Scalable Transparent Access to 5G Edge Services

Josef Hammer, Hermann Hellwagner
Institute of Information Technology, Alpen-Adria-Universität Klagenfurt, Austria
{ josef.hammer, hermann.hellwagner }@aau.at

***Abstract*—The challenging demands for the next generation of the Internet of Things have led to a massive increase in edge computing and network virtualization technologies. One significant technology is *Multi-access Edge Computing (MEC)*, a central piece of 5G telecommunication systems. MEC provides a cloud computing platform at the edge of the radio access network and is particularly essential to satisfy the challenging low-latency demands of future applications. Our previous publications argue that edge computing should be transparent to clients. We introduced an efficient solution to implement such a transparent approach, leveraging Software-Defined Networking (SDN) and virtual IP+port addresses for registered edge services. Building on our already efficient approach, in this work, we propose significant improvements to scale our *transparent solution* to large-scale real-world access networks. First, by improving the modularity of our SDN controller design, we enable various options to distribute both the SDN controller's load and the switches' flows. Second, we introduce the *Unique Mask*, a solution superior to the *Unique Prefix* presented in our previous work that considerably reduces the number of required flows in the switches. Our evaluations show that both algorithms perform very well, with the *Unique Mask* capable of reducing the number of flows by up to 98 %.**

***Index Terms*—Multi-Access Edge Computing, MEC, Fog Computing, Software-Defined Networking, SDN**

I. Introduction

The ever-increasing processing and storage demand of the next generation of Internet of Things (IoT) applications led to an evolution in distributed computing and initiated the transition of cloud services closer to the end users [1]. One promising technology is *Multi-access Edge Computing (MEC)*. MEC services and convenient access to them are important building blocks of 5G networks and applications. Particularly IoT applications can benefit significantly from low-latency offloading capabilities. Benefits may include lower battery consumption and easy access to the latest algorithms without frequently requiring to update the software on the device itself. Furthermore, these devices may already be deployed without an economical way to update them.

Therefore, in [2] and [3], we proposed a solution to *transparently* redirect requests for cloud services to the corresponding services running in local edge hosts (fig. 1). 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.

While we presented a highly efficient approach to *transparently access* edge computing services in our previous work, we focused on a single centralized controller so far. We already suggested various options for using a distributed SDN controller in [2], but we have never gone into the details. For the large-scale real-world access networks, however, a highly scalable controller design is required that allows for both the distribution of the required state and improved reliability in case of a failure of a single controller. Thus, in this paper, we focus on improving the modularity of our SDN controller to distribute the components and tasks to multiple controllers efficiently. Furthermore, we introduce the *Unique Mask*, a solution superior to the *Unique Prefix* presented in our previous work that significantly reduces the number of required flows in the switches.

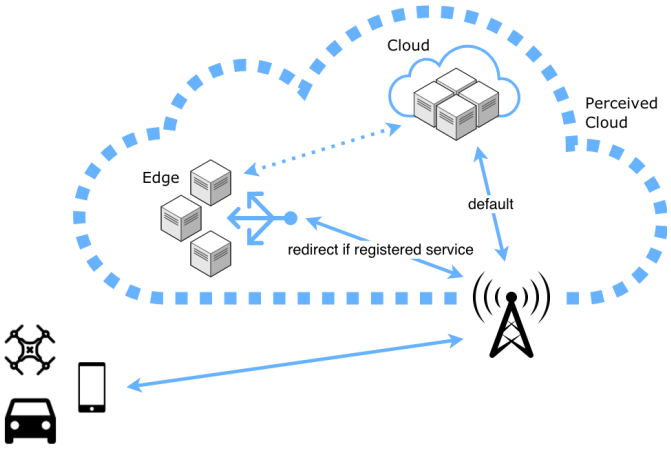

Fig. 1. Perceived cloud vs. real cloud. All requests/responses look like cloud accesses to the client (user equipment; UE) – the redirection to the edge is transparent.

The main contributions of this work are:

- a highly scalable SDN controller design and implementation for providing transparent access to edge computing services at a large scale; and
- a fast solution to minimize the number of flows in the switches that represents a notable improvement over the *Unique Prefix* approach presented in our previous work [3].

The paper has seven sections. First, we provide a short overview of our approach to *transparent access* to edge computing services in section II. Then, we survey the related work in section III. Afterward, we present the architecture of our SDN controller in section IV. Section V introduces the *Unique Mask*, an improvement over the *Unique Prefix* presented in [3]. The corresponding evaluation is shown in section VI, and section VII concludes the paper.

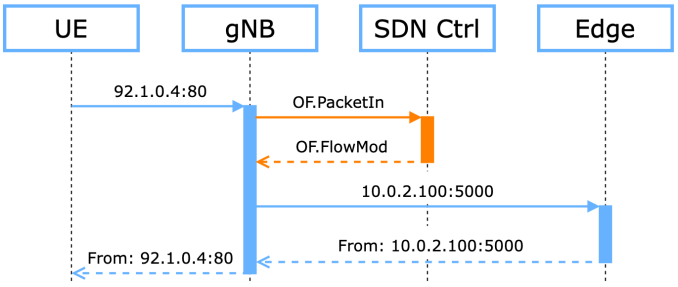

Fig. 2. Routing with a registered service address (IP + port): Using OpenFlow (OF), the request is redirected to the closest edge server, transparent to the client (UE). For subsequent requests to the same service, the redirection rule is already known to the switch (gNB); the packet is forwarded directly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

II. Background: Transparent Access

In our *transparent access* approach [2], [3], the services to be redirected to the edge are first registered with a mobile edge platform provider (fig. 1), identified by their unique combination of domain name/IP address and port number. The network (i.e., an SDN switch) intercepts any request from a client to a registered service and automatically redirects it to the closest available edge server. The edge service processes the request and responds to the client (user equipment (UE) in 5G terms). To achieve transparency toward the clients, our approach uses the packet filtering and rewriting capabilities of OpenFlow [4] (see fig. 2). We refer to [2] for a description of the fundamental solution and to [3] for more details. The latter paper focuses in particular on the performance aspects of our approach and presents a prototype implementation of our SDN controller based on the *Ryu framework* [5] and an evaluation of our prototype on a real edge testbed.

