Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Advanced Computer Networking

IN2097, Winter Semester 2023/24

Project - Optimizing QUIC (10 credits)

Last updated: January 16, 2024 at 6:44pm

The goal of this project is the development of an HTTP/3 server and client, and to optimize the throughput of connections. Therefore, you need to familiarize yourself with the new QUIC protocol [2, 1] and use an existing QUIC and HTTP/3 library to develop a server and client. Your implementations have to be compliant with a predefined test suite that checks for several basic functionalities and allows throughput tests.

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 the Department of Informatics:

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

Contact

For questions regarding this project, please contact acn@net.in.tum.de and add `[QUIC]` to the mail subject. You can also use the respective Moodle forum to discuss problems or ideas with your fellow students.

Participation

- First, request a Gitlab repository if not done already for the exercise:
  `https://acn.net.in.tum.de/auth`
- Merge requires resources from template repository:
```
git remote add template git@gitlab.lrz.de:acn/terms/2023ws/template.git
git remote update
git merge --allow-unrelated-histories template/quic-project
```
- You are only allowed to participate in one project (QUIC or Router)

How to make clear on which project you are working?

- In the root directory of your Git repository you find a file named `project.yml`
- Add the following line to the file:

```
project: quic
```

- Git `commit` and `push` the changes
- We use the content of this file to decide which project we correct for a certain deadline
- If you do not follow these instructions, we will not correct and grade your submission

Infrastructure

Access to Repositories

We use the ACN material repository and LRZ Gitlab to provide tools, templates and the project description. You have access to the following repositories:

- Container
  In this repository we can provide docker images which can be used for example to compile your QUIC applications (Phase 2). If you want additional images or changes to the existing ones, just let us know.
- QUIC Interop Runner
  This is the current version of the Interop Runner which we will use to test your applications against each other.
- Your personal working repository.
- The templates repository with a branch (*quic-project*) for the QUIC project.

We will add the following files to the template repository:

```
└─ quic-project
   └─ implementation
      ├─ build.sh
      ├─ run-client.sh
      ├─ run-server.sh
      └─ setup-env.sh
└─ .project-ci.yml
```

files	
`.project-ci.yml`	configuration file for the CI
`build.sh`	this script is called to compile your binaries
`setup-env.sh`	this script is called to prepare the environment
`run-client.sh`	this script is called to run the client application
`run-server.sh`	this script is called to run the server application

Interop Runner

We provide you with access to the QUIC Interop Runner. It offers a tool to test QUIC implementations against each other. It requires a client and server implementation supporting multiple test cases. Each implemented server is tested against each client.

Implementations need to be added to `implementations.json` to be tested. Each implementation is expected to have a name/identifier and a path. The path has to be a directory that contains a `setup-env.sh`, `run-client.sh` and `run-server.sh` script. The server and client are executed in this directory (see CWD).

The given `implementations.json` will be used by the Gitlab CI to test your implementation. We added an example implementation to the ACN materials that supports all test cases of the project. You can use the Interop

Runner locally or on the VMs provided for this course to test your implementation as well. Therefore, you need to install all requirements, update the `implementations.json` and execute the runner. See Problem 1 for more details. You need to have a new version of Wireshark (version 3.4.2 or newer) to be able to use the runner. Wireshark available for your ACN VM (Debian Bookworm) already meets this requirement.

```
python3 run.py -d -t handshake
```

Container Repo

The Gitlab CI uses Docker container to run jobs. A default image can be specified for all jobs and individual images can be selected for each job. The CI can use publicly available container images (e.g., the official Go Docker image) or a set of images provided by us. Therefore, we offer an additional Container repository that already contains the following base images.

- `debian_bookworm`: is an image based on the default `Debian Bookworm` and contains some additional packages like `wget, make, cmake, curl, zip,...`
- `interop_runner`: is also based on `Debian Bookworm` but additionally contains the Interop Runner repository. This image will be used in the `Test` stage of the CI pipeline.

All containers specified by `Dockerfiles` in this directory are created by us and provided via Gitlab. For example, you can use the Debian Bookworm container by specifying this in your CI file:

```
image: gitlab.lrz.de:5005/acn/terms/2023ws/container/debian_bookworm
```

You have developer access to the container repository but are not allowed to push to `main` branch. If you want to suggest changes to the existing containers, or want your own custom container, please follow these steps:

1. Create a new branch
2. Push your changes into your branch
3. Create a Merge Request in the Gitlab to the `main` branch
4. The CI will check if your implementation compiles, and if this is the case we will merge your branch into `main`

If your build job takes to much time due to the installation of dependencies or tools, you can create your own image. If you create a branch at the given repository and create a merge request, we can create and publish the image for you.

