Intent-based 5G UPF configuration via Kubernetes Operators in the Edge

Ákos Leiter, István Kispál, Attila Hegyi, Péter Fazekas, Nándor Galambosi, Péter Hegyi, Péter Kulics, József Bíró
email: {akos.leiter, istvan.kispal, attila.hegyi, peter.fazekas, nandor.galambosi, peter.hegyi, peter.kulics, jozsef.biro}@nokia-bell-labs.com
Nokia Bell Labs, Bókay János utca 36-42, 1083 Budapest, Hungary,

Abstract—The expected growing number of edge clouds in the telecommunication industry requires new types of configuration management approaches in order to deal with the increased complexity. The Kubernetes Operator pattern widely used for lifecycle management of cloud native applications could be also applied to network management and configuration. In this paper we present our approach on using Kubernetes Operators to automatically adapt the configuration of the edge-located User Plane Functions (UPFs) to intents coming from an Edge Application. The owner of the Edge Application does not need to deal with network-related configuration as the chain of Kubernetes Operators manage it. Furthermore, we also provide numerical results about the speed of automatic configuration.

Keywords—intent, Kubernetes, 5G, ULCL, local-breakout

I. INTRODUCTION

Intent-based networking (IBN) with its declarative description of requirements where only the desired state is requested has depicted new approaches for the networking industry. This is different from the existing approaches where detailed steps of execution are used for tasks. One of the benefits of IBN comes with its layered abstraction levels which provides interfaces to business and low-level technical solutions as well. This is important for application developers too who require specific network settings for their services: they do not have to deal with network configuration at all. Application developers only need to request a specific network service (via intent) and the rest is taken care by the network operator.

In this paper, we advocate breaking down the end-to-end automation problem into ever smaller and smaller automation problems, that are solved by small control loops, called Controllers. This approach is depicted in Figure 1. Each controller receives intents via its Northbound Interface (NBI), and compares the desired state specified in the intent with the actual state of the world in a closed loop. If the two differ, then the controller tries to push the actual state toward the desired state. In our vision most of the controllers only break down their NB intent to lower-level intents, that are handled/realized by their own lower-level control loops. At the end, typically only the lowest-level controllers act on real-world objects (i.e. Physical Network Functions – PNFs, Containerized Network Functions - CNFs, switch/application configurations, cloud resources). Arbitrary numbers of intent processing layers can be introduced based on the location, execution targets etc. of services. The layered approach also implies that every orchestration problem is handled at the lowest possible layer, in other words, only those problems are delegated upwards that cannot be solved at the current layer. Kubernetes' architecture also follows similar principles. It is based on multiple cooperating control loops (i.e., various resource controllers, the Kubernetes scheduler, kubelets, kubeproxies, custom Kubernetes Operators) that are driven by intents called Kubernetes Resources.

Interestingly the concept of independently cooperating control loops is also analogous to how telco network operations used to work in most Communication Service Providers (CSP). The only catch is that the control loops were implemented by human operators (as opposed to Kubernetes Operators). Different groups of operators/engineers were specialized on continuously configuring different parts of the infrastructure, and if they couldn't solve an issue, they delegated it to the upper layer of engineers.

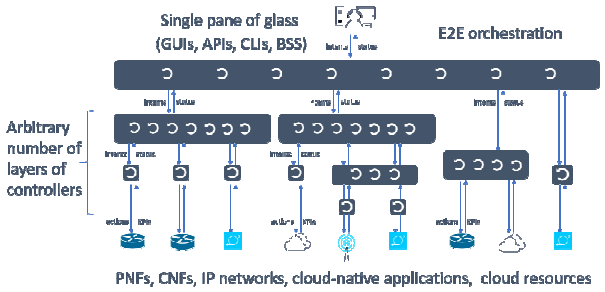

Figure 1 – Intent-based distributed orchestration vision

In this paper, we present how an Edge Application can request local-breakout (as a service) in the Edge without the need of understanding the networking in that particular site. Note that, in our case we use the phrase "local-breakout" as utilizing 3GPP Uplink Classifier (ULCL) [1] functionality for directing traffic locally at the edge, not in the context of roaming. Furthermore, we present numerical results to have an insight on the speed of such a (Day-2) configuration of edge UPF after a particular Kubernetes Service requests local-breakout.

The remaining sections are organized as follows: Section II presents related works. Our edge architecture is shown in Section III. Measurement results are elaborated in Section IV. Conclusion and Future work are placed in Sections V and VI respectively.