III. Related Work

This section reviews existing approaches for transparent access, scalable SDN controller design, and minimizing the number of flows in the switches.

Regarding transparent access, we refer to the broad related work discussion in [2] citing [6]–[8]. These works all focus on the ETSI MEC Reference Architecture [9] and rely on OpenFlow [4] for switching the traffic.

Regarding scalable SDN controller design, Alsaeedi et al. [10], Shirmarz and Ghaffari [11], and Isyaku et al. [12] provide comprehensive surveys on solutions for adaptive and scalable flow control in OpenFlow-based SDN networks. They also emphasize the propagation delay introduced by centralized SDN controllers in widely separated inter-connected data centers, which calls for multiple distributed controllers to improve both scalability and reliability. However, such a distribution introduces a significant overhead for state synchronization, which leads to scalability concerns [13]–[15]. Our approach addresses these concerns by (i) distributing the controllers at the network's edge and (ii) keeping the global state to be synchronized to a minimum.

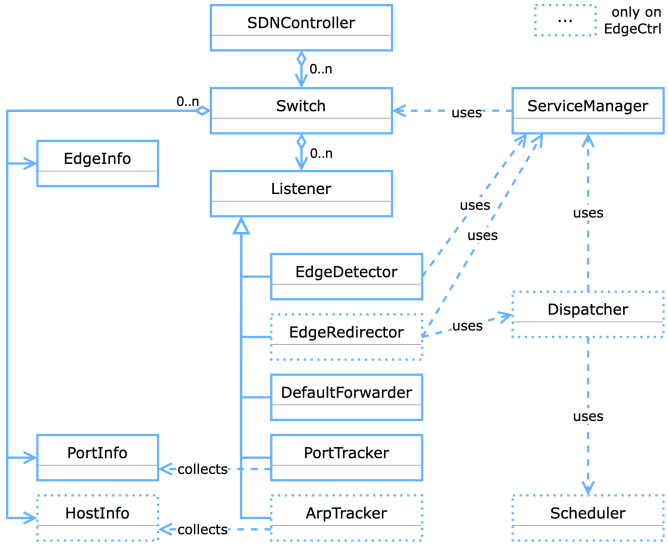

Fig. 3. Architecture of our system. If a separate SDN controller is responsible for the redirection to a specific edge, some components can be skipped on the main controller.

Several proposals exist to cope with the limited memory for flow tables in hardware switches. Nguyen et al. [16] provide a survey on optimal OpenFlow rule placement solutions. Other works propose solutions to reduce the number of flows; among them are [17]–[20]. For their discussion, we refer to the related work in [3]. We also refer to [3] regarding the use of Patricia Tries for finding the largest subnet that does *not* include any key present in the trie [21], [22].

While all these works provide valuable insights and ideas, none of them targets the specific requirements of our *transparent edge* approach. In our approach, we deliberately do not specify the algorithm for switching the *default traffic* to give operators the flexibility to choose their preferred algorithm. Instead, we focus on the algorithm for switching edge-related traffic. This clear separation of concerns allows us to provide an optimized solution for switching edge-related traffic by exploiting the specific requirements for *transparent access*. For example, the forwarding flows for a concrete edge server are a local decision of an edge SDN controller and need not be known to the global SDN controller, avoiding synchronization overhead.

IV. Architecture

In our previous work [3], we presented a modular architecture for our SDN controller designed for efficient transparent access. The entire design targeted optimizing the switching performance for regular, non-edge traffic, which includes minimizing the number of flows in the switches.

This paper presents an improved version of our architecture (see fig. 3). In the previous version presented in [3], while all components operated on a switch-level view, they still had access to the global system view. With the new version, all components are bound to a specific switch and no longer have access to the global network view. We removed the

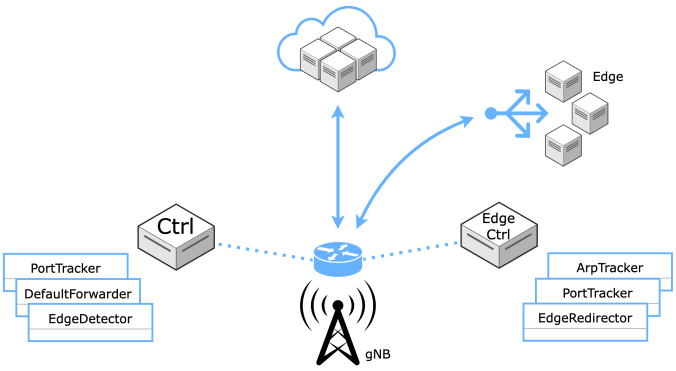

Fig. 4. One switch, two controllers.

