A Network-Slicing System for Software-Defined Mobile Networks

Raini Wu
Electrical and Computer Engineering
University of California, San Diego
San Diego, United States
rainiwu@ucsd.edu

Jiawen Xu
Electrical and Computer Engineering
University of California, San Diego
San Diego, United States
jix034@ucsd.edu

James Villani
Electrical and Computer Engineering
University of California San Diego
San Diego, United States
jfvillan@ucsd.edu

Lisa Takai
Electrical and Computer Engineering
University of California San Diego
San Diego, United States
ltakai@ucsd.edu

I. Introduction

In the past several years, continued interest in software-defined mobile networks has led to several open-source implementations of various virtualized mobile network components, including several radio access network (RAN) and core network (CN) implementations [1] [2] [3] [4]. However, despite the fundamental improvements in configurability afforded by virtualization of network components, the existing implementations of software-defined mobile network components do not provide the means to leverage this configurability. Indeed, the killer feature of software-defined networks - its inherent flexibility - is painfully absent from the currently available open source software-defined mobile network implementations.

One of the most apparent symptoms of this separation between expectation and reality can been seen through a look at network slicing. Network slicing is a technology which is expected to enable a generational shift in network capacity, by allowing different service guarantees to be provided over the same network [5]. Network slicing has long been touted as a core feature of 5G networks, and largely expected to be implemented alongside the softwarization of network components [6]. However, real RAN-level slicing implementations are notably absent from openly available systems.

We seek to fill this gap in existing open source software-defined mobile networks, by implementing a novel network slicing system on top of an existing open-source software-defined RAN as shown in Figure 3. We design and implement our network slicing system on top of srsRAN [1], a leading open-source software-defined RAN implementation. With our contribution, we explore the real-world implementation-level challenges related to realizing RAN-level network slicing, as well as provide some early results on the possible gains RAN-level slicing can provide.

II. Background

Before we discuss the design and implementation of our system, we first need to explore the fundamental principles of network slicing, and the application to srsRAN. Additionally, we also discuss in detail the scheduling behavior of srsRAN, which is relevant to our modifications to the stack.

A. Network Slicing

Network slicing is a technology which improves overall network capacity through the provision of different network-level guarantees for different applications. Network slicing is often touted as a core enabler of the generational improvements provided by 5G networks, and closely tied with the advent of software-defined mobile network systems [5] [6] [7] [8]. The method by which network slicing is achieved may differ, but one such method is through intelligent RAN-level scheduling decisions.

To achieve its gains, network slicing relies on having asymmetric workloads on the same network. Indeed, slicing is only effective when certain applications have more stringent requirements than others on the same network. Without varying application-level requirements, the network cannot effectively be sliced; if all applications need the same amount of resources, it does not make sense to prioritize some over others. Furthermore, as the actual network characteristics are not improved by the common conception of network slicing - i.e. the amount of maximum bandwidth and minimum latency remains the same - the prioritization must be aware of application-level metrics, like quality of experience (QoE). As such, the implementation of any network slicing system must consider two points: the degree to which the network workload is asymmetric, and the application-level metrics to optimize for.

In our work, we choose to implement RAN-level network slicing. Network slicing at the RAN-level is typically implemented as a MAC-layer scheduler improvement [6]. RAN-level scheduling in the MAC-layer can achieve network slicing by prioritizing different users based on the particular slice of the network which they belong to. However, the MAC layer of mobile networks typically do not include application-level

metrics, which are required to achieve network slicing. In the context of RAN-level slicing, the need of application-level metrics necessarily means cross-layer optimization. Recently presented work details a novel architecture, as well as the associated challenges, when requiring cross-layer optimization of software-defined RAN-level scheduling [9]. The work exposes an additional challenge associated with implementing network slicing at the RAN-level specifically - the need to utilize cross-layer optimization necessarily means that data delivered on non-real-time time scales must be translated to real-time scheduling decisions.

B. srsRAN

We choose to implement our network slicing system on top of a leading open-source software-defined RAN, srsRAN [1]. Although both 5G and LTE versions of srsRAN are available, we choose to implement our network slicing system on top of srsRAN version 21.04, which includes only a LTE RAN and CN implementation. This choice of utilizing an LTE stack is primarily motivated by the relative stability and maturity of the LTE srsRAN implementation.

A typical srsRAN deployment is pictured in figure 1. srsRAN consists of several separate processes, including *a)* srsUE, an LTE user implementation, *b)* srsENB, an LTE basestation implementation, and *c)* srsEPC, an LTE core network implementation.