II. RELATED WORKS

The detailed description of Kubernetes Operators [2] and Operator SDK [3] what we used for our implementation can be found in the mentioned references. But scientific papers also investigate their usage. *Ruxiao Duan et al.* [4] present a maturity-level proposal for Kubernetes Operators. This rather pertains for application of lifecycle-management operators. In our case, we have a broader scope of usage for Kubernetes

Operators. It is worth mentioning, that Kubernetes Operators can be used for machine learning applications, *Ali Kanso et al.* [5] presents their KubeRay, a Kubernetes Operator to create Ray clusters. Kubernetes may be needed to be redesigned to fit for Edge Application. *Andrew Jeffery et al.*[6] investigate the bottleneck of Kubernetes, especially etcd in Edge use cases. The usage of Kubernetes Operators has been spreading not just in IT, but in telecommunication industry too; e.g.: *Osama Arouk et al.* [7] show a demo about RAN element deployment by Kubernetes Operators.

III. ARCHITECTURE

A. Edge Architecture

Edge clouds should have a well-defined architecture from both software and networking point of view. Figure 2 depicts our vision about a replicable edge stack. According to this vision every workload (platform service, application or network function) running in an edge cloud is hosted in a Kubernetes cluster. The diagram depicts dedicated orchestration functions for the hardware layer (bare metal provisioning) and for the lifecycle management of the Kubernetes clusters (K8s LCM), but these are less interesting for the topic of this paper. For the purposes of this paper the orchestration of workloads atop the Kubernetes clusters were split into low-level and high-level orchestration parts. The primary goal of low-level orchestration in this sense is to disseminate Kubernetes resources across multiple Kubernetes clusters in multiple edge sites. Examples for low-level orchestration toolsets include ArgoCD [8], FluxCD [9], Redhat ACM [10], etc. High-level orchestration includes end-to-end orchestration of telco networks, slice management functions, automatic placement of software components, e.g. ONAP [11] or vendor-specific products.

There are two basic approaches for cloud resources ownership depicted in the Figure 2 (although practical buildouts and offers may have other options): 1) local, self-managed/self-owned compute, network and storage resources 2) Hyperscalers' edge stack offerings [12] [13] [14]. Platform services provide various APIs for the workload running in the edge cloud, e.g.: service mesh and HW acceleration (GPU, SRIOV etc.). The layered edge stack must contain the basic networking, development and telecommunication services.

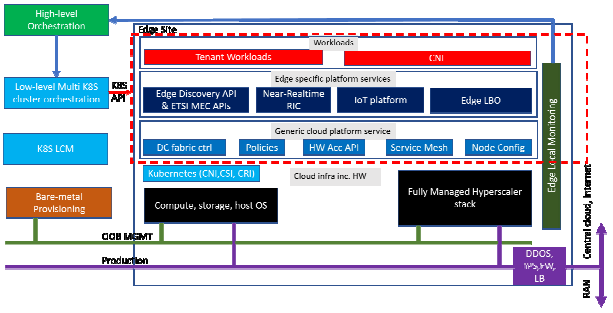

Figure 2 – Our vision on services in edge context

From telecommunication point of view, the presence of UPF is likely but here is the place also for O-RAN services like near-Real Time RAN Intelligent Controller (RIC). On the top of all the above-mentioned items, the real tenant workloads can be placed, but not necessarily using all of platform services. They can be third-party applications (e.g.: remote gaming servers, CDN caches) or additional telecommunication functions (e.g.: vRAN or analytics network elements).

B. 3GPP System view

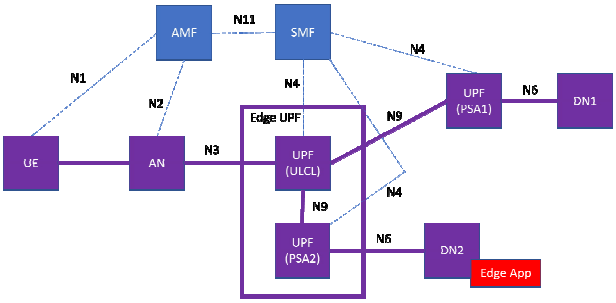

Figure 3 - Uplink Classifier architectural view with Edge Computing

Figure 3 presents how the Uplink Classifier-based approach of edge local-breakout works according to 3GPP[1]. A new edge-UPF is injected between the UE and the central UPF (where the main PDU Session Anchor (PSA1) resides). This edge UPF contains the so-called Uplink Classifier (UL CL). According to configured policies, the UL CL detects which packets of the traffic flowing in the PDU session should be terminated in a secondary PSA2 in the edge UPF and which packets should be forwarded to PSA1. PSA2 has the connection to a local data network (DN2) through the edge UPF's N6 interface. Edge Applications are available via DN2.