Interop Matrix

We use the QUIC Interop Runner to test your implementations against other and create an interoperability matrix. The results will be published on https://acn.net.in.tum.de/interop/.

We fetch your artifacts from the build step at least once per day and test all server and client implementations with the handshake tests. Logs, output and traces are available for debugging purposes as well.

While your implementation has to successfully test against yourself and our provided implementation, additional tests might help to debug and analyze your submission.

Problem 1 QUIC and your Implementation **2 credits**

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

This exercise is designed to familiarize with QUIC and get to know a QUIC library. You will work with the QUIC RFCs and a QUIC implementation of your choice to answer the following set of questions. You will use the selected implementation throughout the remaining project. A major goal of this project is to improve the performance of a QUIC client and server. Therefore, you are not allowed to use the Python based *aioquic* library, but have to use a performant library. We suggest selecting one of the following implementations:

- lsquic (`https://github.com/litespeedtech/lsquic`)
- quic-go (`https://github.com/quic-go/quic-go`)
- quiche (`https://github.com/cloudflare/quiche`)

Throughout this exercise, you need to cite your answers with a respective source, the exact section of a specific RFC or a reference to parts of the source code (including filenames + lines). If no proper citation is available, the subproblem will be graded with **0 points**.

During the second part of Problem 1, you need to deploy the Interop Runner on your ACN VM and run a simple test. This will help you to test your implementations locally in the following problems as well.

The quic-project branch of the template repository contains a directory `quic-project1` with a pandoc and markdown template (`problem1.md`). Add and commit your answers to below questions in this file. You can use markdown syntax to structure the document and improve readability and create a pdf with the Makefile

The report is directly built by a CI job. Make sure the result is properly formatted. We will use the artifact of this stage to grade the first problem.

a) To which related protocol or set of protocols can QUIC be compared? What are the main differences?

b) Explain the difference between a QUIC frame and a QUIC packet.

c) What are QUIC transport parameters and which parameters exist?

d) Select a QUIC implementation you want to use for the remainder of the project and shortly explain your decision? If you select an implementation different to the suggestions above, please make sure it supports the final RFC and implements a wide variety of functionality. Note that HTTP/3 is used in the following steps for tests.

Identify which QUIC versions and congestion control algorithms are supported by your selected library? If multiple exist, how can you configure them?

e) Consider two applications (client and server) communicating with each other via QUIC. Explain all steps between the client sending data to the server until the server sends its reply. Consider the following

- When is the data en-/decrypted?
- How is the data split/reassembled?
- When is the data sent to/read from the UDP socket?

Interop Runner Deployment

The following subproblems will guide you through the steps to use the Interop Runner and test implementations.

The first step is to clone the repository and install all dependencies.

- Clone the QUIC Interop Runner to your local system[1]
- Copy the cloned repository to the VM either using `rsync` or `scp`
- Create a Python3 virtual environment
- Activate the virtual environment
- Install all requirements from the `requirements.txt` file

f) The HTTP/3 server under test needs to be reachable using a domain, namely `server`. In a local setup the domain has to resolve to `127.0.0.1`.

Since this domain is not part of the DNS itself, a local resolution needs to be enabled. This can be done by adding a respective entry to the `/etc/hosts` file.

Which line do you have to add to the file? Why can domain names be important for QUIC especially during the handshake?

g) The Interop Runner can test different implementations against each other. Therefore, all available implementations need to be configured in an `implementations.json`. Each implementation is identified by a name and needs at least a path to the binary/executable.

Download the example implementation from the ACN materials to your VM, unzip the archive and update the `implementations.json` accordingly. Copy the final configuration to your report.

You can test your own implementation in the following phases by updating the configuration.

h) Finally, you are able to execute the Interop Runner and test implementations. Print the output to your problem report.

