Demonstrating Network Slice KPI Monitoring in a 5G Testbed

Niloy Saha*, Alexander James*, Nashid Shahriar†, Raouf Boutaba* and Aladdin Saleh‡
{n6saha, alexander.james, rboutaba}@uwaterloo.ca
{nashid.shahriar}@uregina.ca, {aladdin.saleh}@rci.rogers.com
*University of Waterloo, Canada, †University of Regina, Canada, ‡Rogers Communications, Canada Inc.

***Abstract*—Network slicing has been envisaged as a key enabler to satisfy diverse requirements of 5G networks, by creating multiple isolated end-to-end virtual networks dedicated to different services. An accurate view of these end-to-end 5G network slices is essential for both artificial intelligence (AI) driven slice orchestration, and data-driven automated service assurance. However, the existing open-source implementations of the 5G core do not natively support slice Key Performance Indicator (KPI) monitoring. In this demonstration, we show how to deploy a functional 5G testbed using a combination of open-source frameworks and tools, with a guide for configuring multiple network slices published on GitHub [1]. We also show the feasibility of monitoring and visualizing network slice KPIs using a representative cloud-gaming use-case.**

***Index Terms*—Network Slicing, KPI, monitoring, 5G Network**

I. Introduction

Network slicing, now an important part of Third Generation Partnership Project (3GPP) Release 16, is a core component of the 5G mobile network architecture. One of the motivating factors behind the introduction of slicing is the ability to support a set of applications with heterogeneous Quality of Service (QoS) requirements using the same physical infrastructure.

One of the most important problems preventing the wide adoption of network slicing is the need for orchestration platforms and algorithms that are up to the task of managing the myriad resources allocated to each slice deployed in the network, as well as handling the life-cycle and fault management operations for each slice. Over the past decade Machine Learning (ML), and more specifically Reinforcement Learning (RL), has been applied to many control tasks in a number of different domains. Now mobile network research is also shifting to focus on the use of RL for management and orchestration tasks.

The reliance on learning algorithms to orchestrate and manage the mobile network presupposes that there are large quantities of monitoring data available that accurately captures the relevant state of the mobile network in order to train learning agents to accomplish management and orchestration tasks. However, in surveying the available open-source implementations of the mobile core, we found that none of the current offerings support monitoring in any meaningful way.

Various 3GPP Technical Specificationss (TSs) define a number of slice specific KPIs, with the idea being that these KPIs can accurately capture the state of the mobile network, specifically the state of the constituent slices. Computing many of the 3GPP slice KPIs requires the collection of a number of performance metrics from distinct network functions in the mobile core, complicating the process of slice KPI collection and computation. As such, some work is required to develop a system that is capable of performing efficient and timely collection and computation of slice KPIs as they are defined in various 3GPP TSss.

In this demo, we showcase a system for slice KPI collection, computation, aggregation and visualization using a set of open-source components in a state-of-the-art mobile network testbed. We also demonstrate how this system can be used to collect and visualize slice KPIs in real-time for a cloud gaming application deployed as an edge service in the testbed.

II. System Description

This section provides an overview of the software and hardware components deployed in the testbed. Figure 1 depicts these components.

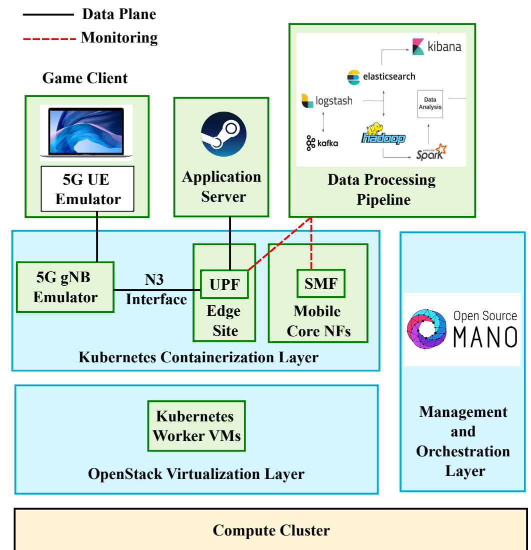

Fig. 1: System Description

Compute Cluster The foundation of the mobile network testbed is a compute cluster managed using OpenStack Ussuri [2]. The cluster consists of 6 compute servers, each with 16 GiB of RAM and 8 CPUs running at 3.3 GHz. Each of the compute servers is connected using 100 Base-T ethernet.

Containerization Layer All of the network functions used in this demonstration are deployed as containers in a Kubernetes cluster. The Kubernetes controller and worker nodes are deployed as virtual machines in the OpenStack cluster, allowing the lifecycle of the Kubernetes worker nodes to be orchestrated via the OpenStack virtualization layer.

