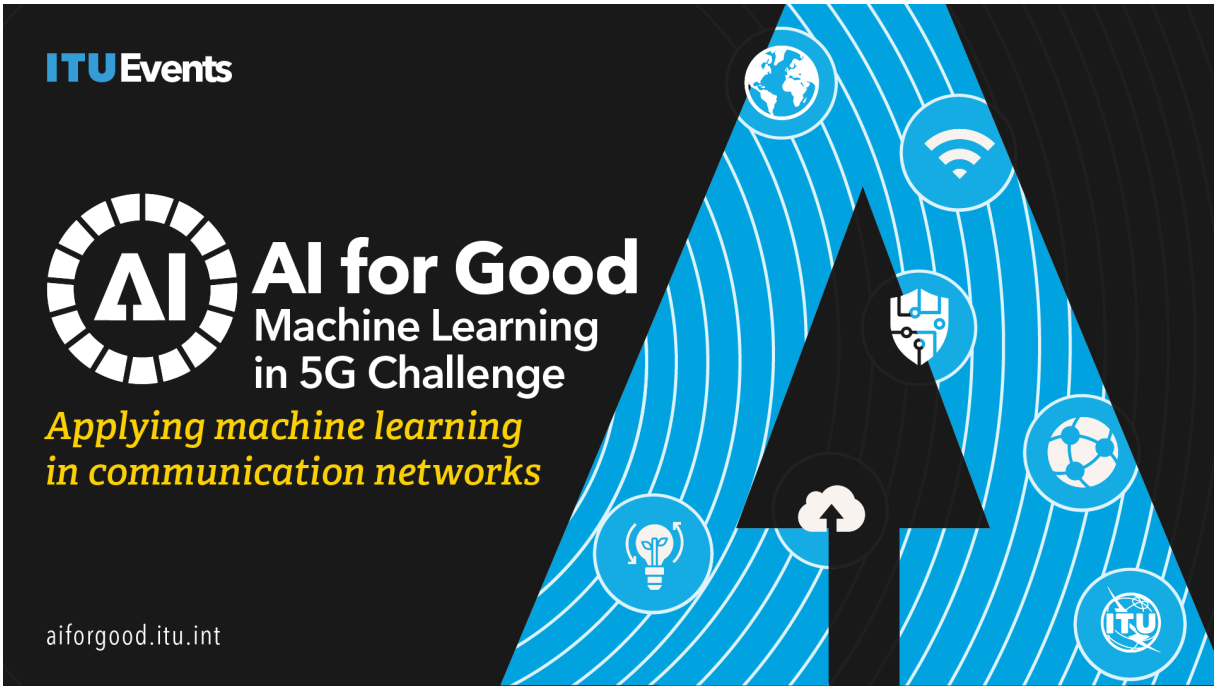

ITU-ML5G-PS-014: Build-a-thon(PoC) Network resource allocation for emergency management based on closed loop analysis

-Activity 4-

TEAM: AUTOMATO

MEHMET KARACA & DORUK TAYLI & ÖZGE SİMAY DEMİRCİ

Our implementation: https://github.com/krcmehmet/ITUChallenge_BuildaThon_Activity4.git

HOST: ITU Focus Group-Autonomous Networks (FG-AN)

DATE: 28 OCTOBER 2021

1. DEFINITIONS

- **Autonomous domain**[1]: An administrative management domain which defines the scope of delegated autonomous behavior.
- **Autonomous network**[1]: A system of networks and software platforms that are able to detect and adapt to network conditions with minimal input.
- **Closed loop** [1]: A type of control mechanism that observes and directs a range of managed entities with the goal of accomplishing a particular objective.
- **Closed loop automation** [1]: Automated processes based upon feedback from measuring, monitoring, and evaluating network traffic. It reduces or eliminates the involvement of humans from the system operation.
- **D2D sidelink** [1]: A5G link between two robots or user equipment (UE) which bypass the base station of 5G RAN.
- **Hierarchical closed loops** [1]: A set of closed loops organized in hierarchical form throughout the network domains to activate autonomous networks.
- **Higher loop** [1]: A closed loop that has an overview of the network which may be spread over a number of network domains.
- **Inference engine** [1]: The processing component that allows runtime environment, which is infrastructure of software and hardware that provides a specific codebase to run in real time, for a ML model and shows corresponding ML model inference capability.
- **Lower loop** [1]: A type of closed loop which has a limited view of the network domain.
- **ML model serving** [1]: Operation of arranging and distributing machine learning models in different deployment environments to provide the use of model inference to machine learning underlay networks.
- **ML sub-model**[1]: Element of a machine learning model which is separated into one or more parts by utilising a model splitting technique.
- **ML underlay network**[1]: A telecommunication network and its connected network operations, that interface with corresponding ML overlays.

