Appendix-Technical Report

1. Appendix-Technical Report

The code to execute these experiments was built on top of the original code of the work [6], which can be found in [1]. When the code was run for the first time following the instructions listed in the original repository several errors were found. We are now going to list all the changes to the original instructions and code in order to get this code to work. After that, we are going to explain how this code works, and how our experiments can be reproduced. We tested this on a Virtual Machine using the Ubuntu 20.10 version. However, we created a repository [2] that has a README with the changed instructions and also all the new code and changes we made, but in any case, we will explain all of those here.

1.1. Build the Environment

Let's first mention the changes to the steps listed in the original repository to make this code run, and then we will focus on what changes were done to the files in the repository. With respect to the steps, the only change was in the step of installing GO, the author mentioned installing GO using the command: **sudo snap install go**. This is not going to work because the latest version of GO does not match with the version that the author used, and the files directory and the outcome of the rest of the commands in the instructions are not expected. Further, problems can be found when the GO files are built. The solution to this problem was to install the specific version that the author used by following the first steps described in [3]. Basically, we change the way of installing GO from running: **sudo snap install go**, to executing this set of commands that appeared as described in figure 1, **the only change is that instead of installing version 1.9.2 as appear on the page and the picture, we had to change it by 1.14**.

The rest of the steps were done as described in the original repository. Let's now describe the changes to the files in the repository. After building this docker image following the previous steps and installing the correct version of GO, still, two essential problems were found:

1. The client and server implemented on GO that was build and used in the experiments was throwing an error like **/App/quic/server_mt: error while loading shared libraries: libhdf5_serial.so.103: cannot open shared object file: No such file or directory**.

2. The Dockerfile was installing the latest version of Keras and Tensorflow, which broke the original code.

For solving the first problem the solution was to replace the first line of the Dockerfile **FROM ubuntu:bionic** by **FROM ubuntu:groovy**, this solves the problem.

However the second problem was a chain of issues, the options were to try to find what was the combination of versions of Keras and Tensorflow that the author used, or try to upgrade the code to be able to work with the latest versions of Keras and Tensorflow. This implies upgrading the kernel that was being used in the jupyter notebook for Python2 to Python3 and with it also the original code, however, we realized that the majority of the code was already in Python3, and for that reason, the majority of the changes were to change all the commands that were using **python** at the beginning for **python3**. It will be really difficult to explain all the places where these changes were to be made, for that reason we have created the repository [4] with all the changes, which means that instead of pull *https://bitbucket.org/marcmolla/multi-path-scheduling-with-deep-reinforcement-learning/src/master/* you only have to pull our repository.

Even when doing the step described before of replacing the files by the ones in our repository that have all the changes is enough to fix all the problems we still are going to enumerate some of the most important changes:

1. All the notebooks and python files were reviewed and all command that was running something like **python .....** was replaced by **python3 ....**

2. Due to the upgrade to the latest version of Tensorflow, the **save_weights** method that saves the weights of the model, when a file is saved with the format **.h5f** it creates three files instead of one, that is what the

[1] https://bitbucket.org/marcmolla/multi-path-scheduling-with-deep-reinforcement-learning/src/master/

[2] https://github.com/Fidac/DeepRLMpQuic

[3] https://multipath-quic.org/2017/12/09/artifacts-available.html

[4] https://github.com/Fidac/DeepRLMpQuic

```
# Download the golang archive
$ wget https://redirector.gvt1.com/edgedl/go/go1.9.2.linux-amd64.tar.gz
# Extract the archive
$ sudo tar -C /usr/local -xzf go1.9.2.linux-amd64.tar.gz
# Add go to your PATH in .profile
$ echo "export PATH=$PATH:/usr/local/go/bin" >> ~/.profile
# Reload the .profile
$ source ~/.profile
# This should return your go version
$ go version
```

Figure 1. The set of commands to install the specific version of GO.

original code is expecting, and also none of them is called as the original code expects. We first try to find out which of the files was the one need it, then rename it to the names the original code was expecting, and then we were going to change the way the weights were being loaded in his code to the way is done with the latest version of Tensorflow, however, the not only the offline agent load the weights, it also loads the weights the online agent that is in the repository https://bitbucket.org/marcmolla/gorl/src/master/, and this agent does not use the load methods from Tensorflow, instead it parses the file and take the weights it needs to put it in the online agent, this broke this approach because the file had to be in the format the original code was expecting so that the parse works. We discover then that the way to save the weights in the latest version of Tensorflow as was in the previous versions was by saving them with the format **.h5** instead. Finally, the changes done were then to go through the notebooks and python codes and replace the creation and loading of files with the format **.h5f** for **.h5**.