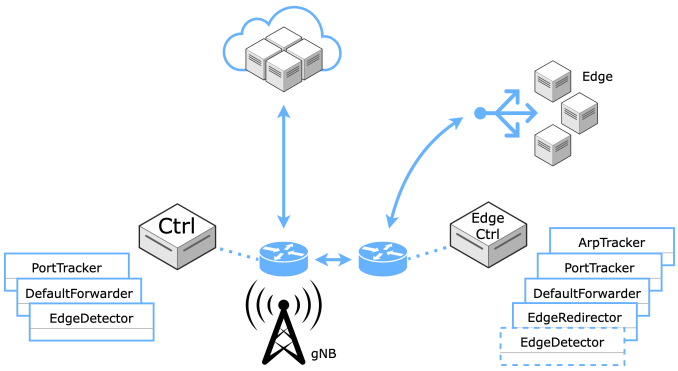

Fig. 5. Two switches, two controllers.

global `Context` component and moved the specific shared information to each switch separately. With the improved modularity of our architecture, we significantly reduced necessary dependencies between the components. As a result, less global data needs to be shared, and it has become easier to skip components not required for a specific switch (see fig. 3).

Thus, in addition to the performance goals mentioned above, the improved architecture allows the distribution of the workload between multiple switches. For high scalability of the entire system, the distribution is done in a way that allows the main ingress switch to focus on edge detection and offloads the edge redirection to other switches. This improved architecture allows several different scaling options. These options can be classified into three main groups as follows.

A. One Switch per Ingress, One Controller

The simple default setup uses one switch *per ingress* and one central controller – the setup we used in previous papers. Note that the central controller is usually connected to multiple ingress switches. While this option is easy to set up, the central controller becomes the bottleneck and the single point of failure of the system. On the other hand, using only a single controller requires fewer redundant controllers to take over in case of a failure.

B. One Switch per Ingress, Multiple Controllers

The previous setup with only a single controller is hard to scale to larger networks. One option for scaling is to offload the edge redirection handling to another SDN controller dedicated to edge service traffic. Fig. 4 shows an example with a single dedicated *Edge Controller*. Setups in this group include (i) sharing a single *Edge Controller* among multiple ingress switches and (ii) distributing the traffic over multiple *Edge Controllers* (*sharding*). With this solution, the main controller only needs to run the components necessary to separate edge traffic from regular traffic – the dedicated *Edge Controller* deals with the actual forwarding.

Although the more straightforward solution, options in this group already allow a more centralized, global core controller that does not require the necessary knowledge for the local edge redirection. Furthermore, since this simplifies the core controller, running redundant instances of the central controller for improved resiliency becomes much more manageable – less state in the core controller means less to synchronize with the redundant controllers.

However, this setup implies that the switch needs to connect to both controllers to send all OpenFlow `PacketIn` messages to both controllers. Depending on the number of `PacketIn` messages, this might increase the traffic overhead significantly.

C. Multiple Switches per Ingress, Multiple Controllers

Another scaling option is to distribute the logic among multiple SDN controllers and the flows among two or more switches (see fig. 5). This solution builds on the previous one and significantly reduces the number of flows per switch. While we could achieve this using established sharding methods, our architecture's filter stages allow us to be more specific and move most edge service-related flows away from the main switch. Thus, the main switch can keep more flows for regular traffic in its fast *ternary content-addressable memory (TCAM)*, speeding up the forwarding.

As presented in [3], our architecture utilizes four filter stages to reduce the number of flows and to improve efficiency: (1) *Pre-Selection*, (2) *Edge Detection*, (3) *Edge Redirection*, and (4) *Default Forwarding*. Each filter stage maps to a separate switch flow table. The *Edge Detection* stage separates regular traffic from edge-related traffic and thus needs to be present in the main switch. However, this design allows us to move the *Edge Redirection* stage with all edge redirection flows to a different switch. Since these flows modify packet headers in both directions (fig. 2), this stage contains twice as many edge service-related flows as the *Edge Detection* stage.

Compared to the previous scaling option, this approach reduces the traffic overhead caused by the `PacketIn` messages. Since the main ingress switch does not hold any edge redirection flows anymore, it does not need to connect to the *Edge Controller* anymore – only the dedicated edge switch needs to connect to the *Edge Controller*. As a result, the *Edge Controller* will only receive `PacketIn` messages for edge service traffic – and not for regular traffic. Note that the edge switches may also be used to switch regular (internal) traffic – we only move them away from the critical ingress