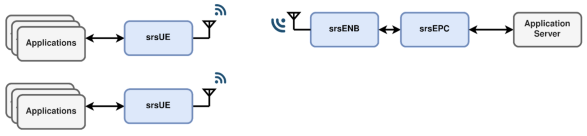

Fig. 1. Typical two-user srsRAN deployment

In our system, we only modify srsENB, the srsRAN basestation component. srsENB contains the full RAN-level stack, including a radio device (RF) interface, physical layer (PHY) implementation, medium access control (MAC) layer implementation, and several upper layer components. srsRAN provides two MAC-layer schedulers by default, a time-level round robin scheduler, and a time-level proportional fair scheduler. As the time-level proportional fair scheduler is unstable, we only evaluate the time-level round robin scheduler as a baseline against our new network slicing scheduler implementation.

C. Round-Robin Scheduling in srsRAN

To provide context for our modified scheduler implementation, let's quickly explore the default round-robin scheduling available in srsENB. The overall round-robin scheduling implementation, as well as its relation to the rest of the server-side srsRAN stack, is pictured in figure 2. The round-robin scheduling is a fully MAC-layer scheduler, which serves to fulfill scheduling requests from the PHY layer. In srsRAN, the srsENB PHY layer continuously senses for communications from the srsUE, as well as the occupancy of a PHY-layer downlink buffer, and sends event-based requests to the srsENB MAC layer for transmissions from both the user and the basestation. The final decision on whether a queued transmission is scheduled to be sent is up to the srsENB MAC layer. Requested transmissions are separated into time-frequency resource blocks (RB), with the time-domain splitting occuring on the level of transmission time intervals (TTI), and the frequency domain splitting occuring as physical resource blocks (PRB). A single TTI corresponds with one millisecond in srsRAN, while a single PRB corresponds with a few hundred kilohertz of frequency bandwidth. The srsENB MAC layer must explicitly allocate PRBs in every TTI to specific scheduling requests from the PHY layer before any data can be transmitted or received by the PHY.

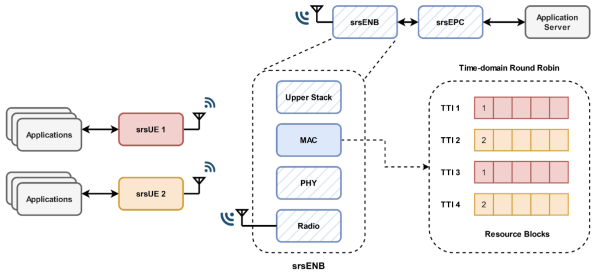

Fig. 2. Round robin scheduling in srsRAN

By default, srsRAN utilizes time-domain round-robin scheduling in the MAC-layer scheduler to respond to scheduling requests from the PHY. For every TTI, a single user is prioritized, and in the next TTI another user is prioritized. Each connected user is prioritized in order, and once all users have been prioritized, the pattern repeats again.

To improve the robustness of its default round-robin scheduling, srsRAN implements some heuristic custom behaviors to the scheduler. A key change over a standard round-robin scheduling implementation is that the round-robin scheme is two-layer. The total list of uplink and downlink scheduling requests are split into new transmissions, and retransmissions. Retransmissions are requests that were previously unfulfilled, while new transmissions are scheduling requests that have been sent for the first time. Retransmissions are fulfilled in a round-robin manner first, and then new transmissions are scheduled in a second round of round-robin. Thus, regardless of the prioritized UE per TTI, all retransmissions would be fulfilled before new transmission requests. This has the benefit of ensuring that no UE is starved during the scheduling allocation process.

We derive our modified scheduler to enable RAN-level network slicing on top of the existing round-robin scheduling implementation within srsRAN. These heuristic improvements to the default round-robin scheduler may result in improved performance for pure round-robin scheduling, but is a detriment for the custom behaviors which we seek to implement. As explored in section III-A, these heuristics are removed in our implementation.

III. Design and Implementation

At a high level, our contribution consists of three key components:

- A *new scheduler interface* into the srsRAN softwarized RAN stack, capable of modifying the behavior of the RAN in real time.
- An *intelligent network slicing application*, capable of taking decisions on how exactly the RAN behavior should be modified in real time.
- An *application simulation framework*, including simulation of various application usage patterns and corresponding quality of experience through the network.

