Implementing a Machine Learning Function Orchestration

Axel Wassington[1], Luis Velasco[1]*, Lluis Gifre[2], and Marc Ruiz[1]

[1] Optical Communications Group, Universitat Politècnica de Catalunya, Spain, *luis.velasco@upc.edu
[2] Nokia Bell Labs, Nozay, France

Abstract Deployment of Machine Learning (ML) applications require from an Orchestrator to create ML functions that are connected as ML pipelines. Orchestrator implementation and demonstration for the deployment and reconfiguration of a ML pipeline related to a lightpath is shown.

Introduction

Software Defined Networking (SDN) defines a centralized control plane architecture with global network vision, which, at the optical layer, allows the SDN controller to find the optimal routing for optical connections (lightpath) at provisioning time and during reconfiguration [1]. Besides, a distributed computing and storage infrastructure has been deployed for virtualizing network functions, which is managed by a centralized Virtual Infrastructure Orchestrator (VIO) [2].

On the other hand, the rise of Machine Learning (ML) algorithms for optical network automation [3] entails analyzing heterogeneous monitoring data collected from monitoring points in network devices. Because network entities can be reconfigured, e.g., lightpath rerouting, it is of paramount importance to link different ML functions (i.e., performance data collection, pre-processing, analysis, storage, visualization, etc.) among them to create a *ML Pipeline* and to the related network entity (e.g., a lightpath).

Based on the definition in [4], in this paper, we present the implementation of a ML Function Orchestrator (MLFO), as an independent orchestrator that manages ML pipelines by placing ML functions in different locations in the network and connecting them to create a chain.

ML Pipeline Management

Let us first focus on the ML Pipeline computation. Let us assume that an entity in the data plane (e.g., a lightpath) requires deploying a ML pipeline with five different ML functions: *i*) collector (Co) in charge of collecting monitoring data from activated monitoring points (M); *ii*) aggregator (Ag) that collects measurements from a number of different collectors and perform some not computationally intensive task, like compute some statistics, e.g., max, min, and average; *iii*) processor (Pr), which performs more computational intensive task on the received data; *iv*) a time series database (DB); and *v*) an user interface (UI). The relation among those ML functions can be specified as a graph $T=(N,A)$ (see Fig. 1a).

Given graph T, the description of the set of nodes N, the constraints for the set of arcs A, as well as some other constraints, like fixed ML function placements, an optimization problem can be defined to find the optimal solution that includes the placement of the ML functions to create a ML pipeline that follows template T and meet the constraints, while minimizing some utility function (e.g., number and capacity of containers/VMs to be deployed, connectivity cost, etc.) subject to the state of the resources in the telecom cloud infrastructure. We name such problem as *ML Pipeline Deployment* (PLD), and it can formally be stated as:

Given:

- *Pipeline Template*: graph $T = (V, A)$, where V = {<allowed types={*type*}>}, A = {<maxDelay, minCapacity>}, and *type*=<container/VM descriptor, max_children>.
- *Constrained nodes*: set N = {<n, F>}, where n = *type* $\in v$."allowed types" | $v \in V$, and F is a set of constraints, e.g., a location, and it can be empty.
- Telecom cloud infrastructure, i.e., edge/cloud computing and connectivity.

Output: The pipeline P to be deployed, i.e., $P = (N', E) \sim T$, $N' \supseteq N$ and every $e \in E$ is an instance of $a \in A$ that satisfies its constraints.

Objective: Minimize some utility function.

To solve the PLD problem, we have developed a heuristic algorithm that first places the constrained ML functions and finds the shortest paths connecting the already placed ML functions. During such process, more ML functions are added to meet the given constraints until graph P following template T is obtained. An example of solution is illustrated in Fig. 1b, where the location of the collector ML functions have been specified to be in the same location as the related monitoring point. Once the PLD problem is solved, the obtained solution is deployed (Fig. 2a).