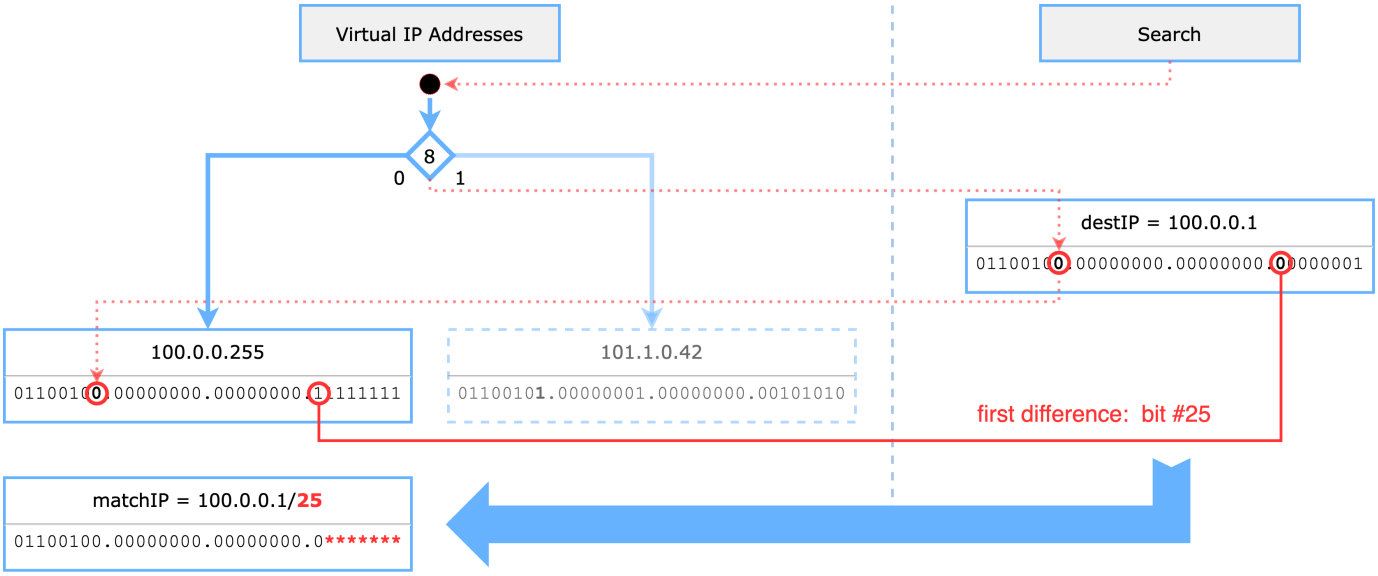

Fig. 6. We use a Patricia Trie to search for registered virtual addresses and to calculate the *Unique Prefix* (shown above). By additionally using all parent prefixes (8 only in this simple example trie) to generate the IP mask for the switch flow, we get the *Unique Mask* (see fig. 7).

traffic. In such a case, the *Edge Controller* needs to run all the components – i.e., including the *EdgeDetector* – to be able to distinguish between regular and edge-related traffic.

V. Unique Mask

For *transparent access* to edge computing services, we cannot only use switching at OSI layer 2 (using the MAC addresses) since we need to detect and redirect requests to registered edge services [2]. Therefore, in the worst case, we would need a separate flow in the switches for each destination address requested by a client. However, minimizing the number of flows is essential for the best utilization of the high-speed and expensive *ternary content-addressable memory (TCAM)* often used for the flow tables in hardware switches [11]. Therefore, in our previous work [3], we presented several strategies to minimize the number of flows, including matching with *Unique Prefixes*.

The *Unique Prefix* (fig. 6) provides an efficient way to minimize the number of flows required in the switches. It utilizes flows with prefix wildcards to match IP addresses for regular traffic. Depending on the specific IP addresses involved, using the *Unique Prefix* already significantly reduces both the number of flows necessary for regular IP addresses and the number of `PacketIn` messages to the controller.

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 match not only `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`. As a result, compared to a flow without our *Unique Prefix*, we can match many regular IP addresses instead of just a single one –

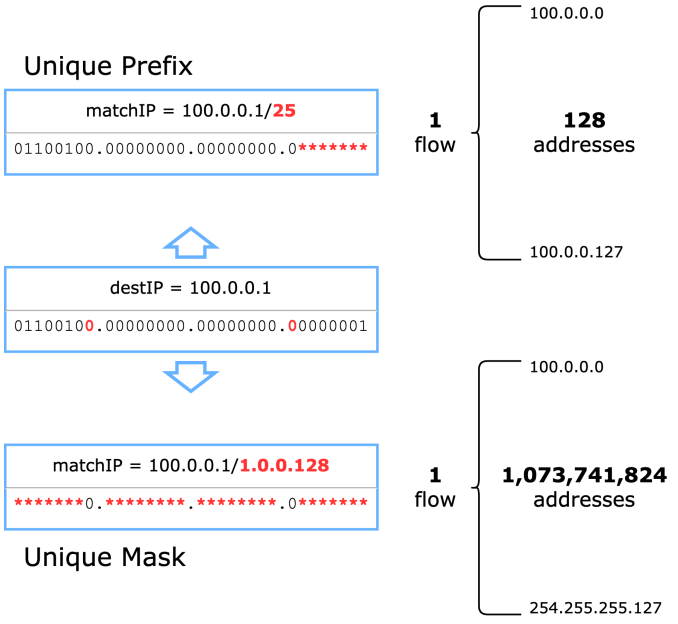

Fig. 7. Unique Prefix vs. Unique Mask. Both lead to a significant reduction in the number of flows by capturing many addresses with a single flow.

in the example above, 128 addresses (see fig. 7). Thus, using our *Unique Prefix* leads to a massive reduction in the number of flows. Furthermore, it also reduces the number of requests to our SDN controller, speeding up and reducing the load of the entire system. Should a client request another IP address within that range, the switch will not bother our controller anymore and will go straight to the default forwarding table.

However, we can do even better by exploiting more information from our Patricia Trie. The OpenFlow Switch Spec-

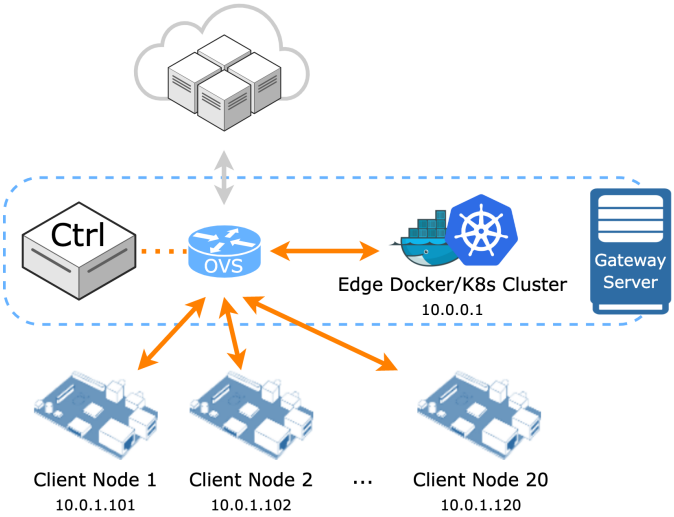

Fig. 8. The topology used for the evaluation. The SDN controller, the virtual OVS switch, and the cluster for the edge services run on the *Edge Gateway Server (EGS)*. The client applications run on 20 Raspberry Pi 4B.