- **Network levels** [1]: Diverse levels of the network where the reference points between these parts of the networks are described in the related network architecture specifications [e.g., cloud-fog-edge]. Also, every network level can possess particular abilities and functions.
- **Shared RAN** [1]: Network of RAN that is shared between private 5G users and public.

2. KEY CONCEPTS

Autonomous Closed Loop [2]:

• A closed loop is a control mechanism that monitors and arranges a group of controlled network elements in order to achieve a certain purpose. Autonomous closed loops can accomplish this without the need for external intervention apart from the input goal. These closed loops may be executed by using methods based on AI/ML models.

Orchestration [2]:

• The term "orchestration" refers to the management of physical or virtual network functions.

Closed Loop Orchestration [2]:

• Closed loop orchestration refers to all of the operations required to manage a closed loop until it reaches its goal.

Intent [2]:

• Intent defines a goal that must be achieved by the closed loop system. Intents can be constituted by machine or user but it is achieved by the machine, such as a closed loop orchestrator.

3. ITU Build-a-thon Challenge: ACTIVITY 4

The general purpose of the Activity 4 is to take an intent from the non-real time RIC closed loop and apply it to the near-real time RIC lower loop. The lower loop monitors RAN resources and makes decisions to achieve the intent.

The details of build-a thon activity 4 are presented in this section. The overall architecture of the system is illustrated in Figure 1.

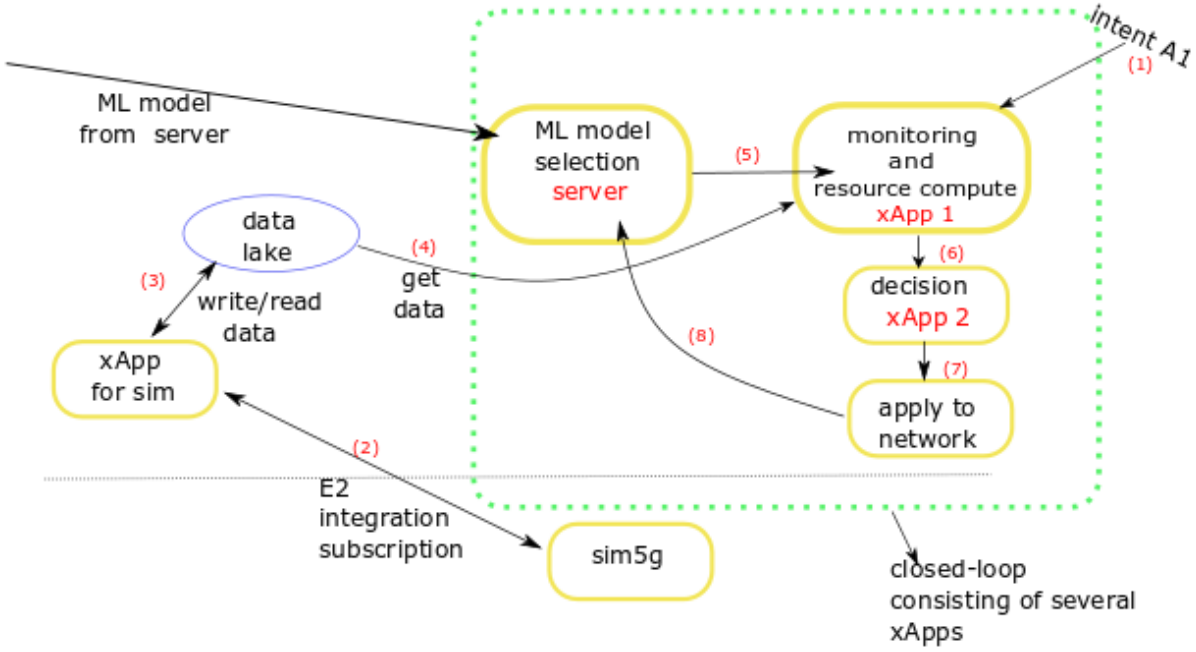

Figure 1: Block of closed-loop implementation for Emergency slice

3.1. STEPS

There are two services developed and implemented as RIC xApps and are deployed using docker containers.

ML Model Selection (server) : Different ML model for inference can be available with different complexity and performance. From a server implemented as a docker container we dynamically select different ML models depending on a periodic basis or a external request.

Monitoring and Resource Compute (xApp 1): Monitoring of RAN resources (i.e., PRB utilization) by using advanced machine learning algorithms. It reads data from a data lake periodically or when it is needed. Then, using this data it predicts how much resource will be available in the near future. This information is used for other xApps to make resource allocation decisions.

Decision (xApp 2): After receiving the forecasted RAN resource in near future, this xApp makes a resource allocation decision for the current and emergence slices depending on their SLA requirements.