These components and their relation to the overall srsRAN architecture is shown in figure 3. The following sections will explore our design of each of these components in detail.

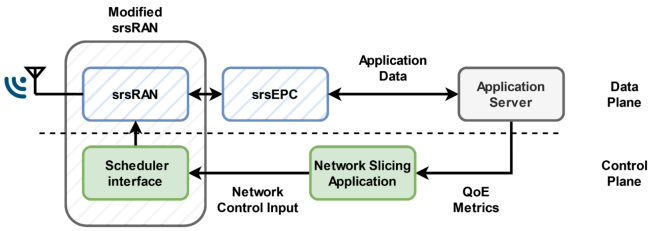

Fig. 3. Overall system architecture

A. New Scheduler Interface

To perform RAN-level network slicing, we need to expose a method of changing RAN-level scheduling behaviors from the outside of the stack. Out-of-stack implementations The existing srsRAN schedulers do not expose any out-of-stack interface by which scheduler behaviors can be changed. To expose a new control interface into the MAC-layer scheduling decisions of srsRAN, we design a new scheduler loosely based on a recently presented work [9]. The overall context of the modification is pictured in figure 4.

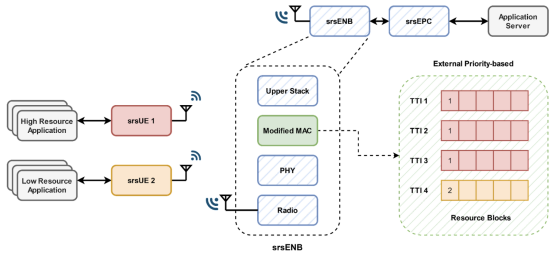

Fig. 4. Modification to srsRAN

The key challenge associated with enabling outside inputs to a real-time system like srsRAN is the difference between processing time-level guarantees. srsRAN has strict time-level guarantees to ensure the system functions in real time. To maintain real connections between users and the basestation, the srsENB implementation must perform its virtualized network functions within strict real-time constrictions. However, the nature of the out-of-stack scheduling directions based on cross-layer inputs is fundamentally different from the strictly clocked internals of srsRAN. Out-of-stack application-layer inputs certainly have no guarantees on the millisecond scale as to when inputs may be provided. As such, there is necessarily a wide disparity between the execution of our scheduling decisions and their initial provision. As explored in section II-C, a failure of the scheduler to fulfill scheduling decisions per-TTI (where a TTI is one millisecond) would directly correspond users actively disconnecting from the network. Then, it is clearly necessary to implement our new scheduling interface in a loosely coupled manner, to allow control inputs when available and to allow continued operation of the RAN when inputs are not immediately available.

To achieve this loosely coupled interface between srsRAN and our network slicing implementation, we rely on the ZeroMQ message passing library [10]. ZeroMQ provides an abstraction layer over several common inter-process communication methods, including TCP and Unix Domain Sockets. Using ZeroMQ-powered Unix Domain Sockets, we implement a new MAC-level scheduler within srsRAN in C++, which can perform robustly with and without control input. We utilize the ZeroMQ publish/subscribe socket pattern, which can allow a publisher to send arbitrary messages into a Unix Domain Socket, and a subscriber to optionally receive messages. By using a subscriber socket in a non-blocking configuration, our scheduler implementation can receive outside input without abandoning real-time guarantees in the situation where input is not immediately available. Relying on an off-the-shelf library like ZeroMQ greatly simplified the socket programming dimension of the project - simple ZeroMQ API calls replaced the requirement for low-level system calls for cross-process synchronization. Following the success of ZeroMQ in the new scheduler interface, ZeroMQ is utilized in all parts of the project, serving as the primary backbone of our integration efforts.

Our actual scheduler implementation is based on the existing round-robin scheduler implemented in srsRAN as described in section II-C. However, instead of prioritizing the scheduling request of each user in turn across multiple TTIs, we can choose to prioritize a single user which is delivered over the ZeroMQ communications channel. The user which is chosen to be prioritized can be changed during any TTI, but will persist in the absence of continued input from an outside control application. This way, we can maintain the scheduling behaviors that are necessary for srsRAN to maintain an active connection between it and its users, while still allowing for custom behaviors. Additionally, we break the heuristic improvement of scheduling all retransmissions before transmissions implemented in the default srsRAN implementation, as this will allow unprioritized UEs to obtain scheduling resources which we do not wish to allocate. To do so, we implement new uplink and downlink scheduling functions within our scheduler, which will prioritize each user over

prioritizing retransmissions. Our changed scheduler is fully integrated into srsRAN, requiring no additional compilation steps or recompilation to swap between the default scheduler and the new scheduler. We slightly modify srsRAN's configuration parsing behavior to allow the utilization of our new scheduler instaed of their default round-robin or proportional-fair schedulers.