ification [4] does not only allow prefix wildcard definitions (i.e., subnet masks) but also arbitrary bitmasks. These bitmasks allow wildcard bits to be set anywhere in the address, enabling much more powerful pattern matching. We use these bitmasks to create flows that match even more addresses with only a single flow. When we look at our tiny tree in fig. 6, we notice that there cannot be any key between the one we are comparing with (`100.0.0.255`) and its parent prefix node (bit `8`). Consequently, all the bits between the parent prefix (bit `8`) and the *Unique Prefix* (bit `25`) also do not matter. If, starting from bit `9`, only the *Unique Prefix* bit `25` matches, we can already guarantee that our flow will not match any of our registered virtual IP addresses.

Going one step further, if we only match all parent prefixes of `100.0.0.255` as well as the *Unique Prefix*, our flow still will not match any of the addresses registered in our tree. Thus, only bits `8` and `25` must match in this example. Fig. 7 shows the two different bitmasks used for *Unique Prefix* matching (top) vs. *Unique Mask* matching (bottom). The difference in the number of potential regular IP addresses matched can be enormous, as shown in this example.

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. Thus, requests to virtual IP addresses always include the port number in the flow.

VI. Evaluation

For the evaluation, we used the same setup as described in [3] – our *Carinthian Computing Continuum* (C^3) testbed presented in [1]. 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 Server (EGS)*, a

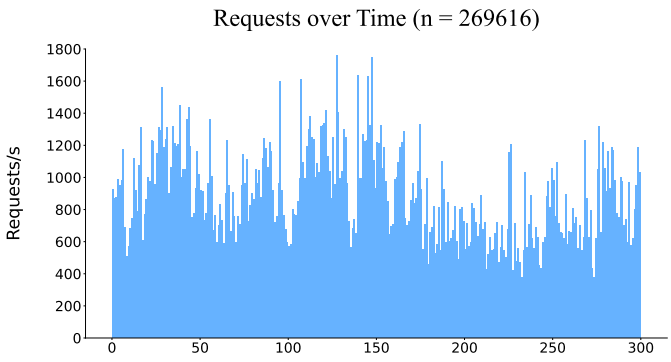

Fig. 9. Distribution of the 269,616 TCP/UDP requests from 99 private IPv4 addresses to 1,595 public IPv4 addresses over the 300-second network trace.

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 – 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 the devices. On top of this system, we use an overlay network that forms a virtual emulation network for our experiments, with the *Open vSwitch (OVS) [23]* virtual switch running on the *EGS* (fig. 8).

A. Unique Mask

In theory, both *Unique Prefix* and *Unique Mask* lead to a significant reduction of flows. However, we wanted to test how well they perform with real-world traffic. To evaluate the effectiveness of both *Unique Prefix* and *Unique Mask*, we used a dataset provided at [24] by the *Tcpreplay* [25] tool. The `bigFlows.pcap` [26] dataset is a "5-minute capture of real network traffic on a busy private network's access point to the Internet" and designed to be "useful for testing performance of switches and network adapters."

The dataset contains 791,615 packets from 132 different applications. Due to our performance optimizations presented in [3], both network-internal traffic and responses from public IP addresses do not affect the number of flows generated by our SDN controller. Thus, we are only interested in requests from private IP addresses to public IP addresses. Out of the total packets in the dataset, there are 269,616 requests from 99 private IPv4 addresses to 1,595 public IPv4 addresses. Fig. 9 shows the distribution of these requests during the 300 seconds. However, since we use five or 20 seconds for the flow idle timeouts, whether we send multiple requests per second or only a few of them does not make a difference. Thus, for identical consecutive 5-tuple requests, we send only one per second, resulting in 55,529 requests.

Remember that our architecture uses a layered design with four filter stages [3], and regular traffic is forwarded by the *Edge Detector* to the *Default Forwarder* (fig. 3). Since these flows do not contain the client's IP address, the number

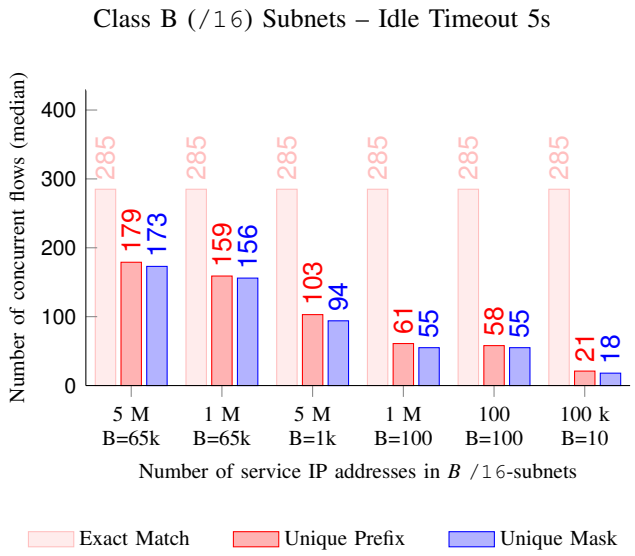

Fig. 10. Number of concurrent flows (median) in the switch when replaying requests from a five-minute real-world network traffic capture with an *idle flow timeout of 5 seconds*. The capture contains 269,616 requests to 1,595 public IP addresses. The service IP addresses are distributed within a specific number of *Class B* (/16) subnets. All service IPs and subnets were generated randomly. The median prefixes are 23, 21, 12, 9, 10, and 7 (from left to right).