Communication between xApps are provided via RMR messaging which is also used within the O-RAN software community. The workflow of our implementation is given next.

3.2. WORKFLOW

The workflow of our implementation shown in Figure 1 is given below:

(1): Get intent from higher-loop. It indicates if there is an emergency case and monitoring xapp is triggered
(2): Subscribe to sim5g xApp to get data from the simulator/testbed.
(3): Write data to data lake to be used later for ML training.
(4): Data is sent to xApp1 for model training and inference
(5): Different ML models can be selected from server here and sent to xApp1 for inference.
(6): xApp1: Resource monitoring such as PRB utilization. Here, ML model which can be fetched from our local repository is used. It also analyzes whether there is an overutilization/underutilization.
(7): Result obtained on (6) is sent to xApp2 which will make the final decision. It decides whether it is needed to allocate more resources on RAN for emergency slice. Then it applies the decision to the real network (allocate more PRB for the emergency slice, E2 CONTROL).

3.3. OUR ACHIEVEMENTS

We list our achievements in this ITU AI challenge as follows. Our code is available (GPR repository): https://github.com/krcmehmet/ITUChallenge_BuildaThon_Activity4.git

3.3.1. Dynamic ML model selection/pulling

In order to activate a low-level closed loop that operates at RAN first a high-level intent that comes from the higher order closed-loop must be generated and parsed. The generation of this intent is not the scope of this activity and it is created by another team in this challenge and is sent to us. As a first step for our implementation to work, we fetch and parse this intent correctly. The intent is a .json file and contains necessary information regarding the existence of an emergency case. The information in this intent file activates us to proactively handle the emergency case by allocation RAN resources effectively and sufficiently. After parsing the intent file and detecting an emergency case, the available RAN resources must be monitored to understand whether it is possible to handle the emergency case. To do that, a machine learning model can be utilized for monitoring. However, there might be different ML models with different prediction performance, level or price. Some ML models can be good at making long-term predictions whereas some can make good short-term predictions. Also, inference with some ML models might have less computation cost than others. Due to the availability of different choices on using ML model, a ML model selection is needed. In our implementation, the code **model_handler.py** handles this ML selection and it makes it possible to dynamically select an ML model stored at a server. More specifically, on the server we store different ML models with different model ID, and we can change the ML model on a time-basis or can request a ML model by providing a specific model ID.

3.3.2. Time-series forecasting of traffic for monitoring using Gaussian Process Regression

Network slicing (NS) is one of the key technologies that comes with 5G. With NS, the network is sliced into several numbers of sub-networks that can have dedicated network resources over different domains. For instance, at the RAN domain, an operator can allocate dedicated frequency resources (PRBs) to each slice. Also, different slices may have different

SLA requirements on latency, bandwidth, reliability, etc.. The operator needs to ensure that the underlying infrastructure SLAs for each slice are guaranteed. The SLA guarantees need to be maintained even though the need for resources of each slice can vary over time under dynamic networking conditions. For example, an operator can deploy a separate slice for video streaming and during the peak hours in a day it needs to allocate additional resources to meet the SLA requirements on the slice. In case of emergency, a new slice (**Emergency slice**) must be deployed by operators to handle the traffic in the emergency area, and the necessary amount of resources must be also allocated to ES. In this activity, we consider NS with ES as a dynamic resource allocation problem in the RAN domain. There might be different numbers of slices with different SLAs and when an emergency case occurs, ES is deployed and the resources needed for ES are maintained in an **autonomous way**.

In order to make correct resource allocation, the traffic prediction of each slice is critical to determine to gather some information on the minimum amount of resource needed for the SLA requirements. However, wireless traffic is highly dynamic and exhibits nonlinear patterns, hence it is not easy to capture the dynamic via linear models. Therefore, it has been becoming a de-facto to use machine learning models to predict the traffic. Neural networks, deep neural networks or recurrent neural networks are commonly applied for the traffic prediction. However, there are two major problems in using NN. First, it has well-known training problems, and secondly, it involves a black-box operation and it is not easy to interpret the outcome of the prediction. On the other hand, Gaussian Process Regression (GPR) has been gaining more attention due to its interpretability and prediction accuracy. In addition to its good prediction accuracy GPR can also provide information on the uncertainty of prediction which is important when making resource allocation allocation. In this activity, we assume that measuring PRB usage can reflect the traffic characteristics and we consider a time-series forecasting problem in which we predict PRB utilization by using GPR.