3. The latest version of Tensorflow need it to use **keras-rl2** instead of **keras-rl**, this implies that we had to do also multiple changes in the docker file, changing everything it was installing for python2 and install it for python3 (this is done simply replacing every it said **python** by **python3**), then we replace the line **python -m pip install keras-rl** by **python3 -m pip install keras-rl2**, also in order to install the python3 kernel in Jupyter we need to add this command to the Dockerfile: **pip3 install Jupyter**

4. However, due to a recent update in Tensorflow and **keras-rl2**, the training of the models stopped working. To fix this we had to install specific versions of Tensorflow and **keras-rl2**. Tensorflow version 2.4.2 and **keras-rl2** version 1.0.4.

5. Finally, the latest version of Mininet demands to clean the environment among tests by hand, which implies that in the notebooks and in the training scripts we had to add the command **mn -c** among experiments so Mininet could work properly.

1.2. How the code works

In this section, we are going to describe how the code works after you successfully build the docker image following our previous steps. In here is included the original files and the new code that we added.

The code is a result of the combination of three repositories:

Original QUIC repository: https://github.com/lucas-clemente/quic-go.

GoRL repository: https://bitbucket.org/marcmolla/gorl/src/master/.

Experiments, offline agent, setup and topology: https://bitbucket.org/marcmolla/multi-path-scheduling-with-deep-reinforcement-learning/src/master/.

The first repository is the original QUIC implementation on GO, the author created a branch on this repository and focus his changes in the next files:

example/main.go This is the code of the server, the changes added were to allow the server to use multiple schedulers, among them the Deep Reinforcement Learning implemented and also the establish how the server and the agent will communicate, this means how the server sends his state to the agent, the agent returns an action and how the server interprets that action what it does.

ackhandler/sent_packet_handler.go This is the code that handles when a packet is sent, the changes add it here were devoted to gathering information to use to the

state and for future measures like Throughput, Goodput and SRTT.

scheduler.go This code was added it by the author to the repository, this code is the "Man in the Middle" between the server and the Online RL agent, and basically build an interface to communicate with the GoRL repository that contains the implementation of the Online Agent, adapting the information that the server sends to the format that the agent is expecting, and then getting the results from the agent, interpreting those results and sending the information of the results (which path to take) to the server.

These files contain the essential changes and logic that the author add it, there are more files it had to touch, that are related to these ones in the matter of configurations, the rest of the files can be checked and study reviewing the commits of the author in https://github.com/marcmolla/mp-quic/commits/master_thesis.

The GoRL repository contains the implementation of the Online Reinforcement Learning agent where to our understanding has the same deep learning model for predicting the output of the Q function as the offline agents. This code loads the weights of the offline agent and transfers them to its model that is the one that returns the predictions to which path to take according to the state sent from the server that is mediated by the **scheduler.go**.

Finally, the repository [5] contains the code for executing the experiments and also the implementation of the offline agent. Let's split the explanation of the code in the three basic modules that are also separated into folders: **mininettest, offline_agent, and noebooks**. In our repository, we respected this distribution of folders, so the explanation works for both. However, we will specify when a code was created by us before explaining.

The **mininettest** folder contains the python codes that run the flows of the client and server that use MPQUIC that was built from the branch of the author in the original repository of QUIC. This folder contains a set of python codes to run experiments where the MPQUIC client and server communicate using different schedulers, among them the Deep Reinforcement Learning one. It also contains a set of scripts that are intended to be run in the client and server get from Mininet, to set the specific parameters of the network like the delay. Let's give a brief description of the python codes files:

[5] https://bitbucket.org/marcmolla/multi-path-scheduling-with-deep-reinforcement-learning/src/master/

basicTopo.py Contains the topology of Mininet used for the experiments.

basicTest.py Contains the unit testing to check that everything is working ok, without the Reinforcement Learning Agent, the code can be run by itself or in combination with **mpTest.py**

congestion.py This code runs the experiment of an MPQUIC connection between a client and the server, setting the specific parameters like delay and include the option of peeking the scheduler (like the Deep Reinforcement Learning one) and also if you want TCP traffic at the same time.

train.bash This bash code is the training algorithm for the reinforcement learning agent. It repeats the following process. First, runs the **congestion.py** with the parameters that are set in the variable **CONGESTION_TEST** and the actual weights of the agent a few times. The results of the experiments are stored in a file called **episode.csv**. Second, the **episode.csv** is read by the code **offline_agent/trainModel.py** that update (train) the weights of the agent simulating offline the experience accumulated in **episode.csv**. Finally, those updated weights are the ones used when it returns to the first step.

congestion-fairness.py This code was added by us. Is, up to some point, a replica of the code of **congestion.py**, we keep both for versioning. This code runs the experiment of an MPQUIC connection between a client and the server, setting the specific parameters like delay, loss rate in each link and include the option of peeking the scheduler (like the Deep Reinforcement Learning one) and also if you want TCP traffic at the same time. Also, it runs tcpdump commands in both links to save information from both links about the MPQUIC and TCP flows that can be used after to do some other analysis about the behavior of the flows in each link through time.