[1] Note that we have access to the VMs and you don't want to copy an SSH key to the VM

Problem 2 Setup **2.5 credits**

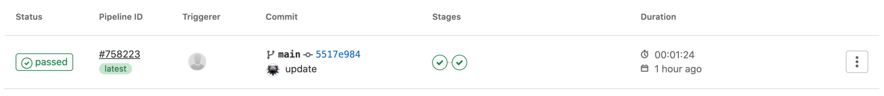
The deadline for this problem is **December 19, 2023, 4:00 PM**.

The second exercise establishes the foundation for the later project. You will create a first simple server and client and provide build scripts. Furthermore, you will familiarize yourself with the Gitlab CI. You can use your local/VM setup of the Interop Runner for tests and build your artifacts manually. However, we will use your CI to build your implementations and create an artifact for later tests. We will use this artifact to test your implementations and perform measurements in later stages of the project. Therefore, you have to make sure that the CI runs successfully for your project for further steps.

You do not have to re-implement detailed QUIC or HTTP logic. If you reuse code, make sure to understand the code and cite the source accordingly.

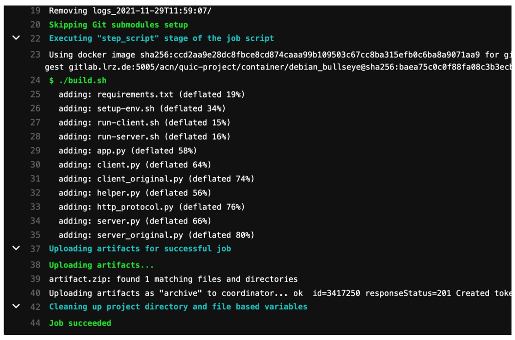
We add a general CI file (see Listing 1) to the template repository with two stages, a build and a test phase. You should not modify this file unless necessary. For example, when you want to use another image in the `build` stage.

- `build`: in this stage your binaries are compiled. Everything should be done within the `build.sh` file and all required files (e.g. binaries, your Python modules, ...) should be put into one `.zip` file called `artifact.zip`. This file will then be propagated to the next stage. Example output in Figure 1b.
- `run_test`: in this stage the artifacts from the previous stage are used to be tested against each other, additionally your implementation will be tested against an example provided by us. Example output in Figure 1c.



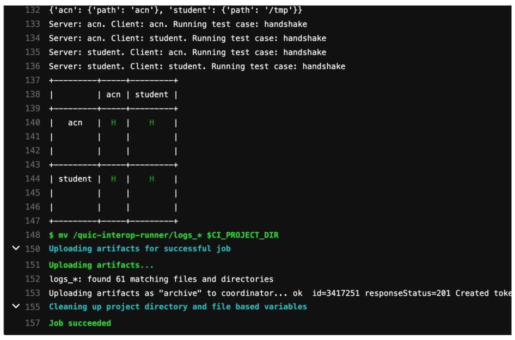
Status	Pipeline ID	Triggerer	Commit	Stages	Duration
passed	#758223 latest		main 5517e984 update	✓ ✓	00:01:24 1 hour ago

(a) Pipelines overview



```
Removing logs_2021-11-29T11:59:07/
Skipping Git submodules setup
Executing "step_script" stage of the job script
Using docker image sha256:ccd2aa9e28dc8fbce8cd874caaa99b109503c67cc8ba315efb0c6ba8a9071aa9 for gi
gest gitlab.lrz.de:5005/acn/quic-project/container/debian_bullseye@sha256:baea75c0c0f88fa08c3b3ecb
$ ./build.sh
  adding: requirements.txt (deflated 19%)
  adding: setup-env.sh (deflated 34%)
  adding: run-client.sh (deflated 15%)
  adding: run-server.sh (deflated 16%)
  adding: app.py (deflated 58%)
  adding: client.py (deflated 64%)
  adding: client_original.py (deflated 74%)
  adding: helper.py (deflated 56%)
  adding: http_protocol.py (deflated 76%)
  adding: server.py (deflated 66%)
  adding: server_original.py (deflated 80%)
Uploading artifacts for successful job
Uploading artifacts...
artifact.zip: found 1 matching files and directories
Uploading artifacts as "archive" to coordinator... ok  id=3417250 responseStatus=201 Created toke
Cleaning up project directory and file based variables
Job succeeded
```