Gaussian Process: a Gaussian process is defined as a collection of random variables, any finite number of which have a joint (multivariate) Gaussian distribution. In Gaussian process regression, we assume the output y of a function **f** at input x can be written as:

$$y = f(x) + \sigma$$

where σ is noise with a Gaussian distribution, $\sigma \sim N(0, \sigma_n^2)$. Moreover, in Gaussian process regression, we assume the function f(x) is distributed as a Gaussian process:

$$f(x) \sim GP(m(x),\ k(x, x^{'})).$$

In other words, with GPR, the function that we want to model is also a random variable that follows a particular distribution [7]. A Gaussian process GP is a distribution over functions and is defined by a mean and a covariance function. The mean function m(x) reflects the expected function value at input x, i.e., the average of all functions in the distribution evaluated at input x. The coverence function which is also called a kernel function $k(x, x^{'})$ models the dependence between the function values at different input points x and x'. Choosing the kernel function for GPR is critical and this choice is based on assumptions such as smoothness of the underlying unknown function. Hence, it is always better to make a good expectation on the behavior of the underlying function to find the most suitable kernel, this may require domain knowledge. Another advantage of GPR is that different kernels can be combined for a better representation of the best kernel.

After giving an overview of GPR, we now explain how we use GPR for the prediction of traffic. We note once again that the traffic is characterized by the PRB usage and hence we predict PRB utilization at the RAN domain. First we obtain real-world data from [3]. In this study [3], in an urban area, PRB utilization (our data) over LTE network is measured for a user and collected and reported at every 500 ms.

Figure 2 shows 1000 samples of PRB utilization data.

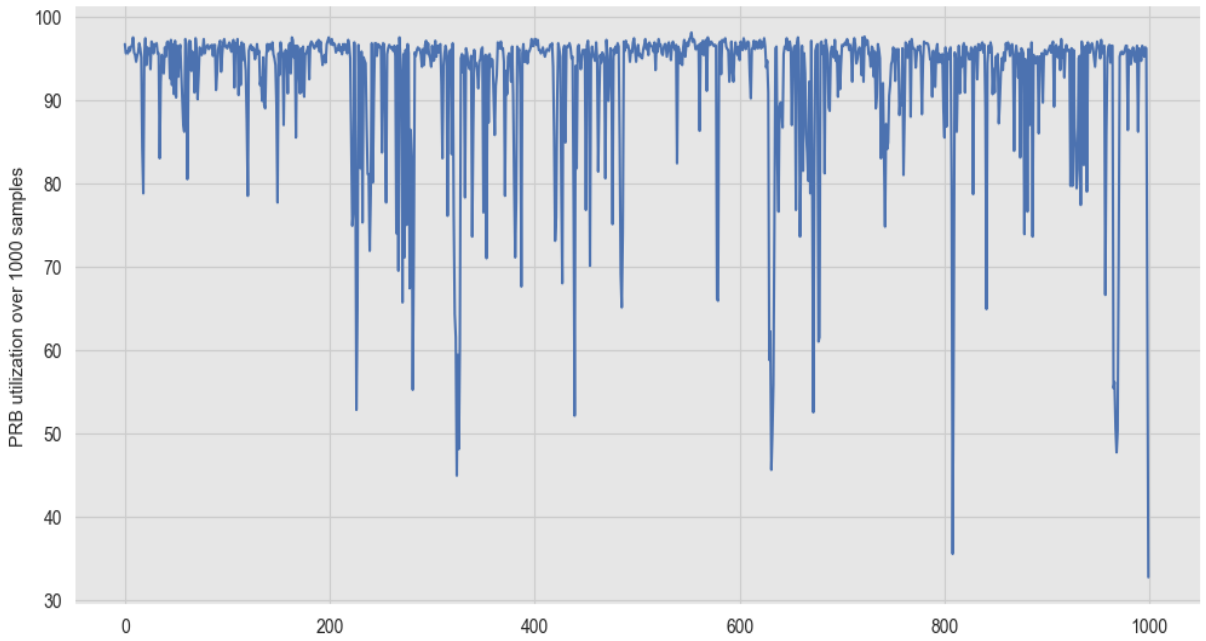

Figure 2: PRB utilization over 1000 sample points

It is important to understand the characteristics of the data to select the best kernel. When we take a closer look to our data, we see both a periodic and varying data characteristics:

