Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

Advanced Computer Networking
IN2097, Winter Semester 2023/24

Project - Designing a Software Router (10 credits)

Last updated: January 15, 2024 at 6:30pm

The goal of this project is the development of a software router. We use the high-speed packet processing framework DPDK to implement a high-performance software router. To simplify the implementation of the router, we provide access to our testbed, where the software router can be developed and tested. The router will only implement IPv4 and associated protocols to simplify the router for the purpose of this exercise.

1 Academic Misconduct

We check your submissions for plagiarism. Participants violating the academic code of conduct will be excluded from the bonus system.

It is allowed and encouraged to discuss the assignment with other students. However, the programming itself has to be done by each student individually. Group work for writing the code is not allowed. Google, StackOverflow, and other Internet sources are allowed (as long as no license is violated). If your submission contains copied code, it has to be clearly marked as such, and the original source has to be referenced. For example, StackOverflow provides a share link. Use that to obtain a link, which then has to be added to your source code. In any case, you have to understand the code you submitted, which means you must be able to explain how it works.

See also code of conduct by our department:

- en: http://go.tum.de/103707
- de: http://go.tum.de/750259

2 Submission via git

For this project, we use git to handle the submissions. We use the same repository for our tutorials. If you do not have access to your repository yet, please read the tutorial instructions and perform the described steps to get access.

3 Testbed

Scientific testbeds are used to execute experiments, such as benchmarking hardware or software components. We aim to create reproducible experiments. Therefore, we automate the entire experiment workflow to minimize the chance of human error or misconfigurations during experiments. Our research group created a framework to manage testbeds called the *plain orchestrating system (pos)* [5]. pos ensures the creation of reproducible experiments using a specific experiment workflow. For the purpose of this lecture, we created a testbed providing a small network of four connected nodes (1 router, 3 clients) for every student. During the course of this project, you will get to know the pos framework, its experiment workflow, and its features.

3.1 Accessing the Management Node of the Testbed

The testbed consists of a single management node and multiple experiment nodes. Figure 1 shows the management host and the four experiment nodes used for implementing this project. The management

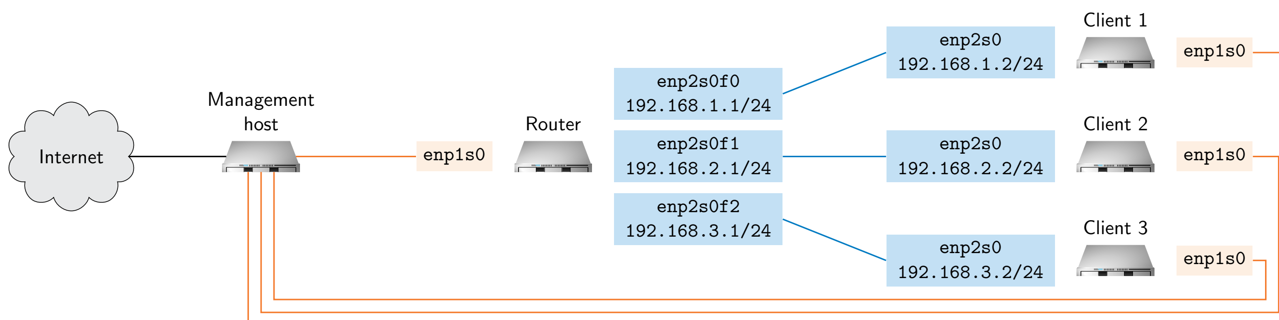

Figure 1: Testbed with management host, clients and router

node acts as the gateway to the testbed. All experiment nodes are connected to the management node via a management network (orange connections). This management network is separate from the experiment network (blue connections). This separation is necessary to avoid any impact of management traffic on measurements using the experiment network. The management node of the testbed can be accessed via SSH using the following command in Listing 1.

```
ssh -p 10022 gitlabUserID@svm0020.net.in.tum.de
```

Listing 1: Connecting to the management node (replace `gitlabUserID` with your actual user ID before trying to log in)

We use your GitLab user ID as the username and the SSH keys you uploaded to the LRZ GitLab for authentication. You can find your GitLab user ID at https://gitlab.lrz.de/-/profile. Make sure that at least one of your personal SSH keys known to GitLab is available on your local machine before trying to log in.

