P4-based Hitless FaaS Load Balancer for Packet-Optical Network Edge Continuum

István Pelle,[1,2,3,*] Francesco Paolucci,[4,*] Balázs Sonkoly[1,2,3] and Filippo Cugini[4]

[1] *HSNLab, TMIT, VIK, BME, Budapest, Hungary,* [2] *MTA-BME Network Softwarization Research Group, Budapest, Hungary,* [3] *ELKH-BME Cloud Applications Research Group, Budapest, Hungary,* [4] *CNIT, Pisa, Italy*

[*] *pelle.istvan@vik.bme.hu, francesco.paolucci@cnit.it*

Abstract: P4 and novel node telemetry are leveraged to provide load balancing of ultra-low latency serverless application to multiple edges. Handling the overload of one edge without observable change in application delay is demonstrated.

1. Introduction

Recent advances fueled by the deployment of 5G and research of beyond 5G technologies clearly identify edge computing as a balancing point between the low latency provided by onboard computations on end-user equipment and the virtually unlimited compute power of the cloud. In the last five years, serverless computing has attracted significant research interest in the edge domain [1]. Serverless and its main implementation, Function as a Service (FaaS), offer fast and efficient deployment and scaling for short-lived stateless components, dubbed functions, running in containerized environments. In converged fronthaul and backhaul packet-optical networks Software Defined Networking (SDN) concepts have shown up to facilitate more flexible network traffic steering. Here, P4 has emerged as one of the most prominent technologies [2, 3]. Recent works have shown that combining FaaS with P4 programmable networks enables fast service re-deployment in overloaded 5G optical networks [4, 5].

In this work, we deepen the interaction between these two technologies to enhance application performance when deployed on edge resources that approach full utilization. Thus, here we present a three-fold contribution. First, we introduce a novel solution enabling effective interaction between FaaS deployment and application components and a P4 load balancer that can spread computation tasks to multiple edge nodes. Second, we give the details on our P4 implementation supporting two different load balancing methods. One of these takes decisions based on online telemetry of edge node CPU utilization, as it highly affects application performance. Third, we evaluate our implementations using a remote vehicle control application (RVCA) in a programmable packet-optical edge environment. We demonstrate that using our best load balancing method, it is possible to steer traffic dynamically to different edge nodes based on their actual CPU load while the application can successfully operate *without* observable latency increase even when one of its edge nodes reaches overload.

2. Serverless Application Deployment and P4-Driven Load Balancing

To realize load balancing, we tie three crucial concepts together. *i*) We use stateless on-demand FaaS functions that are instantiated on multiple edge nodes and can serve requests on any of those. *ii*) We monitor the load on the nodes and *iii*) feed it to a P4 programmable switch that can interpret the metrics and alter its behavior based on them. The top part of Fig. 1 shows a more detailed view on how we combine these. In step 1, we use the *edge infrastructure* description to set up our application *layout, resource and placement* specifications (LRPS) and our novel *P4 Load Balancer* (P4LB). We use layouts where specific application components are deployed to multiple edge nodes among which the P4LB distributes the traffic. In step 2 of Fig. 1, we submit the *application's structure and code* to our *Serverless Deployment Engine* (SDE) that transforms the application into deployable serverless artifacts based on the LRPS. The SDE merges the specified $\mathbf{f}$ application components into $F = W(\mathbf{f})$ FaaS functions (multiple fs can be part of a single F) and adds a special W wrapper. When deployed, W will handle all interactions between fs, datastore access and monitor these actions (capturing latency, rate and transferred data sizes). The SDE adds special *Node Telemetry* functions (NTF) that capture node CPU and memory load and provide those for the P4LB. In step 3, the SDE deploys these artifacts to the given edge (or cloud) infrastructure via a *Provider API* that gives low-level access to infrastructure components. We use Amazon Web Services (AWS) as a provider and leverage its IoT Greengrass v1 service at the edge. After deployment, W works as an extension to the default Greengrass runtime and provides direct connection between edge nodes using UDP. (The default AWS mechanism does not offer direct communication channels between nodes.) The NTFs periodically send load metrics with a rate of $3/\mathrm{s}$ to specific layer 2 and 3 addresses using novel Edge Resource Monitoring (ERM) messages. As shown in Fig. 2, the 4 bytes-long ERM extra-header runs over UDP and includes the protocol version, the sender edge node identifier (i.e., Edge ID) and the list of monitored metrics with their actual values. In this work we convey the

edge CPU load and consumed memory as monitored metrics, both expressed in percentage, however, we consider only the CPU load information for traffic steering operation. The P4LB parses the ERM messages and can alter application traffic steering based on the carried metrics.