B. Application Simulation Framework

In order to properly verify our network slicing implementation provides a meaningful improvement in terms of client performance, it was necessary to have a means of evaluating clients satisfaction across different latency and throughput requirements while still being a metric affected by scheduling. As discussed in section II-A, the core gains provided by network slicing is fundamentally dependent on asymmetric application use cases. This method of evaluation needs to be stable across these varying conditions, with few application types beyond streaming video having as much variability in requirements. With video, the Quality of Experience (QoE) metric that best reflects client satisfaction while being scheduling dependent proved to be stall time. We define stall time as any time when the client does not have a video frame to display to the user and is buffering as a result.

As we specifically want to minimize stall time as a percentage of the total time streaming, we defined our QoE metric concretely as follow:

$$\text{QoE} = [1 - \frac{\text{Total Time Stalled}}{\text{Total Time Elapsed}}] * 100\% \quad (1)$$

As can be seen, a client's QoE is effectively the percentage of time when the device is not stalled and is therefore providing a good quality of experience to the user. The less time the stream is stalled, the higher QoE it has. By changing the frame time and buffer size we can change a client's throughput and latency requirements significantly, greatly reducing complexity as a single client codebase could be used to create four drastically different clients. We will define the *frame time* to be the period of time it takes for a single video frame to be consumed, inversely proportional to the frame rate. *Buffer size* is defined as a storage of saved video frames from where video frames will be consumed once every frame time. For an unbuffered client, we simply define the buffer size to be 1 frame such that it cannot store a cache of future frames. By varying the buffer size and the frame time, we classify the clients into the following four scenarios:

Throughput \ Latency	High	Low
High	High throughput buffered video	High throughput unbuffered video
Low	Low throughput buffered video	Low throughput unbuffered video

Fig. 5. Four scenarios of users

If the buffer size is larger, the chance of stalling is lower since more video frames can be cached locally in advance. If frame time is higher, the chance of stalling is lower since it takes more time to consume a video frames. In other word, among the four scenarios, high throughput unbuffered video will have a lower QoE since the buffer size cannot cache frames in preparation for the next frame time, while low throughput buffered video will have a higher QoE since the buffer size is large and the consumption speed is low and is therefore able to cache many frames in advance.

The video server itself receives requests from clients via ZeroMQ communicating through srsRAN. Each request will include the client's identifier, the current QoE, and if that particular client is requesting an unbuffered chunk of video or a buffered chunk of video. This data is generated on the fly by the server and is sent back to the client via the same ZeroMQ socket as the request was received at. Simultaneously, using the asyncio library, the server is also managing requests for the QoE metrics by the RL model through a separate ZeroMQ socket. This secondary socket is instead communicating over localhost as opposed to going through srsRAN. Once the expected number of clients to be connected to the server has been reached, the server will reply to requests from the RL model with a dictionary consisting of each client's user identifier as the key and the current QoE metric as the value. This is managed simultaneously with requests made on the srsRAN client socket by the server. This allows the RL model generate the priority listing and provide the identification numbers that srsRAN can understand.

C. Intelligent Network Slicing Application

We use reinforcement learning (RL) model to implement our network slicing application for srsRAN. Habiby and Thoppu introduce reinforcement learning used in 5G network scheduling for parameter optimization. [11]. The fundamental algorithm for reinforcement learning model is Q learning [12]. Q learning algorithm consists of agents and environment. Agent acts like a human being who decides the next action, and the environment provides the reward based on how good the action is performed. The action will be either higher or lower the priority of a specific client. A new action will be chosen based on the memory of the agent. Figure 6 explains the inner relationship between our environment and agent.