After logging in to the testbed host, you can access the pos CLI, the central tool to use the testbed. The pos CLI offers extensive documentation; just type in the pos command. An example can be found in Listing 9.

```
svm0020% pos
Usage: pos [OPTIONS] COMMAND [ARGS]...

  Plain-orchestrating service to access and manipulate the test nodes of this
  testbed.

  Quickstart:

  0. Look at available nodes:
      $ pos nodes list
      $ pos allocations list

  1. Allocate testnodes you want to use for one experiment:
      $ pos allocations allocate nodeA nodeB [...]

  2. Configure the nodes (per node):
      $ pos nodes image nodeA debian-stretch
      $ pos nodes reset nodeA

  3. Execute commands on the nodes (per node):
      $ pos commands launch nodeA -- echo $(hostname)

  You may use command prefixes, e.g. "pos n l" for "pos nodes list"

Options:
  --version             Show the version and exit.
  --color / --no-color
  -h, --help            Show this message and exit.

Commands:
  allocations  Define nodes taking part in an experiment.
  calendar     Testbed calendar
```

```
  commands      Execute commands on testbed nodes or roles.
  hooks         Configure hooks in posd that can be used as callbacks
  images        List available/add new images.
  jobs          Jobs (scripts) to be executed by pos at a given time
  nodes         Access testbed nodes or roles.
  roles         Group nodes into logical experiment roles.
```

Listing 2: pos CLI example on management node with help text output

3.2 Accessing the Experiment Nodes

Figure 1 shows the topology used for implementing this project. A router and three client nodes are available. The router is connected to each client via a separate connection. The router node has more RAM (8 GB) and three CPU cores compared to the client nodes, which have 4 GB of RAM and a single virtual CPU core. Every machine has a management interface called `enp1s0` to connect the respective experiment node to the management node. This interface provides SSH and Internet connectivity, i.e., this connection will not be interrupted when messing around with the other interfaces and DPDK.

Participants get their own set of experiment nodes for the project. The names of the experiments can be queried using the pos CLI, shown in Listing 3. This command outputs a table that lists the names (`ids`) of all available nodes and additional information, such as the configured image.

```
svm0020% pos nodes list

id               type  status       allocation                  image           updated

student0-client1 host  ERR booting  1685_231103_210856_811477   debian-bookworm  48h
student0-client2 host  booted       1685_231103_213724_568848   debian-bookworm  48h
student0-client3 host  unknown      None                        boot-local       6d
student0-router0 host  unknown      None                        boot-local       6d
```

Listing 3: pos command to list the available nodes of the testbed

To log in to a specific node, use the SSH command on the management node followed by the ID of the respective experiment node. Keep in mind that the node has to be booted before it can be accessed. Listing 4 shows several nodes. According to the printed output, only the experiment node with the ID `student0-client2` is currently in a booted state and can be accessed via SSH.

```
svm0020% ssh student0-client2
```

Listing 4: Logging in to a booted experiment node

To set up the nodes that are currently in a non-booted state, several steps have to be performed:

1. Allocate the respective node(s).
2. Configure an image to be booted.
3. Reset the machines to load the configured image.
4. Wait for the machine to become available.

Listing 5 lists the respective commands to perform the previously listed steps. We also provide a more extensive example script at `git@gitlab.lrz.de:acn/terms/2023ws/testbed/pos-examples.git`.

```
svm0020% pos allocations allocate student0-client1
Allocation ID: 1685_231105_214817_869983 (student0-client1)
Results in /srv/testbed/results/1685/default/2023-11-05_21-48-17_869983
svm0020% pos nodes image student0-client1 debian-bookworm
svm0020% pos nodes reset student0-client1
```

Listing 5: Allocating and preparing an experiment node with id `student0-client1`