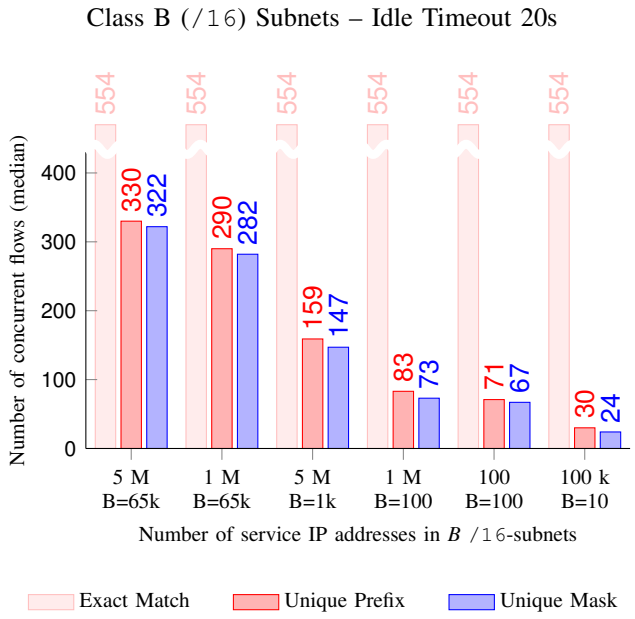

Fig. 11. Like fig. 10 but with an *idle flow timeout of 20 seconds*.

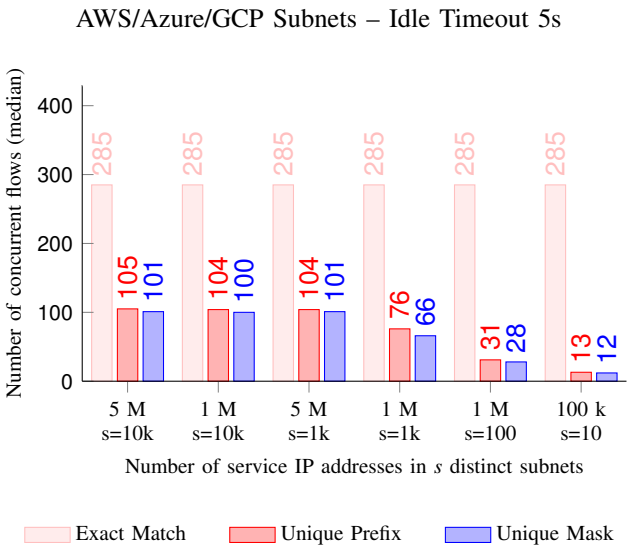

Fig. 12. Number of concurrent flows (median) in the switch when replaying requests with an *idle flow timeout of 5 seconds*. The traffic capture used is the same as in fig. 10; however, this time, the service IP addresses are distributed within a specific number of real-world subnets used by AWS/Azure/GCP. All service IPs and subnets were selected randomly. The median prefixes are 13, 13, 13, 10, 9, and 5 (from left to right).

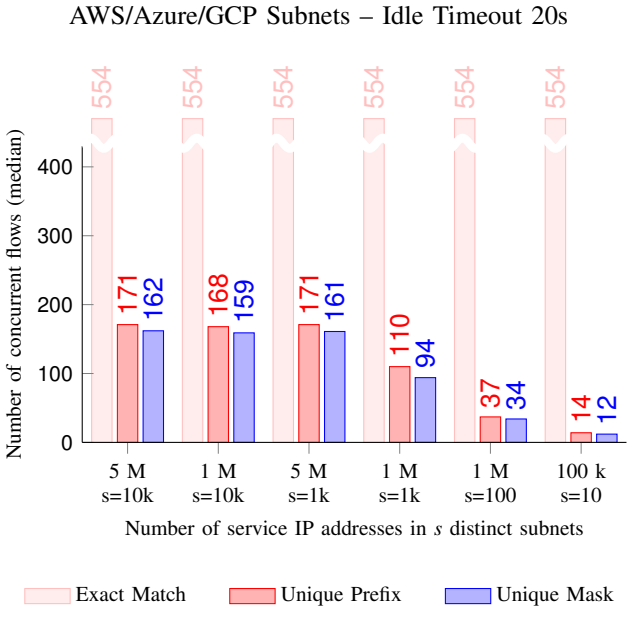

Fig. 13. Like fig. 12 but with an *idle flow timeout of 20 seconds*.

of clients does not affect the number of flows generated. Thus, we distributed the 99 clients available in the dataset on only 20 Raspberry Pis. The 99 Scapy-based [27] client applications send their requests with correct timing according to the timestamps from the dataset, and the SDN controller sets up the flows with a five- or 20-second idle timeout. To read the number of concurrent flows, we used OVS's `dump-aggregate` command once per second. Out of the total 300 seconds where requests were sent, we used only 290 seconds for the evaluation to eliminate bias from the lower number of flows during the ramp-up phase.

Fig. 10 and fig. 11 show the number of concurrent flows (median) in the switch when replaying the requests from the dataset with a five- and 20-second idle flow timeout, respectively. The total number of service IP addresses available in the *Patricia Trie* significantly affects the effectiveness of both *Unique Prefix* and *Unique Mask*; thus, we tested with different numbers of service IPs. Since we assume that a few

big data center providers will host most virtual service IP addresses, we randomly distributed the service IP addresses within 100 *Class B* (`/16`) subnets. Additionally, we added tests with all 65,536 possible `/16` subnets included, tests with 1,000 `/16` subnets, and tests using only ten `/16` subnets. All service IPs and subnets were generated randomly. Note that we use service *IP addresses* only; i.e., we always use the same port since both *Unique Prefix* and *Unique Mask* work on the IP address level only. Using different ports with the same IP addresses for multiple services can multiply the number of services without affecting the numbers shown here.