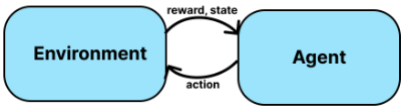

Fig. 6. Q learning: environment and agent

1) Agent: The agent in our design can be analogized as a "human being whose brain is constructed by Q-learning algorithm," who will take actions based on the result of the

Q learning algorithm. Since it is a single-agent model, we are using a standard Q learning function [12]:

$$Q(s,a) = Q(s,a) + \alpha(R + \gamma maxQ(x',a') - Q(x,a)) \quad (2)$$

where α is the learning rate. γ varies from 0 to 1. If γ reaches 0 then the model will only consider the reward for the training process. R is the change of rewards
The action space for N clients in our design is defined as:

$$\text{actions} = \begin{bmatrix} C_1 & -C_1 & C_2 & -C_2 & ... & C_N & -C_N \end{bmatrix} \quad (3)$$

where C_i is the change of priority a specific client.
Out state space is a one hot coding vector with length equals the length of action space. For example, for a 2-client setup whose action space is $[1, -1, 1, -1]$, where $C_1 = 1$, and $C_2 = 1$ the state $[0, 0, 0, 1]$ identifies the 4th action of the action space will be chosen which means the second client reduces its priority by 1 because $-C_2 = -1$. [13]. The Q value will be a function of the state and the action shown above.

2) Environment: Our environment setup is responsible for calculating the reward based on the action taken. The reward is calculated based on the if the action taken is beneficial to improve the overall QoE. The QoE metrics are not calculated within the RL Model. The RL model sets up a socket communicating with the server that sends the vector of QoE metrics which is then parsed and used to calculate the reward. The algorithm for rewards calculation is straightforward as shown below:

```
function calculate_reward(action):
    if target.QoE increases:
        return positive value
    else if target.QoE decreases:
        return negative value
```

The target client is determined by the QoE. We currently simply decide the least QoE to be the target client which is intuitive since we always want to give more resources to the least "satisfied" user. However, there could be an edge case when one user is always not "satisfied" no matter how much resources are allocated to it. We then decided to use some statistical methods involving mean and variance calculation to decide the reward function, but it did not turn out to be a good result. We consider to further improve the reward function if possible; however, based on the time constraint, we decided to place this task into our future plan.

3) Simulated RL behavior: Before we starts testing our RL model in srsRAN, we set up a simulation to test the functionality of our RL model. The simulation has four benchmarks: high throughput buffered, high throughput unbuffered, low throughput buffered, low throughput unbuffered. The simulation is not real time as it only simulate the environment by calculating the buffer consumption and estimating QoE mathematically. Although it cannot represent the performance of srsRAN, it is a good evaluation of the functionality of our RL model. We simulate the benchmarks with round robin scheduling and our RL model.

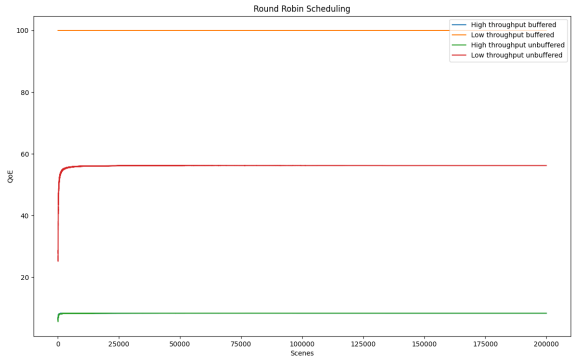

Fig. 7. Round Robin Scheduling Simulation Result

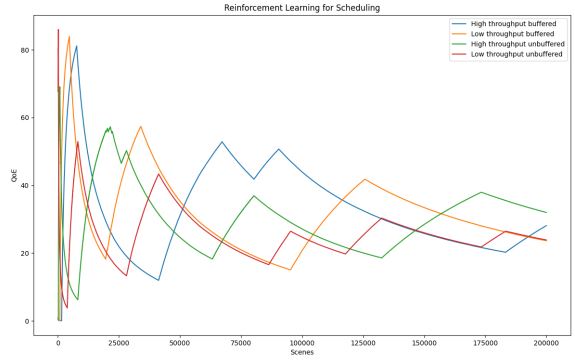

Fig. 8. Reinforcement Learning Model Simulation Result

The simulation result in Figure 7 and 8 matches our expectation. For round robin scheduling, the low throughput buffered benchmark has QoE that is close to 100%, the low throughput unbuffered QoE is close 50%, while the high throughput clients have QoE around 5%. With our RL model apply, a better resource distribution is shown from the result. The QoE of four clients merge to 40%. In other word, the low throughput buffered client, whose QoE is always close to 100% in round robin scheduling policy, will get less resources. The other three benchmarks have significant improvement after our reinforcement learning model was applied.