3.3 Rebooting Experiment Nodes

Nodes can be rebooted from the management node using the command in Listing 6. This also works if the connection between the experiment node and the management node is no longer available, e.g., if the NIC driver was removed.
Important note: All experiment nodes use ramdisks as storage; resetting a node erases everything that was written to this storage. Only your home folder on the management node is permanent.

```
svm0020% pos nodes reset student0-client1
```

Listing 6: pos command to reboot a node with id `student0-client1`

3.4 DPDK Framework

To implement your router, we provide a framework, containing DPDK and a simple forwarding application based on DPDK. We use a slightly adapted version of DPDK for this project. You must use this version for all of the problems of this project to get the bonus. You can get it using the command in Listing 7:

```
git clone git@gitlab.lrz.de:acn/terms/2023ws/testbed/dpdk-framework.git
```

Listing 7: Cloning the DPDK framework repository

3.5 Template Repository

The submission follows a specific file and folder structure. To simplify the creation of this folder structure, we provide a template that can simply be merged into your own repository. To merge the folder structure into your own personal repository, execute the commands of Listing 8.

```
git remote add template git@gitlab.lrz.de:acn/terms/2023ws/template.git
git remote update
git merge --allow-unrelated-histories template/router-project
```

Listing 8: Merging the router-project branch

After merging `commit` and `push` your changes.

Problem 1 Setup **1 credit**

The deadline for this problem is **November 28, 2023, 4:00 PM**.

This exercise is designed to get to know the testbed and to set up the DPDK framework and the client machines, respectively. As part of this setup must be repeated in case of a node reboot, you create scripts to automate this step. Furthermore, a simple DPDK forwarding example is used and extended to test your setup.

Use the commands specified in Subsection 3.5 to create the folder structure required for submission.

Default route You are connected to an experiment node via SSH. Your SSH connection uses the default route installed on your local machine.

a) Why is it a horrible idea to remove this default route? Put your answer into a file named `answer.md` in the `router-project1` subfolder of your git repository.

Experiment script Create an experiment script that allocates all four experiment nodes, loads the provided parameter yaml files for the respective nodes, configures the image, and finally reboots all nodes with the configured image. After that, the experiment script should execute the `client.sh` and `router.sh` scripts in the `router-project1/node` folder for clients and router on the respective experiment nodes. These experiment node scripts are currently empty. The following subproblems will provide content for both scripts.

Hint: Have a look at the example script in `git@gitlab.lrz.de:acn/terms/2023ws/testbed/pos-examples.git`.

b) Put the final script (`experiment.sh`) in the `router-project1` subfolder of your git repository.

Clients The configuration for the client experiment nodes involves three steps:

1. Assign the correct IP addresses (see Figure 1)
2. Set the interfaces up
3. Configure the routes to their respective neighboring clients

These steps must be repeated after a reboot of a node. Create a script to automate these steps for each client individually. Use the Linux tool `ip` [1] for configuration (no credits for `ifconfig`). After executing the script, the node must still be reachable over SSH via the management interface.

The same steps must be executed on all three client nodes. However, the configured addresses differ slightly between clients. To handle this problem efficiently, pos offers the possibility to parameterize scripts on a per-node basis. Parameters can be defined in yaml files, configured on the management node (`pos allocations set_variables`), and queried on the experiment nodes (`pos_get_variable`). An example for such a configuration can be found in the example code at git@gitlab.lrz.de:acn/terms/2023ws/testbed/pos-examples.git.

c) Create a script that configures the three clients (`client.sh`, `client1.yml`, `client2.yml`, and `client3.yml`) and put them into the `router-project1/client` subfolder in your git repository.

Router The steps to set up and compile the DPDK framework can be taken from the README file in the framework repository (`git@gitlab.lrz.de:acn/terms/2023ws/testbed/dpdk-framework.git`). Keep in mind that a configuration of the interfaces via standard Linux tools is not possible after the interfaces are managed by the DPDK application. DPDK uses increasing numeric interface identifiers ordered by the PCI address of the device, i.e., the router uses the interface ids 0 to 2 if all VirtIO interfaces are bound to DPDK.