C. Kubernetes view

Overall, as mentioned above, that orchestration architecture is based on a hierarchy of reconciliation loops (typically Kubernetes Operators) built on top of each other. The reconciliation loops together form a dynamic system that is continuously pushing the state of the actual world toward a desired state defined in the highest-level intents.

1) Lifecycle operator

For the UPF lifecycle management we have implemented a Kubernetes controller, that handles the UPF Custom Resources [15]; we use this custom resource definition to describe a UPF instance. The controller is a Helm operator, generated by the Operator SDK [3] and is responsible for basic lifecycle management tasks of the UPF, like install and uninstall. The Lifecyle Operator uses the *quay.io/operator-framework/helm-operator* Docker image. The Lifecyle Operator continuously watches the UPF Custom Resource's state to validate whether the CR is in the desired state or not.

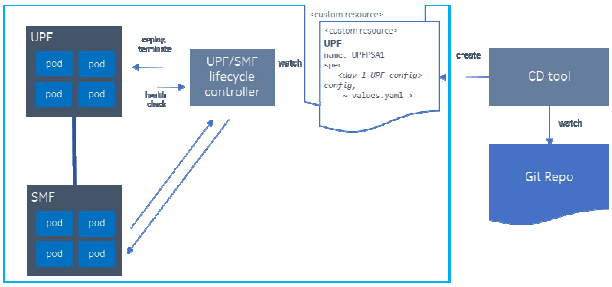

Figure 4 - Lifecycle operator architectural view for installing UPF

Our approach follows GitOps principles. The UPF custom resources are primarily stored in a Git repository, as YAML manifest files. Any changes in those files are automatically detected by a GitOps CD tool (in our case ArgoCD) and the manifests are automatically synched to the target Kubernetes clusters, and that in turn triggers the Lifecycle Operator described above to install/delete the UPF instance. This whole procedure is depicted in Figure 4, while a fraction of a UPF Custom Resource manifest can be found in Code 1. This model allows not just the setting of Kubernetes-related parameters e.g.: NodePort, etc., but it ensures UPF-level configuration too (DNN, PLNM etc.).

Code 1: Example of UPF CR

```
apiVersion: mco.bl.nokia.com/v1alpha1
kind: UPF
metadata:
    name: upf-psa1-operated
spec:
    image:
      repository: registry-test.net/5g
      name: upf
      tag: A-1.0
    service:
      oam:
        telnet:
          nodePort: 30023
          port: 2323
          targetPort: 2323
[…]
    plmn:
    - mcc: "999"
    - mnc: "99"
```

2) Edge Local Breakout Controller

Edge Local Breakout Controller (LBO Controller) is an operator too, depicted in Figure 5. LBO controller has two CRs: LBO Claim and LBO Config. LBO claim is the high-level intent for doing (Day 2) configuration. LBO Claims (Code 2) are responsible for defining the IP address ranges for local-breakout (IP packets with destination address in this range should be sent to the local N6 interface of UPF). If something is changed in the LBO Claims, then it will trigger automatic update to the system via the LBO Controller. LBO Config is just a practical configuration specific CRD containing information such as the SMF connection credentials, or the policies which has to be updated. LBO controller is the main entity which is responsible for hiding vendor specific configuration methods. Meanwhile, the LBO Claim is abstract enough to contain the only needed local-breakout parameters.

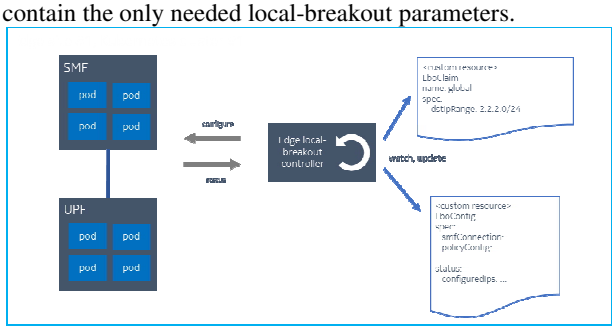

Figure 5 - Edge Local Breakout Controller with its CRs

Of course, these are implementation specific, standardization may be needed with LBO Claims-like objects on the long run. Even though the UL CL sits in the UPF, the configuration is done via SMF configuration and SMF pushes the configuration update to UPF via N4 interface for particular PDU sessions