Frankly, the results from these tests were surprising. While, in theory, the *Unique Mask* should be far superior to the *Unique Prefix*, it performed only slightly better: around 10 %, up to 20 % at best. However, this is partly due to the excellent performance of the *Unique Prefix*. Both algorithms significantly reduce the number of concurrent flows – between 40 % and almost 98 %. A closer look at the prefixes calculated by the SDN controller reveals the reason for the small difference between the two. The median prefixes derived from the test sets are surprisingly low: 23, 21, 12, 9, 10, and 7, respectively. The lower the prefix, the bigger the range of covered IP addresses, the smaller the room for improvement for the *Unique Mask*.

As expected, the effectiveness decreases with the number of service IP addresses in the *Patricia Trie*. When the trie is fuller, the subnets that do not contain any service IP address (as derived by applying the *Unique Prefix/Mask*) inevitably become smaller, matching fewer public IP addresses. However, the most crucial factor is not the number of service IP addresses but the number of *Class B* (`/16`)-subnets containing these service IPs. With 100 `/16`-subnets, we see around 85 % reduction – regardless of whether we use 1 million (10^6) or only 100 (10^2) service IP addresses. In fact, $100,000$ addresses in 10 `/16`-subnets lead to better results than only 100 addresses in 100 subnets. On the other hand, when we distribute 1 million service IPs among all $65,536$ possible `/16`-subnets, the reduction rate decreases to about 50 % and about 42 % with 5 million addresses. Still, even a reduction rate of only 50 % is a significant improvement over using neither *Unique Prefix* nor *Unique Mask*, i.e., setting up flows with an exact match for each public IP address in use.

Not at all surprising is the influence of the two different idle flow timeouts - five and 20 seconds, respectively. When timeouts are shorter, the switch removes idle flows earlier – leading to lower overall flow counts. As a result, lower timeouts slightly reduce the effectiveness of both *Unique Prefix* and *Unique Mask*. However, lower timeouts come at the price of a higher number of requests to the SDN controller. In our tests, the controller created about 30 % to 40 % more flows when using the shorter five-second idle timeouts compared to the longer 20-second timeouts.

To double-check our results, we decided to use real-world subnets used by three big cloud computing platforms (at the time of writing): *Amazon AWS* [28], *Microsoft Azure* [29], and *Google Cloud Platform (GCP)* [30]. We limited the selection

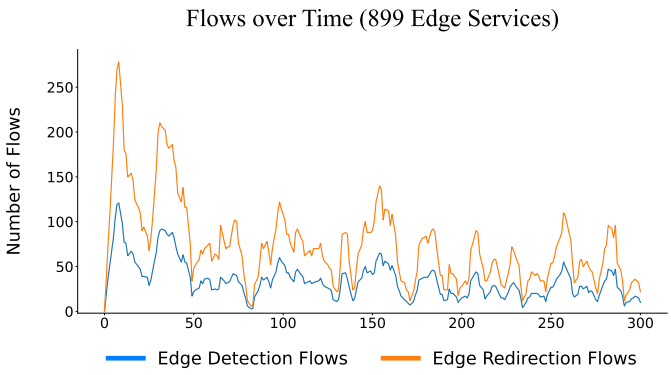

Fig. 14. Distribution of the flows for 899 edge services from 74 clients over the 300-second network trace with an idle flow timeout of five seconds.

to 13,028 subnets that can host at least 256 addresses (`/24` or lower) and chose random sets containing 10,000, 1,000, 100, and 10 subnets, respectively. Please note that these set sizes differ from those in the *Class B*-subnets figures. First, from only 13,028 subnets, we cannot randomly choose 65,536 subnets. Second, with a median size of `/23`, the smaller subnets cannot contain high numbers of IP addresses. Finally, we deliberately present set sizes in the figures that best highlight the tipping points.

Since these subnets are smaller than the `/16`-subnets used in fig. 11, the performances of both *Unique Prefix* and *Unique Mask* are even better. Fig. 12 and fig. 13 show that even with 5 million service IP addresses, we achieve an 80 % and 70 % reduction in the number of concurrent flows, depending on the idle flow timeout. With 1 million service IPs in 100 AWS/Azure/GCP subnets, we have less than half the number of flows than with the same number of IPs in 100 `/16`-subnets. Also, we see that the number of flows remains stable when the number of subnets increases from 1,000 to 10,000. This is reflected by the low median prefixes: 13, 13, 13, 10, 9, and 5. In conclusion, we recommend grouping service IP addresses into only a few `/16`-subnets or grouping them into smaller subnets. This approach allows the best results when using either *Unique Prefix* or *Unique Mask* for reducing the number of concurrent flows in the switches.

B. Distributing the Flows

The third scaling option presented in section IV allows to distribute the flows among two or more switches by moving most edge service-related flows away from the main switch (see fig. 5). In our system, requests to an edge service require three flows per client. Of these three flows, our architecture's filter stages allow moving two to a different switch – the *Edge Redirection Flows*. The remaining third, the *Edge Detection Flows*, must stay on the main switch. To get a perspective about actual numbers, we also ran tests to measure the number of flows required for the network trace used in the previous subsection. However, this network trace neither knows about edge services nor includes requests to these. Therefore, we selected a subset of the available destination addresses as edge

service addresses. First, we used Wireshark [31] to extract all 22,312 *TCP Conversations* from the raw network trace. Then, we filtered for requests going to destination addresses with the target port `80` (HTTP). These 4,610 requests (21 % of the entire trace) from 74 clients resulted in 899 unique destination IP + port combinations, which we configured as edge service addresses in our SDN controller.