Orchestration and Management Layer Deployment and lifecycle management of the mobile core is handled by Open Source MANO (OSM), an important open-source project that conforms to the European Telecommunications Standards Institute (ETSI) NFV architecture. OSM automates the management and orchestration of network services composed of any number of constituent Virtual Network Functions (VNFs). Each VNF is described using a Virtual Network Function Descriptor (VNFD), which is a YAML file specifying the base image for the VNF (either a VM or container image), as well as its dependencies, compute, storage and network requirements and a host of other customizable parameters. Network operators then create a Network Service Descriptor (NSD), which specifies a network topology composed of any number of VNFDs. Once the NSD has been onboarded, the network service can be deployed through either the OSM web or command line interface. OSM will then interface with the various Virtual Infrastructure Managers (VIMs) present in the system to instantiate the service described by the NSD.

Mobile Core The mobile core implementation used in this demonstration is Free5GC [3], an open-source implementation of 3GPP R15. Free5GC contains implementations of the AMF, AUSF, NRF, NSSF, PCF, SMF, UDM, UDR, and UPF. The Free5GC binaries incorporated in container images hosted on DockerHub and subsequently packaged as a Helm chart [4]. The use of Helm allows for straightforward deployment and dependency management for applications deployed over a complex, inter-dependant set of Kubernetes containers.

Radio Access Network Emulation The Radio Access Network (RAN) and User Equipments (UEs) are being emulated using UERANSIM [5]. UERANSIM is an open-source project that implements a 3GPP Release 16 compliant gNB and UE in software. To emulate the over-the-air interface, UERANSIM, uses the Radio Link Simulation (RLS) protocol, which replaces the over-the-air transmissions that typically take place in the RAN with a UDP based link between the UEs and the gNB.

Data Processing Pipeline The data processing pipeline is responsible for the ingestion, cleaning and indexing of monitoring data generated by the mobile network. FileBeat [6], an open-source project managed by Elastic [7], is used to ship the container log files generated by the Kubernetes containers that implement the mobile core to a Kafka queue [8]. Once the container logs have been pushed into the Kakfa queue, Logstash [9], another open-source project maintained by Elastic [7], retrieves the logs from the specified topic and parses them before pushing them into an Elasticsearch instance [10]. Once the logs have been parsed and documents containing the metrics have been ingested into elasticsearch, the performance metrics that compose the slice KPIs are used to compute the actual KPIs. At this point the slice KPIs can be accessed via the Elasticsearch APIs or visualized and analyzed using Kibana [11].

Application Server The application used to demonstrate 5G slice KPI monitoring is Steam [12], a computer gaming application that allows users to stream games from a game server to a remote client. For this demonstration the game server is hosted on a machine with dedicated graphics hardware located outside of the compute cluster that hosts the mobile core.

III. Network Slice KPI Monitoring

Monitoring network slice KPIs involves the extraction and subsequent correlation of performance metrics (PMs) from various network functions (NFs) in the 5G network. In this demonstration, we focus on a particular slice KPI — *average downlink throughput per network slice*. This involves the collection of PMs (such as packet or byte counts) from the user plane functions (UPFs) in the network. Specifically, as defined in 3GPP TS 28.552 (Section 5.4.1.3) [13], this requires the collection of *GTP data packets on N3 interface (from RAN to UPF)*. Moreover, since the UPF does not store slice related information due to the control and user plane separation in 5G, the information from the UPF needs to be correlated with slice information collected from the session management function (SMF). This is done as follows.

A network slice is uniquely identified by its Single Network Slice Selection Assistance Information (S-NSSAI) [14]. An S-NSSAI may have one or more protocol data unit (PDU) sessions associated with it, where a PDU session is an abstraction introduced in 5G to represent an end-to-end connection from the UE to a data network through the UPF. Thus, the average throughput per slice involves computing the sum of GTP-U packets/bytes for the PDU sessions associated with the slice. This information is associated with several packet detection rules (PDRs) at the UPFs. Moreover, the SMF uses the packet forwarding control protocol (PFCP) to manage sessions with the UPF per PDU session. Therefore, the steps involved in computing average throughput per network slice are as follows:

At the SMF: a) Find the mapping between an S-NSSAI and associated PDU sessions, b) identify individual PDU sessions using PDU session ID, c) find the mapping between PDU session and PFCP session, d) identify PFCP session(s) using its fully-qualified session identifier (F-SEID), and e) identify PDRs per PFCP session using PDR ID.

At the UPF: a) Use PDR ID at UPF to identify PDRs at the UPF and b) to extract UL/DL statistics per PDR. UL and DL statistics consist of packet counts for all of the PDRs associated with the PDU sessions in question.

These packet counts were obtained by modifying the `gtp5g` kernel module [15] that terminates the GPRS Tunneling Protocol (GTP) tunnels at the UPF. The kernel module was modified to maintain and increment packet counters for each of the PDRs currently installed at the UPF. We implemented a `netlink` [16] interface to allow these packet count statistics to be retrieved from the kernel module. Finally the UPF