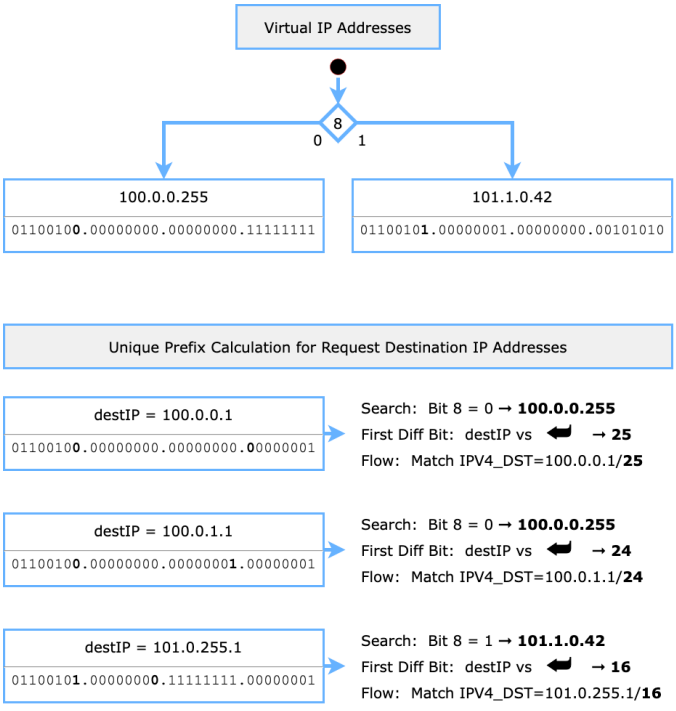

Fig. 6. Patricia tree and unique prefix calculation.

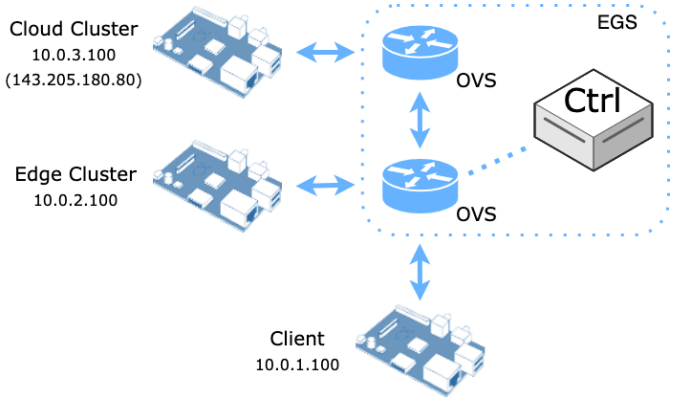

Fig. 7. Virtual network topology used for this paper.

VI. Evaluation

For our research on edge/fog computing we deployed a real testbed named *Carinthian Computing Continuum* (C^3) presented in [11]. C^3 aggregates a large set of heterogeneous resources and includes an edge computing layer. The entry point to the edge layer is the *Edge Gateway System (EGS)*, a 64-bit x86 system with an AMD Ryzen Threadripper 2920X (12 cores, 3.5 GHz, 32 GiB memory) running Ubuntu 18.04.

The other edge resources are 35 Raspberry Pi 4B running Raspberry PI OS (Buster) and five Jetson Nano devices running Linux for Tegra R28.2.1 – all with four cores and 4 GiB of memory. The EGS supports 10 Gbps Ethernet; the other nodes utilize 1 Gbps Ethernet. A layer-3 HP Aruba switch with 1 Gbps ports with a latency of 3.8 μs and an aggregate data transfer rate of 104 Gbps connects these devices.

Various *Ansible*[2] scripts[3] bootstrap the edge layer and configure the nodes to build virtual network topologies. See fig. 7 for the virtual topology used for this paper. The setup includes three Raspberry Pis, representing the client, the edge, and the cloud, respectively. The latter two each run a lightweight Kubernetes cluster (*K3s*[4]). All nodes are connected via two virtual *Open vSwitch (OVS)*[5] switches running on the EGS.

Our SDN controller runs on the EGS, too. Only the switch between the client and the edge connects to the controller. The other switch – connected to our cloud cluster – runs with the default OVS layer-2 switching. For the performance measurement of cloud access, we manually add two permanent redirection flows for the tested public IP address to the second switch – rewriting the destination in the same way as the flows for transparent edge access do. In both the edge and the cloud cluster, we run web services using *Nginx*[6] – all returning a simple plain-text response containing the server name.