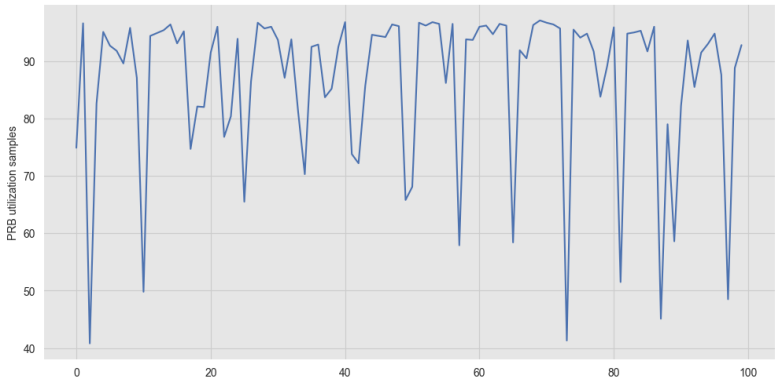

Figure 3: Periodic-type characteristics over some period

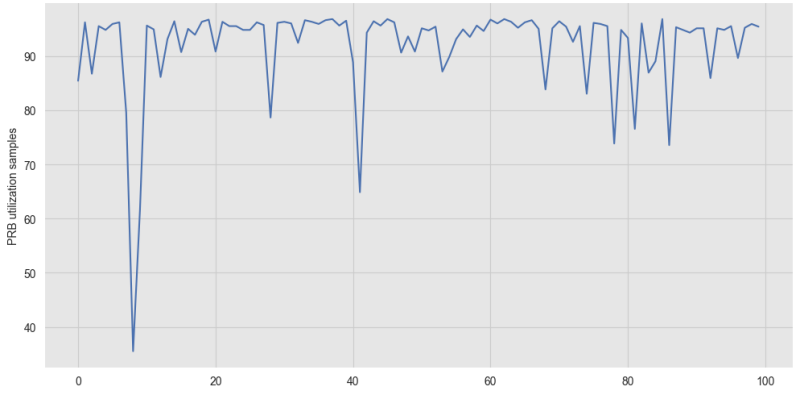

Figure 4: Constant-type characteristics over some period

The observations from Figure 3 and 4 help us to determine a good kernel to be used for PRB prediction with GPR. These observations suggest us to use a kernel represents both types of characteristics. Therefore, we use exp-sine-squared kernel that represents both the periodic-type and also sudden variations in the data. .

Forecasting with GPR:

Figure 5 shows the PRB forecasting with GPR. First, we train the GPR model with the last 100 samples and the chosen kernel as described above. Then, we use the trained GPR to make predictions for the future 50 samples. From Figure 5, it can be concluded that the prediction with GPR is good enough to make efficient resource allocation proactively. We also note that when making predictions for the next 50 points, GPR also provides information on the uncertainty of these predictions as we point as the upper bound. These upper bounds on the predictions can be utilized when making resource allocation to make sure that the correct amount of PRBs are allocated with satisfying the SLA requirements. In our implementation, **prediction_trainer.py** implements and trains GPR in Python. The PRB data is stored in our local repository. It is also possible to use different ML model for inference. We make it possible to switch between different ML models and our code **prediction.py** implements this feature. For O-RAN integration the inference is implemented as a separate xApp and **prediction_xapp.py** creates a docker container for the implementation of the inference as a microservice to be used for O-RAN.

Figure 5: Time-series forecasting of PRB utilization with GPR

3.3.3. Resource allocation at RAN for emergency slice

After the predicted traffic is obtained through GPR, the next step is to determine how much resource should be allocated to ES. We need to take into account the SLAs of other slices in the network. If we allocate more resources than ES needs, then SLAs of other slices may degrade. Also, if we allocate less resources for ES than it needs, then the emergency case cannot be handled. Hence, it needs careful action during resource allocation. We consider that RAN resources are given in terms of frequency resources or PRBs, and different slices are allocated to different amount of PRBs. This amount is determined by the operator and can be fixed. Although this strategy is good to have a dedicated network, it is not always efficient as not every slice is active all the time and usee their PRBs with 100% utilization. That is to say there can be some leftover PRBs that are not used by the corresponding slices [4],[5].

We consider two cases: in the first case, ES does not have any dedicated PRB allocated to it but it can only use the unused PRBs from other slices. The advantage of this strategy is to guarantee the SLAs of other slices but ES can have significant degradation as it can only use the leftover PRBs, and in some time the leftover PRBs may not be large enough to support the emergency case. In the second case, we dynamically borrow PRBs from other slices to support the emergency case with the aim of minimizing the degradation on the SLAs of other slices. With this strategy, the priority is given to ES and we guarantee that the emergency case

is dealt with successfully but we also minimize the negative impact of borrowing PRBs on the SLAs of other slices. We develop two algorithms to implement these two strategies:

Algorithm 1 (ALG1):

ALG 1 implements the first strategy in which only the leftover PRBs from other slices are allocated to ES. The details of ALG1 are given as follows:

The inputs to ALG1:

T = Total available PRBs of the system (i.e., For LTE, 100 PRBs)

W = Prediction window (i.e., next prediction time, 500 ms)

P = Past training window (the last 100 samples)

o = Compensation

N= Number of slices in the network

T_n= Amount of PRBs allocated to slice n.