Because the ML pipeline is linked to an entity in the data plane, when that entity is reconfigured, its ML pipeline might need to be also reconfigured. Therefore, we can define a new optimization problem, where we are given the template T, the current ML pipeline P and the new set of constrained nodes and the objective is to reconfigure P with the minimum cost (e.g., number of changes in P' w.r.t. P and the total cost, etc.). We name such problem as *ML*

 DOI 10.1109/ECOC52684.2021.9605907

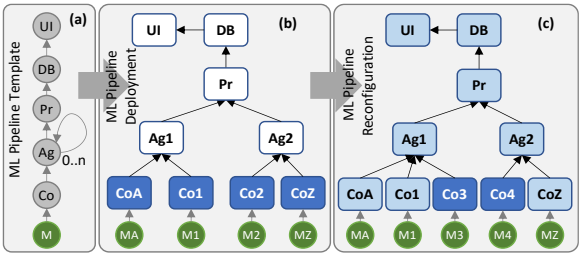


Fig. 1: ML pipeline management.

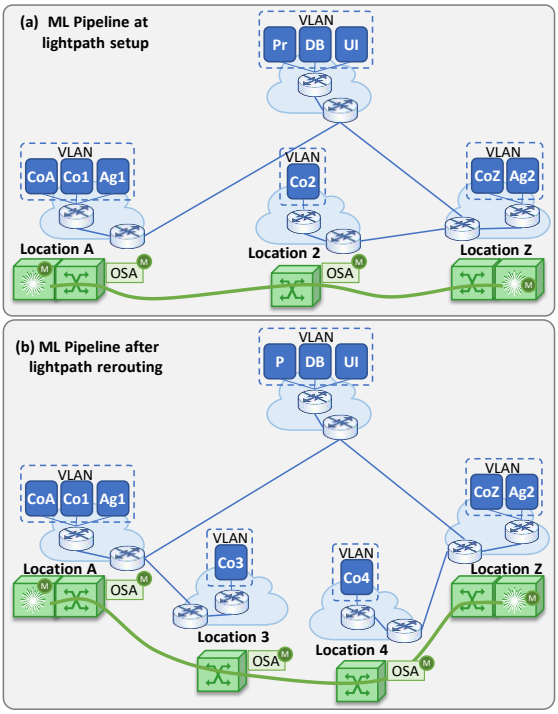


Fig. 2: Example of ML pipelines for a lightpath.

Pipeline Reconfiguration (PLR) and it can formally be stated as:

Given:

- The template *T,* the deployed pipeline *P* and the new set of constrained nodes *N*.
- Telecom cloud infrastructure, i.e., edge/cloud computing and connectivity.

Output: The new pipeline *P'* to be deployed.
Objective: Minimize some utility function.

We have developed a heuristic similar as the one for the PLD problem, where the deployed ML functions that are not in the new constrained set of nodes are first removed, and the nodes that need to be migrated are disconnected and moved. An example of reconfiguration is presented in Fig. 1c, where ML function Co2 is not needed for the new ML pipeline, and Co3 and Co4 have been added. The resulting reconfiguration is shown in Fig. 2b.

Proposed Architecture and Workflows

The proposed architecture is depicted in Fig. 3, where an external management application system triggers ML pipeline deployment. Therefore, the MLFO needs to expose a Northbound Interface (NBI) to receive the description of the ML pipeline, including the

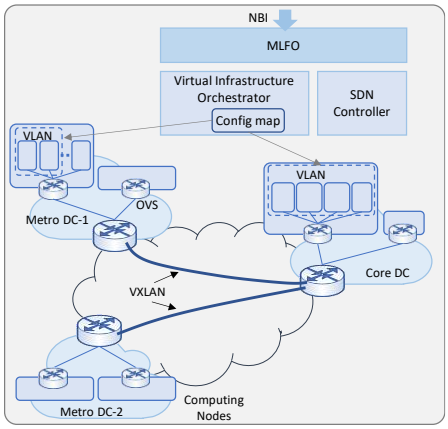


Fig. 3: Architecture of the solution.