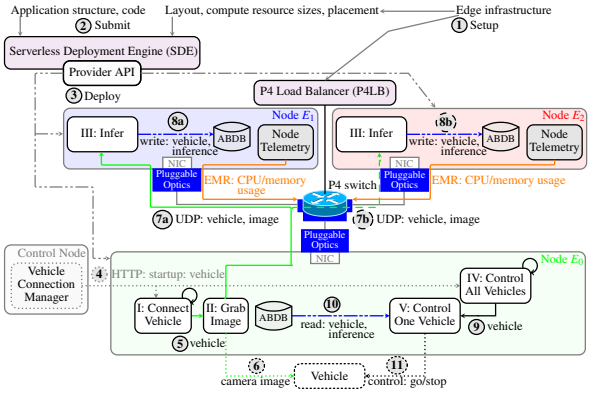

Fig. 1. Application deployment to testbed infrastructure

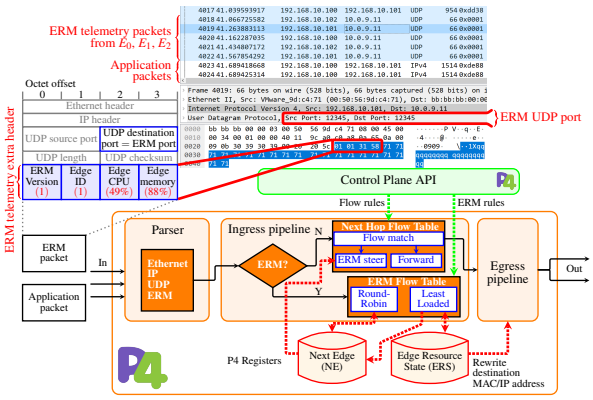

Fig. 2. ERM and P4LB operation schematics

As depicted in Fig. 2, the P4LB includes a parser, two pipelines and two main P4 registers. The *Parser* includes the custom ERM header detection. ERM messages are processed by the *ERM Flow Table* with the aim of updating the state of the sending edge node in terms of current CPU load, stored in the *Edge Resource State* (ERS) register and computing the best destination edge node to be written in the *Next Edge* (NE) register. Two algorithm actions are implemented in this work. The *Round-Robin* action reads the NE and switches its value to the value of the alternative edge node. The *Least Loaded* (LL) action reads the ERS, compares the current CPU loads, computes the least loaded edge node and stores it in the NE. Application traffic is submitted to the *Next Hop Flow Table*, with the possibility to perform either *ERM steer*ing or standard SDN forwarding (*forward* action using control plane API flow rules). In the former case, the assigned output port is read from the NE register and applied to the packet. Moreover, its MAC and IP destination addresses are updated according to the selected edge node. The P4LB design enables three main balancer options: *i*) normal forwarding (NF), using the forward action and ignoring ERM; *ii*) round-robin (RR), using ERM steering and the Round-Robin action, with the result of equally splitting traffic between the two edge nodes; *iii*) CPU load-based (CB), using ERM steering and the Least Loaded action, with the result to steer the traffic to the currently least loaded edge node.