D_n= PRB utilization time-series data for each slice

For each other slice n

Step 1: Train GPR with the latest P training data

Step 2: Forecast PRB utilization over the next W samples with GPR ➡ U_n

Step 3: Calculate maximum possible PRB utilization using upper bound ➡ $C_n = U_n + o_n$

Step 4: Calculate forecasted PRB usage of all other slices over next W samples ➡ $B_n = T_n C_n$

end

Step 5: Calculate available PRBs for Emergency Slice ➡ $P_{ES} = T - \sum_{n=1}^{N} T_n C_n$

Step 6: Allocate PRBs to ES ➡ P_{ES}

To illustrate the operations of ALG1, let us consider two slices and the allocated PRBS to these slices are T1 = 40 PRBs and T2 = 60 PRBs and in total the system has T = 100 PRBs. Let us also assume that the PRB utilization of these slices are %80 and %90, respectively. That means the first slice uses only 40*%80=32 PRBs and the second uses only 60*%90=54 PRBs. Hence, 40-32= 8 PRBs from the first slice and 60-54=6 PRBs from the second slices (in total 14 PRBs) can be allocated to ES with ALG1 for this example.

Algorithm 2 (ALG2)

ALG2 implements the second strategy in which we borrow PRBs from other slice with the minimum negative impact on them. We assume ES needs an E amount of PRBs. We first allocate the available leftover PRBs to ES. If it is not enough then we borrow PRBs from other slices by minimizing the performance degradation on them. The details of ALG2 are given as follows:

- The inputs to ALG2:

T =Total available PRBs of the system

T_n= Amount of PRBs allocated to slice n

W = Prediction window

P = Past training window

o = Compensation

N= Number of slices in the network

D_n= PRB utilization time-series data for each slice

E = Amount PRBs needed for emergency slice

- The outputs of ALG2:

$x_n(t)$ = amount of PRBs needed for slice n at time t

$y_n(t)$ = amount of PRB taken from slice n at time t

The idea is that we borrow PRBs from other slices to meet the requirement of ES and also minimize resource shortage of other slices. With this aim, we solve the following optimization problem:

P1:

$$\min \sum_{t=1}^{W} \sum_{n}^{N} \underbrace{max\{0,\ x_n(t) - (T_n - y_n(t))\}}$$

We damage slice n this amount by taking PRBs from it.

$$\text{s.t. } 0 \leq y_n(t) \leq T_n \ \forall n$$

$\sum_{n}^{N} y_n(t) \geq E$ (guarantee that emergency slice gets enough PRBs)

- $x_n(t)$: PRB usage for slice n at time t
- T_n : Total PRBs given to slice n
- $y_n(t)$: Number of PRBs taken from slice n at time t to be used for emergency slice

The problem is not easy to solve since it involves a non-linear operation with max{.} operator. However, we use an auxiliary trick and convert this problem to an easily solvable integer program. This problem is transformed to a solvable integer problem by using auxiliary variable u_n.

P2:

$$\min \sum_{t=1}^{W} \sum_{n}^{N} u_n(t)$$

$$\tilde{x}_n(t) + o_n(t) - \left(T_n - y_n(t)\right) \leq u_n(t)$$

$$0 \leq y_n(t) \leq T_n \ \forall n$$

$$\sum_{n}^{N} y_n(t) \geq E$$

$$u_n(t) \geq 0$$

$$x_n(t) = \tilde{x}_n(t) + o_n(t)$$

$x_n(t)$: Actual PRB usage at time t in future. This cannot be known in advance.

$\tilde{x}_n(t)$: Estimated PRB usage with GPR

$o_n(t)$: Estimation error. Upper bound can be used provided by GPR.

The optimization problem P2 is solved by using **GEKKO Python package**. The package is developed to solve linear-integer programming. The solution of P2 gives how many PRBs we should take from each other slices and allocate to ES. Our code **decision.py** implements these two algorithms, and **decision_xapp.py** creates a docker file to run this implementation as microservice to be ready for the use in O-RAN integration.

We would like to also mention our data format. In a simulation scenario, lt us assume that we have two slices with different PRB requirements

- **T1**: number of PRBs assigned to Slice 1 by the operator(e.g., T1=40)
- **T2**: number of PRB assigned to Slice 2 by the operator(e.g., T2=60)
- **T** : total number of PRB in the system (e.g., T= 100 PRBs)
- Timeseris PRB utilization for each slice:

Our input data for slice 1 ➡ **[timeseris PRB utilization in %]** –granularity 100 ms, 200 ms

Our input data for slice 2 ➡ **[timeseris PRB utilization in %]** –granularity 100 ms, 200 ms

Example ➡ [86.3 88.1 92.4 90.3 91 89.45 84.3 .....] (PRBs utilization in percentage)

Emergency slice needs **E** number of PRBs (e.g., E = 20)
Our output data ➡ number of PRBs decided to be allocated to Emergency Slice

We have evaluated the performance of ALG2 under the scenario that there are two other slices and T1=40 and T2 = 60 PRBs allocated to them. However, these slices do not always need that many slices and by applying ALG2, we borrow PRBs to satisfy the requirement of ES. We assume the ES needs 20 PRBs. Figure 6 shows how many PRBs are taken from other

slices over 45 time-instants. Depending on the predicted PRB usage of other slices, from the first and second slices ALG2 takes 11 or 12 PRBs and 8 or 9 PRBs, respectively, and in total 20 PRBs are ready to be used by ES at every instant.