create_packets.bash This script was added by us. It uses the saved information from the tcpdump commands used during running an experiment and extracts for each flow (MPQUIC and TCP) its results of throughput against time.

generatefairnessfigure.py This code was added by us. It takes as input the output of the **create_packets.bash** and creates plot figures that show the results.

train$_{c}hanging_{e}nv.bash$ This script was added by us. Again, is almost identical to the **train.bash** the only change is that the **CONGESTION_TEST** variable uses the **congestion-fairness.py** instead and in that variable, you can set with which delay, loss rate and

other parameters you want that the agent trains, instead of minimal delay and not loos rate by default.

The **offline_agent** folder contains the implementation of the Deep Reinforcement Learning, in this case particularly a DQN, and also the code to train this agent based on the episodes.csv that are returned from the Online Agent in the GoRL repository, we also note that all this implementation is based on the repository: https://github.com/keras-rl/keras-rl/blob/master/rl/agents/

Finally, the notebooks folder contains the logical order and commands to run the experiments and figures that the author described in [6]. These notebooks are divided into five types:

DQNAgent_10K-{delay}ms This type of notebook runs the experiments using the trained agent with a specific delay set.

DQNAgent_130K-TCP-{delay}ms This type of notebook runs the experiments using the trained agent with a specific delay set and also TCP traffic running at the same time.

DQNAgent_10K This notebook contains the logic to train the offline agent, running multiple times the experiment of **congestion.py** and using the **episodes.csv** generated. Is important to clarify that the training is done with 1ms of delay, no like the other notebooks that put a specific delay between 10 and 50ms. The agent is trained with 1ms of delay and is tested in the other notebooks with a bigger delay.

first_test-{delay}ms This type of notebook run the basic experiments from the **mpTest.py**.

Paper This notebook is the one that is run after all the previous ones (except the **first_test-{delay}ms**) were run correctly and contains the steps and logic to generates the figures of all the experiments that were then added to the paper.

There is the last notebook called **Validating** that we didn't find anything it does that the previous one doesn't.

However, in order to reduce the amount of code and also test the new experiments we added three new notebooks:

DQNAgent_experiments This notebook basically compresses all the **DQNAgent_10K-{delay}ms** type of notebooks. You only need to set the variables correctly before running the experiment and you will get the same results with only one notebook.

DQNAgent_fairness_experiments This notebook has already implemented a set of cells that you can run with the parameters you prefer and the scheduler you like and see the fairness behavior of the MPQUIC flow using your parameters and a normal TCP flow in both links.

1.3. How to reproduce experiments

In this section, we are going to explain how to reproduce our experiments using the previous code described. However, we are going to explain how to reproduce it using our repository, since we think is easier and is also the one that contains the new code for the fairness experiments.

For the case of running the Deep SARSA and Double DQN agents instead of the DQN agent, the changes are pretty simple, we modify only the **agentModel.py** inside the **offline_agent** folder, the changes were the following:

1. We add at the beginning of the file the line **from rl.agents.sarsa import SARSAAgent**
2. We comment the line: **dqn = OfflineDQNAgent(model=model, nb_actions=2, memory=memory, nb_steps_warmup=200, target_model_update=1e-4, policy=policy, enable_double_dqn=False)** and replace it by the line: **agent = SARSAAgent(model=model, nb_actions=2, nb_steps_warmup=200, policy=policy)**.

In the case of the Double DQN agent is, even more, simpler you only need to set in the original dqn initialization line: **enable_double_dqn=True**.

There is another important step to be able to generate the graphs, since the baseline used is $\bar{T}_s$, and the experiments to get those results are still giving problems in Proxmox and locally. For that reason, we unzip first the file called **resutls_paper.tar.gz**, which contains the files that have the results of the experiments done for the author to get $\bar{T}_s$. Since these values are not dependent on the agents, we do not see any harm in using those results directly. In our repository those files are already extracted in the output folder, so you don't have to do it, but is important to know from where they come from.

After these changes, we built the docker image, and then we run the notebooks in the following order:

1. We ran the notebook **DQNAgent_10K** for training the new offline agent, this could take days running depending on the amount of power your Virtual Machine has

and the amount of training time you set. After running this notebook a folder called **train_0** is created inside the **output** folder, this **train_0** folder contains multiple weights files that are the weights of the offline agent after some determined number of steps. At the end of this notebook, one of the graphs is generated that is steps against throughput. Then from the analysis of the last cells from this notebook the better weights are selected, which we believe is selected empirically taking as guide the graph that is shown in one of the cells to look which weights are the ones getting the higher throughput. Also, is important to run the rest of the cells of the notebook using the weights selected, because those results are going to be used later.