3. Experimental Testbed and Results

We deploy our RVCA to four edge nodes, as depicted in Fig. 1. The RVCA is controlled from the *Control Node* that supplies data of multiple v vehicles (step 4). The deployment contains a detection (DL) and a control loop (CL) using 5 FaaS functions. In the DL (steps 5–8), *Connect Vehicle* picks a v_i from which *Grab Image* collects a camera image and *Infer* analyzes it. In the CL (steps 9–11), *Control All Vehicles* takes a v_j and *Control One Vehicle* (COV) interprets the result of the corresponding inference and sends a control message to v_j. The DL is triggered with a rate of 12/s and the CL 10/s. To avoid the cold start phenomenon of the FaaS functions, our W wrapper warms up 4 instances of each on-demand function that is sufficient for handling the given trigger rates (based on benchmark function execution delays). Infer, the most compute-intensive function, is deployed to nodes E_1 and E_2 while the rest of the RVCA and vehicle emulation are deployed to E_0. These three nodes also host an AnnaBellaDB (ABDB) [6] cluster used by the Infer and the COV functions to share data. ABDB is an in-memory key-value store that provides access to a key from all nodes but stores it only on the node where it is most frequently used (to reduce mean access delay). The Control Node and E_0 are physical machines. E_1 and E_2 are virtual machines (8 Intel Xeon E5-2650 v3 vCPUs, 6 GB RAM) that run Greengrass v1.11.3 and are initialized with an $\sim$37% CPU load. All nodes are connected via an optical fronthaul through the P4 BMv2 software switch executing the P4LB on a server with 6 Intel Xeon E5-2620 (2.10 GHz) CPU cores, 16 GB of RAM and three Gigabit Ethernet Network Interface Cards. The P4LB determines which node receives the image of a certain v_i (steps 7a/b). Each testbed device runs Ubuntu 18.04.

Fig. 3 shows RVCA DL delay (DLD, latency of producing inference for an image) under varying CPU load and different forwarding behaviors. In the first $\sim$560 s of the test, the P4LB executes the NF balancer option where the application determines which node to use for the Infer function. Under base load (BL) mean DLD is 53 ms. At the *NF3* point (see top of Fig. 3), we add an extra $\sim$30% partial load (PL) on E_1 by an external application which increases DLD but does not overwhelm the node. We increase this load to a level that overloads (OL) E_1 at *NF4* which causes mean DLD to rise to 193 ms. Forcing the application to E_2 at *NF5* reduces DLD to high-normal level (62 ms) which is caused by a slight difference between E_1 and E_2. Between 580–1000 s in our test in Fig. 3,

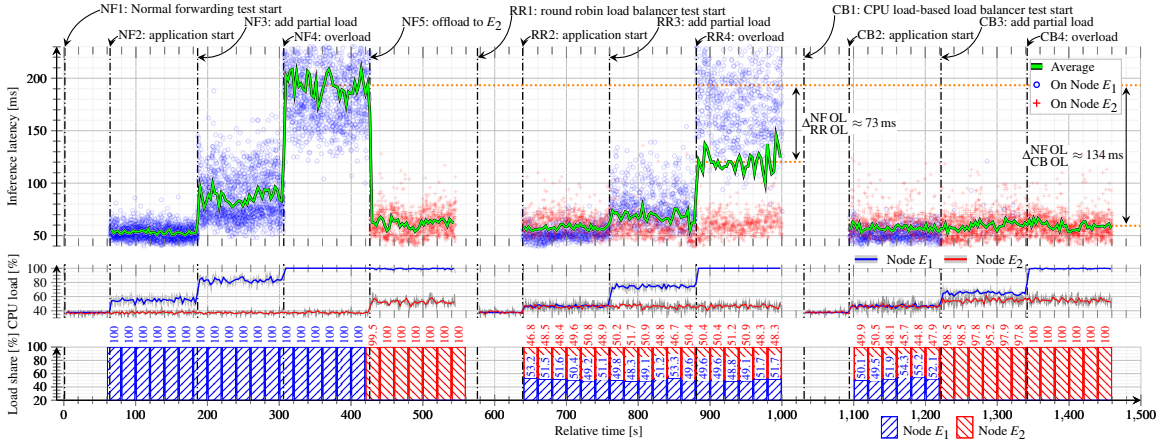

Fig. 3. Experimental results. Top: latency of the Infer function capped at 230 ms (3 s average: green; individual executions on E_1: blue; on E_2: red). Middle: CPU load on E_{1-2} (1 s [gray] and 3 s [colored] averages). Bottom: ratio of Infer functions executed on E_1 (blue) and E_2 (red) in 20 s windows.