IV. Evaluation

We evaluate our system by running the full stack on a contemporary desktop workstation, a single CPU and GPU machine with an Intel i7-11700K CPU, NVIDIA GTX 2060 GPU, and 32GB of RAM. The full stack includes various components of a modified srsRAN (Figure 4, as well as our network slicing application and video streaming application simulation framework. As we unfortunately did have access to enough software-defined radios and attached computers to realize testing srsRAN over the air (we required three radios and three separate machines to realize over-the-air testing),

we utilize srsRAN's built-in ZeroMQ based RF module for testing. srsRAN has a built-in ZeroMQ RF module which ships fully processed IQ data over a ZeroMQ socket, instead of delivering the processed data to an software-defined radio interface. This is effectively the same as running over-the-air tests, with the exception that channel effects are not captured. As we are not interested in evaluating our system in the presence of real wireless channels, we believe that this is a reasonable tradeoff to achieve multi-user testing. However, srsRAN does not support by default the option to use multiple UEs with ZeroMQ-based RF devices. To get around this, we utilize GNURadio, an open source flow chart based RF processing software, which conveniently provides ZeroMQ inputs and outputs appropriate for our use case. We utilize GNURadio to read the complex output from all output sockets of each user, and perform a simple addition operation to create a single stream to feed into the srsENB input socket.

Several tests of a round-robin baseline or our network-sliced implementation run, with multiple users for each iteration of a test. For each test we will have 2 or 4 users, and we will configure the users based on the parameters shown in Table 1, making them stream a video with some combination of buffered or unbuffered, and low or high throughput.

	Buffer	Unbuffered
High Throughput	frame time: 0.008s buffer size: 500 recovery time: 0.064s	frame time: 0.008s buffer size: 1 recovery time: 0.008s
Low Throughput	frame time: 0.032s buffer size: 500 recovery time: 0.064s	frame time: 0.032s buffer size: 1 recovery time: 0.008s

Table 1: Configuration for UE parameters

In the following sections, we summarize our insights into the collected data. The full dataset analyzed in this evaluation will be made available as an artifact. During each test we collected QoE traces, with each test iteration running for at least a minute. The results are shown in Table 2, showing a small loss in the low throughput buffered video for our network-sliced implementation, but a significant improvement in the high throughput unbuffered video. Although there is a small loss in the low throughput buffered video experiment, the loss is insignificant enough that we can assume that this loss came from varying noise.

	Low Throughput Buffered	High Throughput Unbuffered
Round-robin Scheduling	average QoE = 98.62	average QoE = 55.25
RL-based Scheduling	average QoE = 97.17	average QoE = 60.94
Improvement	-1.48%	10.29%

Table 2: RL-based model against Round-Robin scheduling experiment results

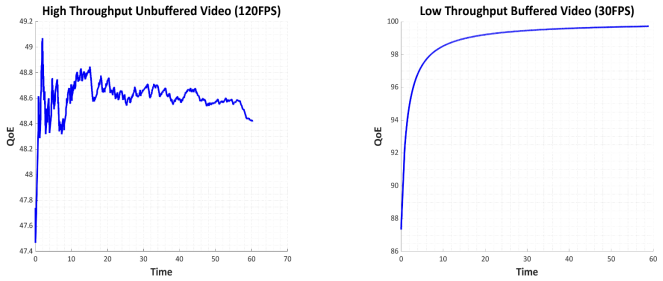

Fig. 9. QoE of High throughput unbuffer streaming (left) and QoE of Low throughput buffer streaming (right)

We take a measurement of our metrics QoE on two different users - high throughput unbuffer and low throughput buffer individually with srsRAN. The measurement matches our expectation based on our definition of the QoE. The high throughput unbuffer client in Figure 9 (left) is supposed to have lower QoE, and the low throughput buffer client in Fig 9 (right) is suppose to have higher QoE.