Fig. 14 shows the number of edge service-related flows over time when replaying the network trace. For this test, we used an idle flow timeout of five seconds. Without using the third scaling option, both the *Edge Detection Flows* and the *Edge Redirection Flows* are present in the main switch. With the third scaling option, we move the latter (two-thirds of the flows) to a different, dedicated edge switch. This separation would remove almost 300 flows from the main switch at the peak, given the selected network trace and edge services.

VII. Conclusion

In our previous publications [2], [3], we highlighted the multiple benefits of transparent access to edge computing services. Furthermore, we presented how to implement such a transparent approach efficiently. Building on our already efficient approach from these two papers, in this work, we proposed the *Unique Mask* as an effective improvement for further reducing the number of flows in the switches. Our evaluations show that both *Unique Prefix* and *Unique Mask* can reduce the number of flows by 40 % to up to 98 %, with the *Unique Mask* being about 10 % more effective. Thus, the *Unique Mask* is a crucial element for the best utilization of the high-speed and expensive ternary content-addressable memory (TCAM) used in hardware switches. Furthermore, our results show that using fewer subnets for the service IP addresses significantly improves the effectiveness of both algorithms. Additionally, we significantly improved our SDN controller's modular architecture to enhance our system's scalability and presented two different scaling options for large-scale networks.

Acknowledgement

This work received funding from the Austrian Research Promotion Agency (FFG; Grant 888098, Kärntner Fog).

References

[1] 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. [Online]. Available: https://ieeexplore.ieee.org/document/9321525/

[2] 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/

[3] J. Hammer and H. Hellwagner, "Efficient Transparent Access to 5G Edge Services," in *2022 IEEE 8th Int. Conf. Netw. Softwarization*. IEEE, 2022, pp. 91–96. [Online]. Available: https://ieeexplore.ieee.org/document/9844066

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

[5] "Ryu – Component-Based Software-Defined Networking Framework." [Online]. Available: https://ryu-sdn.org/ (accessed 2022-10-30).

[6] T. Taleb, P. Hasselmeyer, and F. G. Mir, "Follow-me cloud: An OpenFlow-based implementation," *Proc. GreenCom-iThings-CPSCom*, pp. 240–245, 2013.

[7] A. Aissioui, A. Ksentini, A. M. Gueroui, and T. Taleb, "On Enabling 5G Automotive Systems Using Follow Me Edge-Cloud Concept," *IEEE Trans. Veh. Technol.*, vol. 67, no. 6, pp. 5302–5316, 2018.

[8] E. Schiller, N. Nikaein, E. Kalogeiton, M. Gasparyan, and T. Braun, "CDS-MEC: NFV/SDN-based Application Management for MEC in 5G Systems," *Comput. Networks*, vol. 135, pp. 96–107, 2018.

[9] ETSI, "GS MEC 003 - V2.1.1 - Multi-access Edge Computing (MEC); Framework and Reference Architecture," ETSI, Tech. Rep., 2019.

[10] M. Alsaeedi, M. M. Mohamad, and A. A. Al-Roubaiey, "Toward Adaptive and Scalable OpenFlow-SDN Flow Control: A Survey," *IEEE Access*, vol. 7, pp. 107 346–107 379, 2019.

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

[12] B. Isyaku, M. S. Mohd Zahid, M. Bte Kamat, K. Abu Bakar, and F. A. Ghaleb, "Software Defined Networking Flow Table Management of OpenFlow Switches Performance and Security Challenges: A Survey," *Futur. Internet*, vol. 12, no. 9, p. 147, 2020.

[13] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN)," *Comput. Networks*, vol. 112, pp. 279–293, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2016.11.017

[14] D. Kreutz, F. M. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[15] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, "Software defined networking: State of the art and research challenges," *Comput. Networks*, vol. 72, pp. 74–98, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2014.07.004

[16] X. N. Nguyen, D. Saucez, C. Barakat, and T. Turletti, "Rules Placement Problem in OpenFlow Networks: A Survey," *IEEE Commun. Surv. Tutorials*, vol. 18, no. 2, pp. 1273–1286, 2016.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] "Open vSwitch – Production Quality, Multilayer Open Virtual Switch." [Online]. Available: https://www.openvswitch.org/

[24] "Tcpreplay: Sample Captures." [Online]. Available: https://tcpreplay.appneta.com/wiki/captures.html (accessed 2022-11-13).

[25] "Tcpreplay: Pcap Editing and Replaying Utilities." [Online]. Available: https://tcpreplay.appneta.com/ (accessed 2022-11-13).

[26] "Tcpreplay: bigFlows.pcap." [Online]. Available: https://s3.amazonaws.com/tcpreplay-pcap-files/bigFlows.pcap (accessed 2022-11-13).

[27] "Scapy: Packet crafting for Python2 and Python3." [Online]. Available: https://scapy.net/ (accessed 2022-11-13).

[28] "AWS IP Address Ranges." [Online]. Available: https://ip-ranges.amazonaws.com/ip-ranges.json (accessed 2022-11-14).

[29] "Azure IP Ranges and Service Tags – Public Cloud." [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=56519 (accessed 2022-11-14).

[30] "Google Cloud IP Addresses." [Online]. Available: https://www.gstatic.com/ipranges/cloud.json (accessed 2022-11-14).

[31] "Wireshark – Network Protocol and Packet Analyzer." [Online]. Available: https://www.wireshark.org/ (accessed 2023-01-21).