For measuring the performance of our solution we use *Curl*[7]. With our `timecurl.sh`[8] script we call the web service 1000 times and save the `time_total` for each request to a JSON file. This value provided by Curl includes everything from the time Curl starts establishing a TCP connection until it gets a response for the `HTTP GET` request. To avoid distorting the result diagrams too much with extreme outliers, we cut off the diagrams (not the data) at 10 ms.

Fig. 8 (left) shows the results for transparent vs. direct access to the edge. Both cases use our SDN controller and require a single hop only. For *Transparent Access* we use a public (service) IP address and port (`143.205.180.80:80`) to access the edge service, whereas with *Direct Access* we access it directly using the server's private IP address (`10.0.2.100`) together with the *NodePort*[9] exposed by Kubernetes. For both types, we see a similar median response time of about 1.5 ms, with both the 25th percentile and the 75th percentile being quite close. The minimum and maximum (excluding outliers) are around 1.1 ms and 2.3 ms, respectively.

Since we use a private destination IP address, the *Direct Access* request goes straight to the *Default Forwarding* stage. Furthermore, the request is unlikely to be sent to the controller in our setup - with layer-2 forwarding, we assume the destination MAC address is already available in the switch tables.

For *Transparent Access* this is different, though. Since these flows are client-specific (only at the *Edge Redirection* stage), any new combination of client and edge service needs to be sent to the controller to set up the corresponding flow. Furthermore, for these flows there might be a far shorter

[2]https://www.ansible.com/
[3]https://github.com/josefhammer/c3-edge/
[4]https://k3s.io/
[5]https://www.openvswitch.org/
[6]https://www.nginx.com/
[7]https://curl.se/
[8]https://github.com/josefhammer/timecurl/
[9]https://kubernetes.io/docs/concepts/services-networking/service/

`idle_timeout` value, after which the flow needs to be set up again. Setting up the flow adds additional latency to the first packet/request – about 6 milliseconds in our tests (*Flow Setup* in fig. 8). Therefore, a well-chosen timeout value plays a significant role – machine learning might help to find the best value based on actual usage patterns.

Remember, though, that our controller is implemented in Python – a real-world SDN controller written in a compiled language would undoubtedly be significantly faster. Furthermore, our proactive return flow setup did not always help – often, the responses arrived faster than the software switch would install the flow. In those cases, the first response was also sent to the controller, adding additional latency. Nevertheless, even with a slow first request, it is still significantly faster than any non-transparent solution that requires a redirection via the cloud. Please also note that with a different *Default Forwarder* the situation might be different – if the first *Direct Access* request needs to go to the controller, then a similar latency penalty would apply as with *Transparent Access*.

Once the flows are set up, both approaches show similar performance – not very surprising given that the switch does all the work and the flow tables are kept short by our algorithm. The benefits of letting the switch do the work would be even more significant with a hardware switch that uses ASICs.

In another test, we measure the overhead introduced for regular cloud access by our layer-4-based switching approach compared to standard layer-2-based forwarding. Since we are interested in the overhead only, the cloud server is just two hops away, without artificial latency (see fig. 7). Again, we compare *Transparent* vs. *Direct Access* – however, this time, we use a public IP address to access a web service in our cloud cluster. Another difference is that – for *Direct Access* – we do not provide a controller at all; instead, we use the OVSs' default layer-2 switching capabilities.

Fig. 8 (right) demonstrates that also for cloud access, both approaches show similar performance once the flows are set up. For both approaches, we see a similar median response time of about 1.7 ms. The 25th/75th percentiles and minimum and maximum are close but further apart than with edge access due to the additional hop. However, the first request (that needs to go to the controller with transparent access) shows a far better performance than with edge access. The faster response is because only a single temporary flow needs to be set up (instead of up to three with edge access; see section IV). Keep in mind that even for the first request, a flow might exist already – due to our unique prefix matching (section V).