The DPDK setup must be repeated after a restart. Create a script to automate these steps so that a DPDK application can be started after running this script.

Hint: You can reboot any experiment node anytime from the management node to test your script.

Note: Interfaces that are bound to the DPDK driver will **NOT** be listed in tools such as `ip` any longer.

d) Create a script that configures the router (`router.sh`) and put it into the `router-project1/router` subfolder in your git repository.

Forwarder test To test your setup, a simple unidirectional forwarding application (`fwd`) is provided. This application takes the incoming Ethernet frames from a source interface (e.g., `-s 0`), increments the MAC source address by one and sends it via a destination interface (e.g., `-d 1`).

To see if the forwarder actually works, use the tools `ping` and `tcpdump`. Run the forwarder on the router node and try to ping the client2 node from the client1 node on `eth1`, respectively. On the client2 node, run `tcpdump` on `eth1`.

e) Which kind of packets do you expect to arrive at client2 and what kind of packets do you actually observe. Briefly explain the observation. Put your answer into the previously created `answer.md`.

Bidirectional forwarder `fwd` only forwards unidirectionally, i.e., it forwards packets from a specified source interface port to a destination interface port. Extend the forwarder example to support bidirectional forwarding, i.e., forwarding from the source interface port to the destination interface port and vice versa.

Hint: By creating a multi-threaded forwarder this exercise can be solved with less than five lines of code.

f) Update the original `fwd.c` file and put it into the `router-project1` subfolder in your git repository.

Commit and **push** the contents of the `router-project1` subfolder in your git repository.

Problem 2 Routing **4 credits**

The deadline for this problem is **December 22, 2023, 4:00 PM**.

Your router application uses two arguments for configuration. The first parameter configures interface ports (`-p`) of the router and the other argument configures routes (`-r`). The `-p` feature receives the interface id of the router interface followed by an IP address. The `-r` feature specifies routes. Routes are a tuple of a network and a simplified next hop. A network is an IP address and a netmask given in CIDR notation. We use a MAC address and destination interface id as a simplified next hop. Both arguments can be passed multiple times to configure several interfaces or routes. When passing an unknown argument, the usage of the router should be explained to the user. If everything is correct, print out the configuration on start up.

```
./router -p 0,10.0.10.1 -r 10.0.10.2/32,52:54:00:cb:ee:f4,0 -p 1,192.168.0.1 -r [...]
```

Note: The parsed arguments will be used to fill the routing table structure that is part of future subproblems.

a) Implement the command line interface in `router.c` and put the file into the `router-project2` subfolder in your git repository.

Handling multiple interfaces is best done by starting one thread per interface. Each thread handles incoming packets from a single device and sends them to the corresponding output device. Network cards (even virtualized ones) are designed for such a scenario: they offer multiple queues that can be used independently from each other. However, a queue must not be used by more than one thread.

This means we need as many transmit (tx) queues per device as we have devices: each thread has to transmit to all other devices at the same time. We only need a single rx queue per device, but our virtualization environment only supports symmetric configurations, so the device configuration function configures the same number of rx queues as tx queues. Use the provided `recv_from_device()` function to receive from multiple queues of a device. A useful mapping for tx queues is: thread 0 handles packets from device 0 and sends them to tx queue 0 of all other devices. Thread 1 handles packets from device 1 and sends them to tx queue 1 of all other devices, etc.

b) Configure the devices via `configure_device` with the appropriate number of queues depending on the requested configuration. Start a thread for each device and pass the device and queue ids assigned to it as argument. Implement your code in the `router.c` file. Put the file into the `router-project2` subfolder in your git repository.

The router shall perform the IP header validation according to RFC 1812 [3] on each IPv4 packet received. All other packets are dropped. You must free `mbuf`s of all dropped packets, refer to the DPDK documentation for further information about memory handling. We do not require ICMP responses for dropped packets. You can

assume that all links have the same MTU, handling fragmentation is not required.

c) Implement the IP header validation checks according to RFC 1812 in `router.c`, drop other packets. Do not leak memory. Put the file into the `router-project2` subfolder in your git repository.