Code 2: Example of LBO Claim

```
apiVersion: mco.bl.nokia.com/v1beta1
kind: LboClaim
metadata:
  name: lboclaim-1
spec:
  dstIpRange: 10.0.0.1/32
```

3) External Edge Service Controller

We also consider edge applications on Kubernetes bases. Figure 6 shows how edge applications fit into the existing picture. So far, the IP address range was static in the LBO Claim, in this section we present how it can be dynamic.: This is needed, because in practical cases the edge application's IP address will only be assigned after it's deployment and is not known in advance. They should only annotate their service – in this case a Load-balancer Service – with "external-edge-service=yes" annotation. This is watched by a new operator called External Edge Service Controller. (Edge) Applications usually need IP address for external reachability.

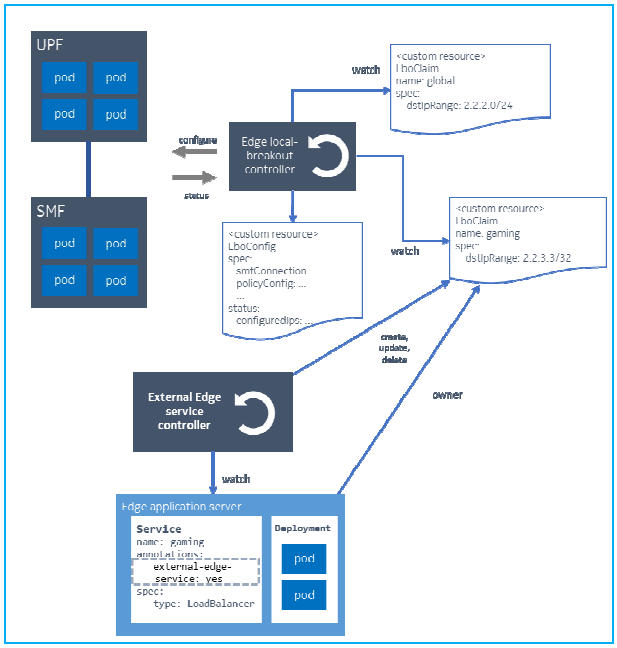

Figure 6 - Exposing Kubernetes Services via the Edge UPF

This IP address - which is an externally reachable IP address - is different from the default Cluster IP and can be configured by external Service types like the LoadBalancer. Thus, there should be another Kubernetes entity for external IP address assignment, in our case, it is MetalLB[16]. The external service controller then gets the assigned external IP and creates an LBO Claim from it. The claim is then being processed by the previous controller.

IV. MEASUREMENT

We have designed a measurement scenario to conclude how much time it takes for an application's external IP address to be configured for local-breakout in the edge-located UPF (depicted in Figure 7). The measurement starts with creating a Kubernetes Service for the particular fraction

of Pods. Later, external IP address is assigned by MetalLB. Then, the External Edge Service Controller recognizes that particular Kubernetes Service annotation which tells that, the Service needs external reachability. At the end of the configuration loop through the chain of the previously mentioned Kubernetes Operators, the UPF is configured with the actual local-breakout configuration for that particular Kubernetes Service.

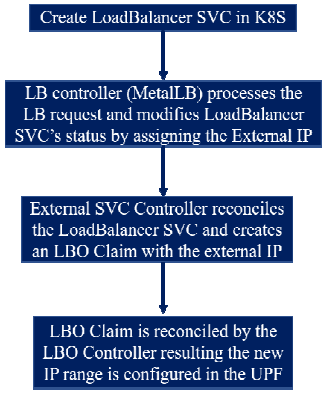

Figure 7 - Process of LBO creation

Meanwhile, the measurement framework watches continuously if the configuration change is applied in the UPF. If yes, then it concludes the measurements. Figure 8 presents the box plot of 1000 times of measurements while Table 1 shows the results numerically too. Note that, there are some uncertainties in the measurement conclusion: new configuration assignment can be even lower but the test framework needs time to download the UPF configuration and process it. Furthermore, IP address assignment by MetalLB also adds uncertainty to the system. The underlying hardware is Nokia OpenEdge platform.