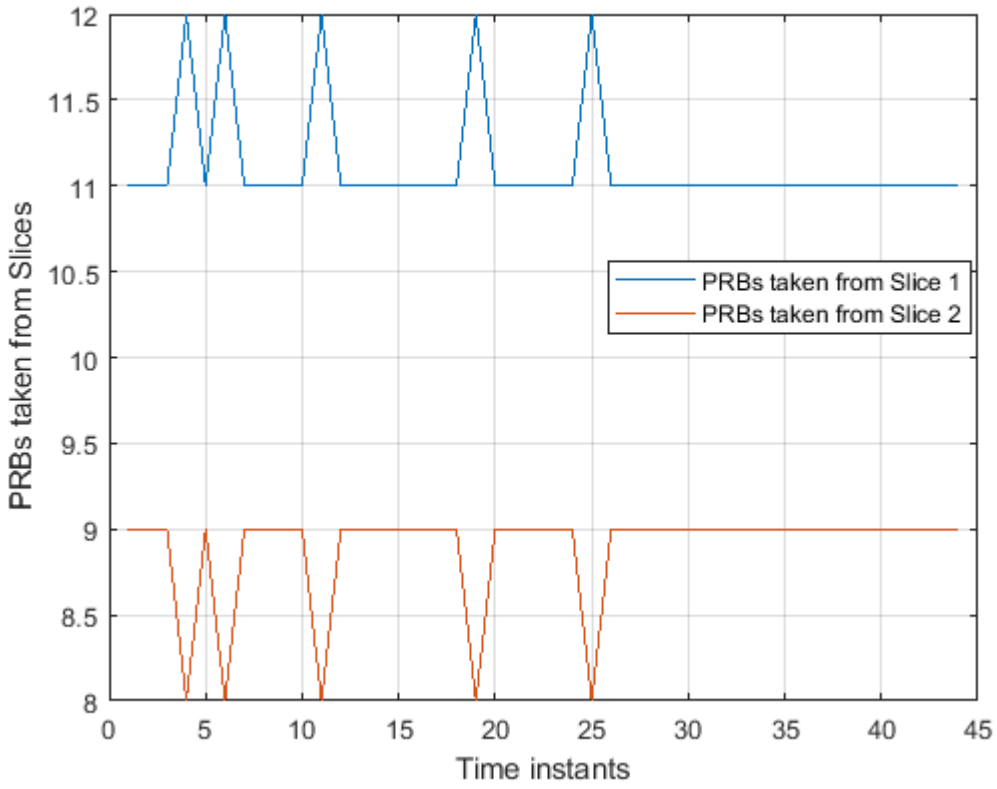

Figure 6: PRB allocation for ES

3.4. IMPLEMENTATION

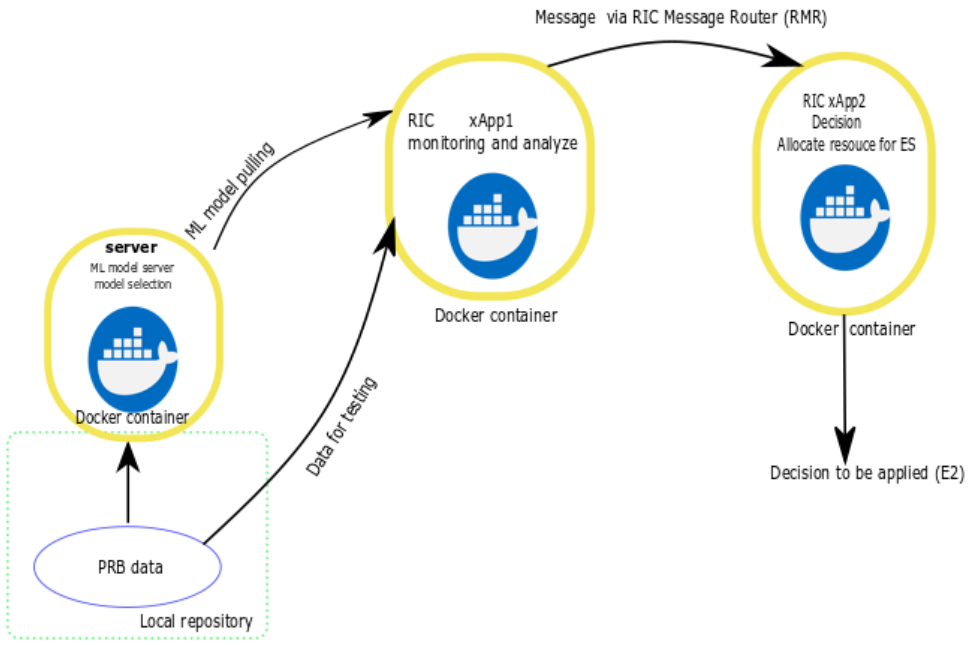

Figure 7: Our implementation details

First, we receive a high-level intent and parse it with read_intent.py. The output of the intent activates us to start a low-level closed loop that monitors and computes RAN resources and makes a resource allocation decision for emergency case. The **xApp1** monitors RAN resources and makes forecasts for the future PRB usage of the network. It also computes the available resources at the RAN domain. The forecasting and resource information is sent **to xApp2**, which is our decision xApp, via **RIC message router (RMR).** RMR is developed by O-RAN Software Community (SC) and we also utilize this messaging protocol in our implementation. xApp2 receives the necessary information from xApp1 and solves the problem P2 (or P1) to find out the necessary PRB resources needed for ES and make the resource allocation. The output of xApp2 will be sent to the real network to be applied through the E2 interface of O-RAN when the real integration starts. Another docker container acts as a web **server** where we keep different ML models to be used for monitoring or any other activities. **model_handler.py** implements ML selection/pulling task in which we select ML dynamically depending on the performance of the current ML. For instance, with this

implementation, it is possible to call another ML that may have a better performance than the current model.

The basic properties of a xApp can be summarized as follows:

xApp:

- xApp itself (Python)
- Docker file
- -xApp description (json)
- RIC message Router to send messages between xApps (RMR library defined in O-RAN SC)
- Data format (json in O-RAN SC)

4. CONCLUSION

In this ITU AI Challenge-Activity 4, we have developed and implemented a low-level closed loop that autonomously handles an emergency case. The closed-loop consists of several microservices deployed as docker containers. Each microservices has its own duty such as monitoring, computing, ML selection and resource allocation. The microservices communicate through the RIC Message Router (RMR) that has been developed by O-RAN SC [6]. As