Forwarding packets is done by looking up the destination in the routing table. We use the dummy routing table implemented in `dummy_routing_table.c`, this connects Client 1 and Client 2 via the router. Adjust the MAC addresses in the file to match the addresses of your nodes.

d) Extract the destination IP address. Look it up and forward the packets accordingly. Drop packets to unknown destinations. Handle the TTL field as described in RFC 1812 except for the ICMP time exceeded message.

To simplify the router, we do not require a full ARP implementation. The router does not need to request the MAC addresses of attached devices as the routes already specify the MAC address of the next hop. However, the router should be able to answer ARP requests by clients. Ensure that only ARP requests that are directly addressed to the router are answered, other packets must be dropped properly. **Hint:** Re-use the request mbuf for the ARP reply to simplify memory handling.

e) Implement the ARP reply in `router.c` and put it into the `router-project2` subfolder in your git repository.

f) Configure the network on Client 1 and Client 2. They should be able to ping each other. Try `ping -t 1` to send a packet that should be dropped. Use `tcpdump` on the clients to help with debugging.

Commit and **push** the contents of the `router-project2` subfolder in your git repository.

Note: We will copy your code into a fresh clone of the framework base repository for grading. This fresh clone includes DPDK that is compiled according to the README file. Make sure that it compiles and that the command line options behave exactly as specified. If you adapted other files besides `router.c` (e.g., `router.h` or `main.c`) you need to include them into your commit. You will get no credits if your code does not run, we will not debug your code for you. We will subtract at least one credit if you get memory handling wrong; this includes leaking mbufs, use-after-free bugs and double frees.

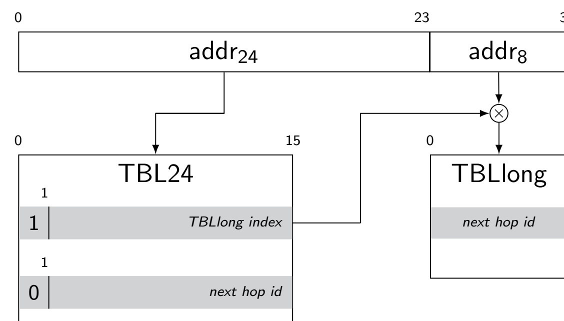

Figure 2: Table data structures used by DIR-24-8

Problem 3 Routing table **3.5 credits**

The deadline for this problem is **January 16, 2024, 4:00 PM**.

This problem can be solved independently of the previous exercises.

The routing table is one of the key data structures of routers. This data structure is necessary for answering the forwarding decisions based on the routes added. For this exercise, we want to implement a reasonably simple structure called DIR-24-8-BASIC described by Gupta et al. [6].

DIR-24-8-BASIC uses a two stage process:

- **First stage:** This stage is used for routes with a prefix of /24 or lower. The IP address to be looked up is split in two parts, the upper 24 bits ($addr_{24}$, see Figure 2) and the lower 8 bits ($addr_8$, see Figure 2). For

this stage, only $addr_{24}$ is used. $addr_{24}$ acts as an index to table TBL24. TBL24 contains 2^{24} 16 bit entries, i.e., exactly one entry for each of the 2^{24} values $addr_{24}$ can have. Such an entry in TBL24 has the most significant bit set to 0 and directly contains the next hop id.

- **Second stage:** This stage is used for routes with a prefix /25 or higher. In this case, the 2^{24} entries in TBL24 are not enough, and therefore a second table called TBLlong is used. Such a more specific entry has the most significant bit set to 1, the 15 least significant bits act as an index into the TBLlong table. This TBLlong index, together with the $addr_8$ part of the IP address, is used as an index to TBLlong to get the corresponding next hop id.

The next hop id can then be used to look up the corresponding destination MAC address and interface in a separate data structure (not part of Figure 2) For more details on the algorithm, please read the publication by Gupta et al. [6].

We provide a header file (`routing_table.h`) which must be included by the implementation of your routing table (in `routing_table.c`). The cmake file containd in the provided framework uses the dummy routing table. You need to adapt the cmake file, create a new makefile and recompile the router to use your own routing table.