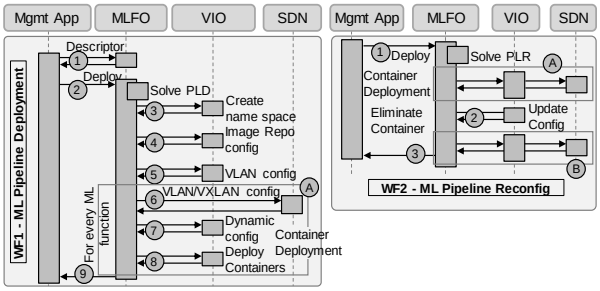
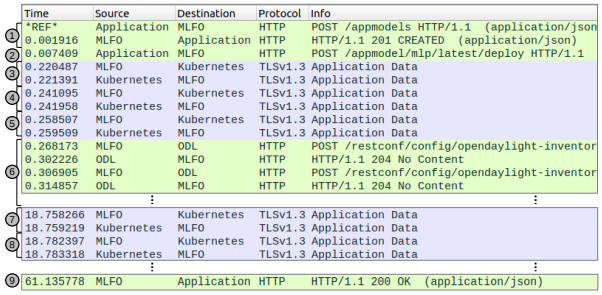


Fig. 4: Deployment and reconfiguration.

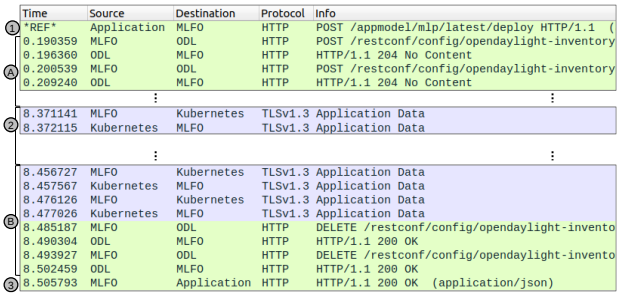
template, constrained nodes, etc. (collectively named ML pipeline *descriptor*). The MLFO runs the PLD and PLR optimization problems to obtain the ML pipeline to be deployed (*deployment plan*), and coordinates with the VIO and the packet layer SDN controller. A specific VLAN is created for each ML pipeline for intra-DC communications, whereas we assume that connectivity between two DCs is based on pre-established connections (e.g., VXLAN tunnels).

Fig. 4 presents the workflows for the initial ML pipeline deployment and for any subsequent externally-triggered reconfiguration. Let us start with the deployment workflow (WF1). The management application initiates WF1 by sending the deployment plan (message WF1/1 in Fig. 4). The descriptor contains a template for the ML functions and for the connectivity. Next, the deployment is triggered (2) and the MLFO starts a series of steps. First, the MLFO solves the PDL using the constraints, resulting in a mapping between the ML functions and the datacenters, and the connectivity and the deployment plan is computed. A list of iterations is generated that includes the communication of the MLFO with the VIO (e.g., Kubernetes) for the deployment of the ML functions (e.g., encapsulated into containers [5]), and with the SDN controller for managing the connectivity among the ML functions. The list iterations include: *i*) the namespace creation (3); *ii*) the configuration of an image repository storing the different computing images that are retrieved when a new ML function instance is deployed (4); *iii*) the configuration of the ML pipeline network that entails creating the VLAN (5) and pairing it