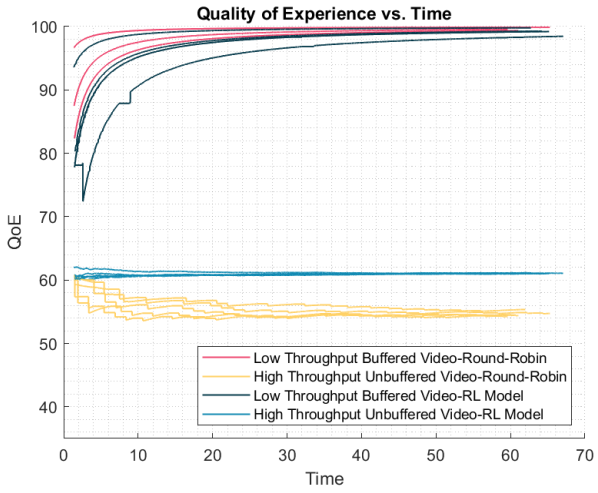

Fig. 10. Comparison of QoE with multiple clients using round-robin and RL model scheduling

Our initial testing setup includes four user cases. We first start from testing two users (low throughput, buffered and high throughput, unbuffered), and Figure 10 shows the result of our tests set using round-robin scheduling versus reinforcement-learning based scheduling. The result shows there is an improvement for high throughput unbuffered video after applying reinforcement learning model. The low throughput buffered is getting less resources when RL model is applied as it is easier to be "satisfied".

Unforunately, we were not able to produce promising results for the 4-user case experiments. We suspect that this is due to congestion in the uplink that we did not modify or control in srsRAN, so the round-robin scheduler was actually doing much better than our network-sliced implementation.

V. Challenges

During our implementation and design process, we faced a variety of challenges. In this section, we will explore the various challenges and solutions that we encountered during our development process, as well as discuss some challenges that remain unsolved.

One of our primary challenges is integration, as our network slicing application, srsRAN, video servers, and clients are all implemented in parallel and individually. We connect each module using ZeroMQ for socket programming [10]. The client and server pass messages through srsRAN while the server, RL model, and srsRAN communicate over localhost. The debugging process for sockets connecting is time-consuming. For integration, we have to schedule to work in group which requires good time arrangement as well. Each connection between the various parallel implementations requires its own interface and method of communication.

For reinforcement learning model, one of the challenges is defining the reward function. Although our rewards function is intuitive, for some of the edge cases, it does not always work. For example, when a UE does not improve QoE no matter how much resource is distributed, ideally we should move on and give up scheduling this UE. However, based on our current algorithm, the scheduler will keep prioritizing this UE since it is "least satisfied." We include some statistical calculation which seems to alleviate this situation; however there may be further work needed to resolve this problem.

Another challenge is ensuring that srsRAN is provided the correct device identifiers so that the prioritizations are properly ordered. Currently, we assign the device identifiers manually upon client startup to match the defaults assigned by srsRAN. This works well for a low number of devices that are used in our tests but as the number of devices increase this solution is no longer as scalable. If the device identifiers do not match or srsRAN skips an identifier, the RL model will provide incorrect prioritization data to srsRAN.

VI. Conclusion and Future Work

We successfully implemented the network slicing application, scheduler interface, video servers and clients, and modification in srsRAN. Before we integrate each module, we unit test each module by either simulation or checking for communication. We use ZeroMQ for communication between different modules [10]. After integration and fully testing the connection of each module, we run srsRAN with 2 user equipment using Round-robin scheduling and reinforcement learning model. The result shows that using our network slicing application for 2 user equipment model improves the behavior of the scheduler by increasing the net performance of the clients. The scheduler was successfully instructed by the RL model to prioritize the UE who has the greater need for resources.

Unfortunately, for 4 UE tests the RL model does not significantly improve the resource distribution. We suspect the reason could be lacking consideration of correlation among UEs when calculating rewards in the RL model, and that we may be causing some congestion in the uplink that we did not control. However, because of the time constraints, it is difficult moving forward to redefine another reward function and verify our hypothesis since we have to research on more statistical resources. Improving this reward function to allow for 4 UEs to see similar improvements as 2 UEs is a promising direction for possible future work.

VII. Individual Evaluation

A. Raini Wu

Raini Wu primarily contributed the design and implementation direction for this project, leveraging existing knowledge about relevant software libraries like srsRAN, ZeroMQ, and language specific concurrency libraries (such as Python asyncio and C++ STL threads) to achieve an overall implementation and design that was possible to achieve in the limited time frame that we have. Additionally, Raini was responsible for the implementation of the new scheduler, and the new scheduler interface, into the existing codebase of srsRAN. Data collections involving active deployments of srsRAN were also done by Raini, leveraging the existing knowledge of the srsRAN stack to ensure that deployments are done robustly, utilizing tools like GNURadio and the Linux ip-utils tool for creating network namespaces (to allow multiple UEs and basestation instances to be deployed on the same test machine). As part of the data colletion process, Raini also implemented new GNURadio flowcharts to accommodate the ZeroMQ-based testing of the full srsRAN stack. GNURadio flowcharts were necessary to ensure that srsRAN communicated properly. Finally, Raini also contributed in the testing and integration of the rest of the system, ensuring that all components worked together, and notifying the responsible parties when they did not.