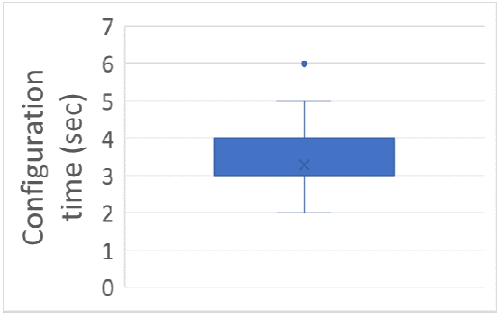

Figure 8 - Box plot of configuration time measurements

MIN	AVG	MED	MAX	STDEV
2	3.283	3	6	0.51

Table 1. – Numerical results of measurements in sec

V. Conclusion

In this paper, we presented how application developers can request network services without knowing the underlying telecommunication infrastructure. We have shown how a chain of Kubernetes Operators can hide and automate the whole process. The measurement results show that it takes only a few seconds to configure the local UPF instance according to the needs of the operations engineers. It is very hard to compare this approach to other similar ones, but according to GSMA[17], the installation and configuration of a new PNF takes time in the range of days. In this paper we showed that, the reconfiguration is within seconds which we believe clearly shows the power of network automation and Kubernetes Operators. Also, the usage of CRDs and Kubernetes controllers for configuration fits pretty well into the GitOps and Configuration-as-Code principles, which is essential for next generation network automation. Overall, we argue for the benefits of using Kubernetes APIs for configuring NFs. GitOps also adds an additional layer of security to the system which – we believe – is a big advantage. A user who has rights to change codes in Git, does not need to have direct access for a particular network function (e.g. no direct ssh). Furthermore, this is the place where revalidation can also be taken place: policies, hooks etc. can validate the config changes before they are actually pushed to the system. Based on our measurements it is also clear, that if a real-time control loop is required to solve a problem (e.g. various SDN or SON use cases), then those real-time control loops must be in the lowest-layer, since delegating intents to lower layers takes time.

VI. Future Work

It is worth investigating how a complete 5G network can be deployed and maintained by Kubernetes Operators. It should pertain for lifecycle and configuration management as well. Prevalidation hooks for security are also on the table in this research area.

References

[1] '3GPP TS 23.501: System architecture for the 5G System (5GS)'.

[2] 'Kubernetes Operators'. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (accessed Jan. 15, 2022).

[3] 'Operator SDK'. https://sdk.operatorframework.io/docs/building-operators/helm/tutorial/ (accessed Apr. 15, 2022).

[4] R. Duan, F. Zhang, and S. U. Khan, 'A Case Study on Five Maturity Levels of A Kubernetes Operator', in *2021 IEEE Cloud Summit (Cloud Summit)*, 2021, pp. 1–6. doi: 10.1109/IEEECloudSummit52029.2021.00008.

[5] A. Kanso *et al.*, 'Designing a Kubernetes Operator for Machine Learning Applications', in *Proceedings of the Seventh International Workshop on Container Technologies and Container Clouds*, New York, NY, USA, 2021, pp. 7–12. doi: 10.1145/3493649.3493654.

[6] A. Jeffery, H. Howard, and R. Mortier, 'Rearchitecting Kubernetes for the Edge', in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, New York, NY, USA, 2021, pp. 7–12. doi: 10.1145/3434770.3459730.

[7] O. Arouk and N. Nikaein, '5G Cloud-Native: Network Management & Automation', in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–2. doi: 10.1109/NOMS47738.2020.9110392.

[8] 'Open Network Automation Platform (ONAP)'. https://www.onap.org/ (accessed Jan. 10, 2022).

[9] 'ArgoCD'. https://argoproj.github.io/cd/

[10] 'FluxCD'. https://fluxcd.io/

[11] *Red Hat Advanced Cluster Management*. Accessed: Apr. 15, 2022. [Online]. Available: https://www.redhat.com/en/technologies/management/advanced-cluster-management

[12] 'Amazon Wavelenght'. https://aws.amazon.com/de/wavelength/ (accessed Apr. 15, 2022).

[13] 'Google Distributed Cloud'. https://cloud.google.com/distributed-cloud (accessed Apr. 15, 2022).

[14] 'Azure Edge Stack'. https://azure.microsoft.com/en-us/products/azure-stack/edge/#overview (accessed Apr. 15, 2022).

[15] 'Kubernetes - Custom Resource Definitions'. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ (accessed Apr. 15, 2022).

[16] 'MetalLB Kubernetes load-balancer'. https://metallb.universe.tf/ (accessed Apr. 15, 2022).

[17] GSMA, 'Migration from Physical to Virtual Network Functions: Best Practices and Lessons Learned'. https://www.gsma.com/futurenetworks/5g/migration-from-physical-to-virtual-network-functions-best-practices-and-lessons-learned/ (accessed Jan. 15, 2022).