	Time	Source	Destination	Protocol	Info
1	*REF*	Application	MLFO	HTTP	POST /appmodels HTTP/1.1 (application/json
	0.001916	MLFO	Application	HTTP	HTTP/1.1 201 CREATED (application/json)
2	0.007409	Application	MLFO	HTTP	POST /appmodel/mlp/latest/deploy HTTP/1.1
3	0.220487	MLFO	Kubernetes	TLSv1.3	Application Data
	0.221391	Kubernetes	MLFO	TLSv1.3	Application Data
4	0.241095	MLFO	Kubernetes	TLSv1.3	Application Data
	0.241958	Kubernetes	MLFO	TLSv1.3	Application Data
5	0.258507	MLFO	Kubernetes	TLSv1.3	Application Data
	0.259509	Kubernetes	MLFO	TLSv1.3	Application Data
	0.268173	MLFO	ODL	HTTP	POST /restconf/config/opendaylight-inventor
	0.302226	ODL	MLFO	HTTP	HTTP/1.1 204 No Content
6	0.306905	MLFO	ODL	HTTP	POST /restconf/config/opendaylight-inventor
	0.314857	ODL	MLFO	HTTP	HTTP/1.1 204 No Content
	⋮				⋮
7	18.758266	MLFO	Kubernetes	TLSv1.3	Application Data
	18.759219	Kubernetes	MLFO	TLSv1.3	Application Data
8	18.782397	MLFO	Kubernetes	TLSv1.3	Application Data
	18.783318	Kubernetes	MLFO	TLSv1.3	Application Data
	⋮				⋮
9	61.135778	MLFO	Application	HTTP	HTTP/1.1 200 OK (application/json)

Fig. 5: Messages exchanged during WF1.



	Time	Source	Destination	Protocol	Info
1	*REF*	Application	MLFO	HTTP	POST /appmodel/mlp/latest/deploy HTTP/1.1 (
	0.190359	MLFO	ODL	HTTP	POST /restconf/config/opendaylight-inventory
	0.196360	ODL	MLFO	HTTP	HTTP/1.1 204 No Content
A	0.200539	MLFO	ODL	HTTP	POST /restconf/config/opendaylight-inventory
	0.209240	ODL	MLFO	HTTP	HTTP/1.1 204 No Content
	⋮				⋮
2	8.371141	MLFO	Kubernetes	TLSv1.3	Application Data
	8.372115	Kubernetes	MLFO	TLSv1.3	Application Data
	⋮				⋮
	8.456727	MLFO	Kubernetes	TLSv1.3	Application Data
	8.457567	Kubernetes	MLFO	TLSv1.3	Application Data
	8.476126	MLFO	Kubernetes	TLSv1.3	Application Data
	8.477026	Kubernetes	MLFO	TLSv1.3	Application Data
B	8.485187	MLFO	ODL	HTTP	DELETE /restconf/config/opendaylight-invento
	8.490304	ODL	MLFO	HTTP	HTTP/1.1 200 OK
	8.493927	MLFO	ODL	HTTP	DELETE /restconf/config/opendaylight-invento
	8.502459	ODL	MLFO	HTTP	HTTP/1.1 200 OK
3	8.505793	MLFO	Application	HTTP	HTTP/1.1 200 OK (application/json)

Fig. 7: Messages exchanged during WF2.

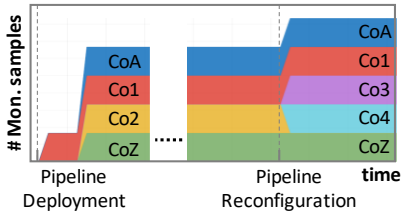


Fig. 8: #Monitoring samples vs time.

to the VXLAN tunnels (6); *iv*) the creation of a volume for the dynamic configuration of the computing instances (7); and *v*) the deployment of the containers (8). Steps 6-8 are followed for every ML function to be deployed (block A). A reply is eventually sent (9).

WF2 is triggered when the ML pipeline needs to be reconfigured. The management application initiates WF2 by sending a new set of constraints to the MLFO (message WF2/1 in Fig. 4). The PLR problem is then solved considering the received configuration. Then, the MLFO finds the changes to be performed and prepares a plan with the creation of new ML functions (block A) and removal of existing ones (block B). Besides, the dynamic configs are updated to reflect the changes in the system (e.g., changes on the IPs) (2). When all the steps are executed, the result is sent back to the management application (3).