we can observe the RR balancer option. As expected, it always splits the load between E_1 and E_2 approximately equally (see Fig. 3 bottom). Under BL (between the *RR2* and *RR3* points), CPU load also reaches the same level on both nodes while DLD is 57 ms. In later phases (after *RR3*), the increase of the mean DLD is around half of what we see in the respective phase of the NF case. Under PL, we see only a 10 ms addition, while we achieve 73 ms lower DLD under OL. The CB balancer option acts similarly to RR under BL halving the load between the used nodes. Adding PL already produces a higher CPU load on E_2 than in RR's case. Looking at the mid chart of Fig. 3, we can see that occasionally the CPU load at E_1 dips below that of E_2 which gives reason for the P4LB to keep 1%–5% of the inference executions at E_1. Thanks to the severely reduced number of executions at E_1, the node receives a smaller load from the application than in the RR case which enables it to complete executions with a 15 ms lower delay on average (calculated on the individual executions displayed in the top chart of Fig. 3). When overloading E_1, all inferences are executed at E_2. CB manages to keep a low DLD: under BL it matches RR's 57 ms while it achieves 59 ms throughout the PL and OL phases, without observable increase between them.

The Wireshark capture of Fig. 2 (top) shows an excerpt of the packets processed by the P4LB with the CB option. Application packets (e.g., frame 4017) are first sent by E_0 (IP address 192.168.10.100) and steered to E_1 (101). Then, ERM telemetry packets are received from all the nodes, triggering ERS and NE register updates. The capture shows the details of the metric fields sent by E_1 reporting its current CPU load at 49% (frame 4019). In this case, traffic continues to be steered to E_1 due to a higher E_2 CPU load (see frames 4023–4024).

The latency introduced by the P4LB when application traffic is processed (i.e., intra-switch latency) is around 250 μs with NF and around 300 μs with RR and CB. Thus, the impact of ERM-based stateful steering is an approximately 50 μs increased latency with respect to plain forwarding, corresponding to the additional ESR register read operation performed by both the RR and LL actions.

4. Conclusion

In this work we presented a novel framework jointly exploiting serverless computing and P4 network programmability specifically designed for latency-sensitive 5G applications running at the edge. The framework successfully enables serverless deployments, on a per-packet basis, between two edges for load balancing (or reliability) purposes. Results show effective dynamic distribution of computational load with negligible increase of application latency with respect to static forwarding towards a single edge.

Acknowledgment. This work was supported by *I*) the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund through projects *i*) no. 135074 under the FK_20 funding scheme, *ii*) 2019-2.1.13-TÉT_IN-2020-00021 under the 2019-2.1.13-TÉT-IN funding scheme, *II*) the B5G-OPEN Project, funded by the European Union under grant agreement No 101016663.

References

1. G. A. S. Cassel, V. F. Rodrigues, R. da Rosa Righi, M. R. Bez, A. C. Nepomuceno, C. A. da Costa, "Serverless computing for Internet of Things: A systematic literature review," in *Future Gener. Comput. Syst.*, vol. 128, 2022.
2. F. Cugini et al., "Applications of P4-based Network Programmability in Optical Networks," in *OFC 2022*, 2022.
3. Y. Yan, A. F. Beldachi, R. Nejabati and D. Simeonidou, "P4-enabled Smart NIC: Enabling Sliceable and Service-Driven Optical Data Centres," *J. Lightw. Technol.*, vol. 38, n. 9, pp. 2688-2694, 2020.
4. I. Pelle, et al., "Fast Edge-to-Edge Serverless Migration in 5G Programmable Packet-Optical Networks," in *OFC*, 2021.
5. I. Pelle, et al., "Latency-Sensitive Edge/Cloud Serverless Dynamic Deployment Over Telemetry-Based Packet-Optical Network," in *IEEE J. Sel. Areas Commun.*, 2021.
6. M. Szalay, P. Matray, L. Toka, "AnnaBellaDB: Key-Value Store Made Cloud Native," in *CSCM 2020*, 2020, pp. 1–5.