To simplify your implementation, you can assume that all routes are given at the startup of the router. This means that routes are added using the `add_route()` function until `build_routing_table()` is called and no more routes can be added.
Note: Take care that adding less specific routes to the routing table does not overwrite previously added, more specific routes.

To request a matching entry from the routing table, the `get_next_hop()` function is used. This function requires the IP address of the destination and returns a `struct routing_table_entry*` containing the id of the outgoing interface and the MAC address of the next hop's interface. In case no matching next hop entry is found `NULL` is returned.

Your router does not need to address more than 254 distinct next hops, so 8 bits are enough for encoding the next hop ids. Additionally, you can assume that only 255 subnets require prefixes longer than /24, so TBLlong only needs to contain up to 255×256 entries.

Printing functions may also be implemented but are not required to get the 3 credits of this problem.

We prepared a short test file for your routing table and included the source code of the tests in the framework repository. The tests intentionally cover only very basic functionality. The analysis of data structures and identifying potential corner cases is essential for such an implementation. Therefore, we encourage you to develop your own test cases to check more advanced features or corner cases of your own implementation.
Note: The README file of the framework repository contains a description on how to compile the gtest library. Your nodes contain a compiled version of the gtest library, so you are not required to recompile the library as described in the README file but can use the version already available on your nodes.

a) Create an implementation of a routing table with the specified behavior and put your solution implemented in `routing_table.c` into the `router-project3` subfolder in your git repository. **Grading will be done by running your implementation against our own test suite, implement the header `routing_table.h` exactly as given.**

Commit and **push** the contents of the `router-project3` subfolder in your git repository.

Problem 4 Measurement — 1 credit

The deadline for this problem is **January 30, 2024, 4:00 PM**.

Note: This problem can be solved independently of the previous exercises. If you did not perform the previous exercises or your router does not work correctly, you can use the Linux router instead of your own software router. To enable the Linux router, do **not** load DPDK, activate IPv4 forwarding (`sysctl -w net.ipv4.ip_forward=1`), and configure the interfaces according to Figure 1. Please put a note in your final report stating which router implementation you used.

This problem measures the performance of the implemented router. The measurements are inspired by the RFC 2544 [4] standard that specifies benchmarks for interconnecting devices. RFC 2544 contains several key

performance indicators. For this benchmark, we investigate the throughput of the software router. Therefore, we measure the throughput, i.e., the number of packets that were successfully processed and sent out by the router. This is done by comparing the packet rate at the ingress and egress interface of our router.

We will use the iperf3 traffic generator [2] here. Do **not** use iperf2. The measurement setup is depicted in Figure 3, with Client 1 acting as traffic source, the Router as device under test, and Client 2 as traffic sink. On Client 1 and Client 2 iperf3 has to be installed (`apt-get install iperf3`). The traffic sink runs iperf3 in server mode, the traffic source runs iperf3 in client mode. Use iperf3 in UDP mode (`-u`) to be able to control the packet size with `-l`.

You can find prepared files in the template repository that define the structure to be used for this Problem. Subsection 3.5 explains how to access this template. The template repo contains the same experiment files already defined in Problem 1. The code of Problem 1 can be reused for Problem 4. For Problem 4, three measurement scripts were included (`client1-measurement.sh`, `client2-measurement.sh`, and `router-measurement.sh`) and a file to specify the loop variables used for benchmarking (`loop-variables.yml`). The new sh files will be used to specify the benchmarking process, the yml file for the investigated parameters.

Figure 3: Measurement setup

A throughput benchmark according to RFC 2544 has to be measured using different packet sizes. Because of resource constraints of our node setup, we additionally limit the maximum packet rate of the experiment. The given sizes and rates are listed below:

- Packet sizes [bytes]: 64, 128, 256, 512, 1024, 1280, 1518
- Packet rates [packets per second]: 20 000, 40 000, . . . , 200 000 (**Important:** Convert to bits/s for iperf3!)