Illustrative Results

We implemented the MLFO in Python 3.8 that exposes a REST API NBI. Kubernetes was used as VIO and docker as the container technology [6]. The MLFO uses the Kubernetes API through a python client library. A private image repository was hosted in Docker Hub. Multus and Open vSwitch (OVS) Container Network Interface (CNI) plugins were used for the VLAN configuration thorough Kubernetes. Kubernetes ConfigMap was used for the dynamic

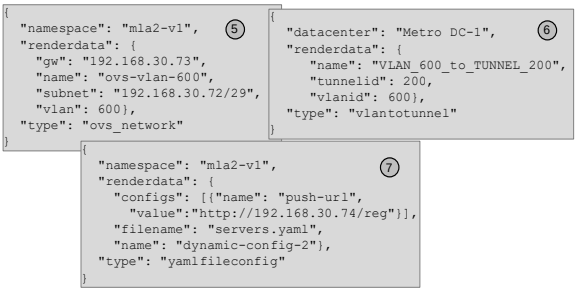


Fig. 6: Details of selected messages.

configuration files mounted into the containers as read-only files. The ingress-nginx controller was used for the reverse proxy; the configuration is done through the Kubernetes' ingress resource, which exposes a service through a load balancer (MetalLB). As for the SDN controller, we used OpenDayLight (ODL), which controls intra-DC switches through the OpenFlow protocol. VXLAN was used as tunneling technology for inter-DC tunnels. The MLFO pairs VLANs with VXLANs on each of the intra-DC switches that are involved on a connection between two containers.

Fig. 5 shows the messages exchanged during WF1 to deploy the ML pipeline in Fig. 2a; the number of the messages is that in Fig. 4 for the sake of clarity. The details of messages 5-7 are presented in Fig. 6. The included information is expanded using jinja2 templates to create the final message for the VIO or the SDN controller. The dynamic config (7) is performed by creating a Kubernetes *configmap* (a series of key-values, that are written to a file and mounted in the container filesystem) for the dynamic configuration of the computing instances. Computing instances might have also external endpoints that can be accessed through a reverse proxy. Containers' deployment is carried out through Kubernetes; information includes: the datacenter identifier, the VLAN and the IP address, the dynamic configuration name, the image, a name, and a list of configurations. Total deployment time was about 1 min.

Fig. 7 shows the exchanged messages during WF2 to reconfigure the ML pipeline as in Fig. 2b. Just to mention that WF2 follows a make-before-break approach, i.e., after the creation of new containers, the MLFO waits for the container to be available before continuing with the next steps to avoid losing monitoring samples. Total reconfiguration time was 8.5 sec.

Finally, Fig. 8 shows the number of measurements collected in the DB node of the ML pipeline to demonstrate ML pipeline deployment and its reconfiguration.

Acknowledgements

The research leading to these results has received funding from the Spanish MINECO TWINS project (TEC2017-90097-R) and from ICREA.

References

[1] L. Velasco et al., “In-Operation Network Planning,” IEEE Communications Magazine, vol. 52, pp. 52-60, 2014.

[2] M. Bonfim, K. Dias, and S. Fernandes, “Integrated NFV/SDN Architectures: A Systematic Literature Review,” ACM Computing Surveys, vol. 51, pp. 1-39, 2019.

[3] D. Rafique and L. Velasco, “Machine Learning for Optical Network Automation: Overview, Architecture and Applications,” (Invited Tutorial) IEEE/OSA Journal of Optical Communications and Networking (JOCN), vol. 10, pp. D126-D143, 2018.

[4] “Unified architecture for machine learning in 5G and future networks,” Focus group on Machine Learning for Future Networks including 5G, ITU-T, 2019.

[5] L. Toka, G. Dobreff, B. Fodor and B. Sonkoly, “Machine Learning-Based Scaling Management for Kubernetes Edge Clusters,” IEEE Transactions on Network and Service Management, vol. 18, pp. 958-972, 2021.

[6] D. Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes,” IEEE Cloud Computing, vol. 1, pp. 81-84, 2014.