VII. Conclusion and Future Work

The multiple benefits of transparent access to edge computing services are highlighted in [1]. This paper presented our approach to efficiently providing this transparent access. Our modular architecture can easily be distributed onto multiple switches/controllers and track thousands of virtual edge services without noticeable performance impact for regular traffic. Our performance evaluation shows a slight delay for the first request when the SDN controller needs to set up

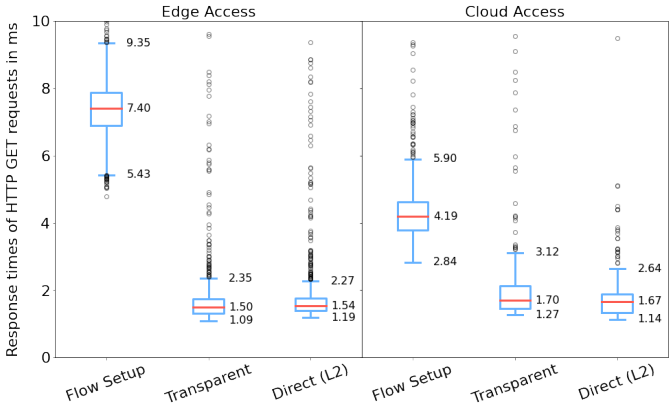

Fig. 8. Transparent vs. direct access (n = 1000) to edge services (left; one hop away) and cloud services (right; two hops away (fig. 7)). *Flow Setup* is mainly required for *Transparent Access* but unnecessary in some *Cloud* cases.

a new flow. This delay is typical for SDN solutions, and our strategies minimize the number of flows and requests to the controller. Moreover, even with a slow first request, our transparent approach is still significantly faster than any non-transparent solution that requires a redirection via the cloud.

The current implementation is highly flexible, and our testbed provides an excellent environment for further research and development. We plan to extend our performance evaluations to hardware SDN switches using realistic traffic data and evaluate client and service migration scenarios.

References

[1] J. Hammer, P. Moll, and H. Hellwagner, "Transparent Access to 5G Edge Computing Services," in *2019 IEEE Int. Parallel Distrib. Process. Symp. Work.* IEEE, 2019, pp. 895–898. [Online]. Available: https://ieeexplore.ieee.org/document/8778343/

[2] Open Networking Foundation, "OpenFlow Switch Specification v1.5.1," Open Networking Foundation, Tech. Rep., 2015.

[3] A. Shirmarz and A. Ghaffari, "Performance issues and solutions in SDN-based data center: a survey," *J. Supercomput.*, vol. 76, no. 10, pp. 7545–7593, 2020.

[4] P. Nallusamy, S. Saravanen, and M. Krishnan, "Decision Tree-Based Entries Reduction scheme using multi-match attributes to prevent flow table overflow in SDN environment," *Int. J. Netw. Manag.*, vol. 31, no. 4, pp. 1–20, 2021.

[5] B. Leng, L. Huang, X. Wang, H. Xu, and Y. Zhang, "A Mechanism for Reducing Flow Tables in Software Defined Network," *IEEE Int. Conf. Commun.*, vol. 2015-Septe, pp. 5302–5307, 2015.

[6] H. Zhu, M. Xu, Q. Li, J. Li, Y. Yang, and S. Li, "MDTC: An efficient approach to TCAM-based multidimensional table compression," *Proc. 2015 14th IFIP Netw. Conf. IFIP Netw. 2015*, 2015.

[7] W. Braun and M. Menth, "Wildcard Compression of Inter-Domain Routing Tables for OpenFlow-based Software-Defined Networking," *Proc. - 2014 3rd Eur. Work. Software-Defined Networks, EWSDN 2014*, vol. 12307, pp. 25–30, 2014.

[8] R. Sedgewick, *Algorithms in C++*, 3rd ed. Addison Wesley, 2002.

[9] M. Portnoi, M. Swany, and J. Zurawski, "An information services algorithm to heuristically summarize IP addresses for a distributed, hierarchical directory service," *Proc. - IEEE/ACM Int. Work. Grid Comput.*, pp. 129–136, 2010.

[10] S. Natarajan, X. Huang, and T. Wolf, "Efficient Conflict Detection in Flow-Based Virtualized Networks," in *2012 Int. Conf. Comput. Netw. Commun.*, 2012, pp. 690–696.

[11] D. Kimovski, R. Matha, J. Hammer, N. Mehran, H. Hellwagner, and R. Prodan, "Cloud, Fog, or Edge: Where to Compute?" *IEEE Internet Comput.*, vol. 25, no. 4, pp. 30–36, 2021.