2. After that we ran the notebook **DQNAgent_experiments** to get the results of the performance of the agent which each of the specific setups of the mininet network, in our case using 10, 20, 30, 40 and 50ms of delay in the network in each case. However, here is an important problem, for running all of these experiments, one file has to be selected among all the weights files that are in the **train_0** folder, we didn't find in the code a way to select the weight file, our guess of what the author did (and is what we did) was to use the final results that are obtained in the final cells of the previous notebook (**DQNAgent_10K**) that shows the throughput of the agent according to the number of steps and peek the one with the highest throughput since that for each point (steps, throughput) plot in that graph there is a weight file generated in the **train_0** folder called **weights_[number of steps]_steps.h5**.

3. We ran the **Paper** notebook that generates all the graphs using the results from the previous notebooks. However, in this notebook is also plotted the results from the notebooks of the type **first_test-{delay}ms** that are the ones running the experiments with the usual scheduler, however, we couldn't get to run those notebooks to make the comparison for some strange reason, in our local machines was too much and in Proxmox incredibly the system breaks each time one of those notebooks was run, however, the good thing is that since we were changing only the offline agent we could compare with the previous results of the author. But well we are mentioning this here, because there are cells in the **Paper** notebook that if the results of the experiments from the notebooks **first_test-{delay}ms** weren't generated, then these cells are not going to run, but the other parts related only to the results of the Reinforcement Learning agents will work without troubles. However, the **Paper** notebook is only to give the results of the comparisons of the Aggregation Benefit metric [6] among agents and the usual scheduler.

4. We ran the **DQN_fairness_experiments** with the parameters specified in the Experiments section and each of the agents to get the fairness graph with a TCP flow in each link. In these notebooks you only have to run the experiment with the scheduler and the parameters you desire on the network. Also, the details of the parameters are in the description of each one in the **congestion-fairness.py**. However, we will add a description of each parameter at the end.

Another important detail is that among the parameters is the Reinforcement Learning model that is going to be used for the scheduler. All the trained models until the moment are in the **models** folder on our repository. You only need to pass the weight file parameter of the model you are interested to experiment.

Let's now describe each of the parameters that can be used in the **congestion-fairness.py**. This will make it easier to understand the experiments at **DQN_fairness_experiments**.

weight_file Put the path of the weight file that contains the weight of the Reinforcement Learning Agent that is going to be used as scheduler.

epsilon This parameter belongs to the definition of the Epsilon Greedy [8] concept in Reinforcement Learning that defines with which probability an agent will take the best action or will explore with other action.

rtt Is the delay that is going to be added in the network.

background-tcp Is a boolean that states if you want to run the experiment with TCP flows running together with the MPQUIC.

different_start This parameter tells the program which kind of connection you want to run first. That is, if you want it to run at the same time, set this to 0. If you want MPQUIC connection to run first, set it to 1, and if you want to run TCP connection first, set it to 2.

gap Is to tell the program how long the first connection will run before starting the second connection.

lethl Specify the loss rate in percent at the eth-0 link.

lethr Specify the loss rate in percent at the eth-1 link.

References

[1] Mahmoud Abbasi, Amin Shahraki, and Amir Taherkordi. Deep learning for network traffic monitoring and analysis (ntma): A survey. *Computer Communications*, 2021.

[2] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies*, pages 160–166, 2017.

[3] Alan Ford, Costin Raiciu, Mark Handley, Olivier Bonaventure, et al. Tcp extensions for multipath operation with multiple addresses, draft-ietf-mptcp-multiaddressed-09. *Internetdraft, IETF (March 2012)*, 2012.

[4] Abhiram Bhaskar Kakarla. Towards novel multipath data scheduling for future iot systems: A survey. *arXiv preprint arXiv:2105.07578*, 2021.

[5] Nicolas Kuhn, Emmanuel Lochin, Ahlem Mifdaoui, Golam Sarwar, Olivier Mehani, and Roksana Boreli. Daps: Intelligent delay-aware packet scheduling for multipath transport. In *2014 IEEE International Conference on Communications (ICC)*, pages 1222–1227. IEEE, 2014.

[6] Marc Mollà Roselló. Multi-path scheduling with deep reinforcement learning. In *2019 European Conference on Networks and Communications (EuCNC)*, pages 400–405. IEEE, 2019.

[7] Huaizhou Shi, R Venkatesha Prasad, Ertan Onur, and IGMM Niemegeers. Fairness in wireless networks: Issues, measures and challenges. 2014.

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[9] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[10] Hongjia Wu, Özgü Alay, Anna Brunstrom, Simone Ferlin, and Giuseppe Caso. Peekaboo: Learning-based multipath scheduling for dynamic heterogeneous environments. *IEEE Journal on Selected Areas in Communications*, 38(10):2295–2310, 2020.