related to ITU FG-AN, within this activity: intent parsing, traffic monitoring, resource computing and allocation are performed in an autonomous way. The key lessons that we take from this activity are: creating a closed-loop needs careful design as it can be composed of different modules (microservices). The more microservices we add to create a closed-loop, the more modular structure is achieved. However, creating a closed-loop with several modules brings communication and computation problems. Overall integration including A1/O1/E1 interface integrations is critical and which parts of this integration can be realized in an autonomous manner is a question to be addressed. The real-time system performance will have to be tested to ensure compliance with closed loop specification.

5. REFERENCES

[1] A. Dandekar, J.Schulz-Zander, H.Wissing, Fraunhofer HHI, “Use case and requirements for orchestration of AI/ML basedclosed loops to enable autonomous networks”, Fraunhofer HHI, Apr., 2021.

[2] A. Dandekar, “ Introductory Tutorial for ITU ML5G Build-a-thon”, presented at ITU FGAN 3rd Virtual meeting, Germany, June 17, 2021.

[3] V. Raida, P. Svoboda, M. Rupp: “Real World Performance of LTE Downlink in a Static Dense Urban Scenario -An Open Dataset“, IEEE GLOBECOM2020.

[4] X. Foukas et al., “Orion: RAN Slicing for a Flexible and Cost-Effective Multi-Service Mobile Network Architecture,” in ACM MobiCom, 2017.

[5] A.Okic, L. Zanzi,V. Sciancalepore, A. Redondi, X. Costa-Pérez,“ π-ROAD: a Learn-as-You-Go Framework for On-Demand Emergency Slices in V2X Scenarios“, IEEE INFOCOM, 2020.

[6] https://wiki.o-ran-sc.org/pages/viewpage.action?pageId=3605041

[7] C. E. Rasmussen and C. I. K. Williams, Gaussian Processes for Machine Learning. MIT Press, 2006.

Our Team: AUTOMATO

Mehmet Karaca: He is an Asst. Prof. at the Dept. of Electrical-Electronics Eng. , TED University, Ankara, Turkey.

Doruk Taylı: He works at Q Bio Inc., California, USA, as a Lead Computational Software Engineer.

Özge Simay Demirci: She is an undergraduate student at the Dept. of Electrical-Electronics Eng., TED University, Ankara, Turkey.

This work has been supported by Scientific and Technological Research Council of Turkey (TÜBİTAK, grant no. 120E485)