For each of the packet sizes all given packet rates have to be measured. For this problem, both parameters should be specified as loop variables for the experiment (using the `loop-variables.yml` file). One measurement should last 30 seconds. The experiment scripts can be used to automate the entire benchmarking process, i.e., setting up the experiment nodes, starting the router and the packet generator, and uploading the result files to the management host. The iperf3 client should be started on Client 1, the iperv3 server on Client 2.

Note: We added an example to demonstrate the usage of loop variables at:
`git@gitlab.lrz.de:acn/terms/2023ws/testbed/pos-examples.git`.

a) Specify the benchmarking code and the loop variables in the respective files of the `router-project4` subfolder in your git repository.

To evaluate the measurements, plots can be created from the log files of the clients. If the log files were uploaded to the management host (using the `pos_upload` command on the experiment nodes), you can find the files at the following location: `/srv/testbed/results/{gitlab-id}/default/{timestamp}`. The `pos allocate` command prints the exact location. You can choose any file format supported by iperf3 for your log files. We added a jupyter notebook to plot the measurements to the template repository.

The measurements should be visualized using a line plot. The y-axis shows the average packet rate that was measured for an individual measurement. The x-axis of the graphs shows the different packet rates that were configured during your series of measurements. The graph must contain two separate line plots, the first one showing the rate that was actually generated by the traffic source (Client 1), the second plot showing the number of packets received at the traffic sink (Client 2). The throughput must be given in packets per second (pps). Generate a separate graph for each of the given packet sizes. In the end, your notebook should contain 7 separate graphs, one for each packet size. Each graph should contain 20 measurement points—10 points for the average generation rate, 10 for the throughput of the router—in 2 line plots. You can use the `*.loop` files generated by pos to extract the data for a specific packet size.

To evaluate the data we prepared a new image for the testbed called `debian-bookworm-evaluator`. You can select the image using the `pos nodes image` command. We recommend using the router experiment node for

evaluation as it has more RAM. The evaluator image has the Jupyter already preinstalled. To access the jupyter notebook on the router experiment nodes we recommend using an SSH tunnel with svm0020 as a JumpHost:

```
ssh -L localhost:1337:localhost:1337 -J [gitID]@svm0020.net.in.tum.de root@[stu0]-router0
```

Listing 9: SSH command to tunnel to router experiment node via `svm0020`. Replace `gitID` and `stu0` with the values of your actual setup.

To copy the result files from the management host to the router node you can use `pos nodes copy`. We recommend to create a separate experiment script that configures the evaluator image, reboots the router node, and copies the experiment results to simplify the evaluation process.

b) Add the prepared `project4.ipynb` containing the plotting scripts into the `router-project4` subfolder of your git repository.

You can use your Jupyter notebook to create a short text answering the following questions:

- Is there a trend for the beginning of packet loss?
- What is more important: the raw transfer rate in Mbit/s or the packet rate?

c) Add your text to `project4.ipynb` in your git repository.

Commit and **push** the contents of the `router-project4` subfolder in your git repository.

Problem 5 Feedback on the project — 0.5 credits

The deadline for this problem is **January 30, 2024, 4:00 PM** at 12pm (noon).

The project4.ipynb notebook cointains several questions on the project. You would help us if you could give us feedback about the project.

a) Include your answers into `project4.ipynb` of the previous subproblem. Do not forget to commit and push.

References

[1] ip(8) - Linux man page. https://linux.die.net/man/8/ip.

[2] iperf. https://iperf.fr/iperf-doc.php.

[3] Fred Baker. Requirements for IPv4 routers. *RFC1812*, 1995. https://tools.ietf.org/html/rfc1812.

[4] Scot Bradner and Jim McQuaid. Benchmarking Methodology for Network Interconnect Devices. *RFC2544*, 1999. https://tools.ietf.org/html/rfc2544.

[5] Sebastian Gallenmüller, Dominik Scholz, Henning Stubbe, and Georg Carle. The pos framework: a methodology and toolchain for reproducible network experiments. In *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021*, pages 259–266. ACM, 2021.

[6] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1240–1247. IEEE, 1998. https://doi.org/10.1109/INFCOM.1998.662938.