B. Jiawen Xu

For this project, Jiawen Xu focused on implementing network slicing application module. The network slicing application is using reinforcement learning (RL) model as we expect the network slicing application to run synchronously with our QoE metrics input and output the priority to scheduler interface. The RL model includes a Q learning agent and environment. The Q learning agent performs the training and output the action while the environment will produce reward based on the action. Jiawen also implements the simulation to verify the functionality of the RL model. After finishing the RL model, Jiawen implements the communication in network slicing application using ZeroMQ to talk to the server and the srsRAN. The network slicing application have two sockets. One of the socket communicates with the video server, and the other socket communicates with srsRAN through a scheduler interface. The network slicing application will send QoE request to video server and wait for QoE periodically, meanwhile sending the current prioritized user to srsRAN. These two communication are programmed in parallel using asyncio in Python.

C. James Villani

For the project, James Villani primarily focused on developing the QoE metrics as well as the video server and its associated client-server architecture. In developing the QoE metrics, it was necessary to find a metric that could be calculated on the client side with parameters we could expect to be available while still reflecting any improvement that may occur with changes to the srsRAN scheduler. By limiting the clients to various types of video streaming to ensure direct comparability it was found that buffering ratio was the preferred metric as it reflects the client experience in a way that is directly affected by scheduling. For the video server, it was necessary to create a server that could serve multiple client different types of data at different rates simultaneously and report the received QoE metrics to the RL model. The server receives requests from each client requesting larger buffered data packets or smaller unbuffered data packets and generates an appropriately sized dummy video chunks on the fly. This server was written in Python using the ZeroMQ library to communicate with the clients and the RL model. This also required defining how the server and client communicate in a way that allows for the client to request the correct type of data and also report back a QoE metric that the server can access.

D. Lisa Takai

In this project, Lisa Takai focused on implementing the design of client side of a client-server architecture for a video streaming application. She also researched different Quality of Experience (QoE) metrics that could be used to measure the quality of the video from the user's perspective. The client calculates the QoE metric and sends it to the video server, which then passes it on to a reinforcement learning model using a ZeroMQ socket. The client also waits for the video manifest to be sent from the server and then processes the information to control the video buffer. There are two main functions that the client uses to control the buffer: filling the buffer and consuming the buffer. These functions run in parallel to ensure a smooth video streaming experience. The client is also responsible for handling command-line arguments that determine the parameters for the streamed video. This allows the client to handle both buffered and unbuffered video, and to calculate the QoE based on the stall time and total elapsed time. The client uses the ZeroMQ library to generate the necessary sockets for communication with the video server.

REFERENCES

[1] "srsran, inc.," 2021. Available at https://www.srslte.com/.

[2] "Mosaic5g," 2016. Available at https://mosaic5g.io/.

[3] "Open5gcore." Available at https://www.open5gcore.org/.

[4] "Openairinterface," 2014. Available at https://openairinterface.org/.

[5] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[6] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck, "Network slicing and softwarization: A survey on principles, enabling technologies, and solutions," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 2429–2453, 2018.

[7] "5g network slicing." Available at https://www.viavisolutions.com/en-us/5g-network-slicing.

[8] P. Subedi, A. Alsadoon, P. W. C. Prasad, S. Rehman, N. Giweli, M. Imran, and S. Arif, "Network slicing: a next generation 5g perspective," *J Wireless Com Network*, 2021.

[9] H. K. Dureppagari, U. Dinesha, R. Wu, S. Ganji, W.-H. Ko, S. Shakkottai, and D. Bharadia, "Realtime intelligent control for nextg cellular radio access networks," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, (New York, NY, USA), p. 567–568, Association for Computing Machinery, 2022.

[10] P. Hintjens, "ømq - the guide," March 2013.

[11] A. A. M. Habiby and A. Thoppu, "Application of reinforcement learning for 5g scheduling parameter optimization," *CoRR*, vol. abs/1911.07608, 2019.

[12] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms: A comprehensive classification and applications," *IEEE Access*, vol. 7, pp. 133653–133667, 2019.

[13] A. Paszke, "Reinforcement learning (dqn) tutorial."