(b) Build stage



```
{'acn': {'path': 'acn'}, 'student': {'path': '/tmp'}}
Server: acn. Client: acn. Running test case: handshake
Server: acn. Client: student. Running test case: handshake
Server: student. Client: acn. Running test case: handshake
Server: student. Client: student. Running test case: handshake
+---------+-----+---------+
|         | acn | student |
+---------+-----+---------+
|   acn   |  H  |    H    |
|         |     |         |
|         |     |         |
+---------+-----+---------+
| student |  H  |    H    |
|         |     |         |
|         |     |         |
+---------+-----+---------+
$ mv /quic-interop-runner/logs_* $CI_PROJECT_DIR
Uploading artifacts for successful job
Uploading artifacts...
logs_*: found 61 matching files and directories
Uploading artifacts as "archive" to coordinator... ok  id=3417251 responseStatus=201 Created toke
Cleaning up project directory and file based variables
Job succeeded
```

(c) Test stage

Figure 1: Output log and artifacts of the Gitlab CI can be accessed via the web interface.

```
stages:
  - build
  - test

build:
  stage: build
  only:
    - main
  image: "gitlab.lrz.de:5005/acn/terms/2023ws/container/debian_bookworm:latest"
  script:
    - cd quic-project/implementation
    - ./build.sh
    - mv artifact.zip $CI_PROJECT_DIR
  artifacts:
    paths:
      - artifact.zip

run_test:
  stage: test
  only:
    - main
  image: "gitlab.lrz.de:5005/acn/terms/2023ws/container/interop_runner:latest"
  needs:
    - job: build
      artifacts: true
  script:
    - unzip artifact.zip -d /tmp
    - mv quic-project/implementation/implementations.json /quic-interop-runner/
    - (cd /quic-interop-runner && python3 run.py -d -t handshake)
    - mv /quic-interop-runner/logs_* $CI_PROJECT_DIR
  artifacts:
    paths:
    - "logs_*"
```

Listing 1: `.gitlab-ci.yml` file you will use for your project.

a) Using the selected library from Phase 1, implement a simple QUIC server that supports HTTP/3. The server needs to use variables defined in Table 1. It has to listen on the given IP address and port and run until actively terminated. Logs, qlogs and key logs should be written to the provided directories or files. The key log will be required by tests to decrypt packet captures. Remaining logs will be attached to test artifacts and might be helpful for debugging. Parameters like IP address, port, and directory names are set as environment variables, which we note *UPPERCASE* in the description.

The server should use the `priv.key` and `cert.pem` in *CERTS* as X.509 certificate and corresponding private key and serve files from the *WWW* directory.

Specific test cases (e.g., a simple handshake) will be configured using the *TESTCASE* variable. If your server does not support a testcase, it should terminate with the return code 127. This will be checked by the Interop Runner before executing any tests. Required test cases can be found in Table 3 and for now only the `handshake` test must be supported.

The Interop Runner will start your server with the `run-server.sh` script and all variables set as environment variables. You can adapt the script to start your implementation and handle variables. You can either use them as environment variables in your implementation or use the script to transform them to command line parameters/a config file.

b) Using the selected library from Phase 1, implement a simple QUIC client that supports HTTP/3. The client needs to use variables defined in Table 2. Logs, qlogs and key logs should be written to the provided directories or files. The key log will be required by some tests to decrypt packet captures. Remaining logs will be attached to test artifacts and might be helpful for debugging.

The client will receive a list of requests as space separated *REQUESTS* variable. Request are URLs and can look like the following:

- `https://localhost:4433/index.html`
- `https://127.0.0.2:1337/asdASGaASD`
- `https://[::1]:23344/priv.key`

For each listed request, a QUIC connection should be established, the file downloaded and saved to the

Table 1: Server Environment Variables

Variable	Description
SSLKEYLOGFILE	It contains the path and name of the file used for the key log. The output is required to decrypt traces and verify tests. The file has to be in the NSS Key Log format.[1]
QLOGDIR	qlog results are not required but might help to debug your output. However, they have a negative impact on performance so you might want to deactivate it for some tests.
LOGS	It contains the path to a directory the server can use for its general logs. These will be uploaded as part of the results artifact.
TESTCASE	The name of the test case. You have to make sure a random string can be handled by your implementation.
WWW	It contains the directory that will contain one or more randomly generated files. Your server implementation is expected to run on the given port 443 and serve files from this directory.
CERTS	The runner will create an X.509 certificate and chain to be used by the server during the handshake. The variable contains the path to a directory that contains a `priv.key` and `cert.pem` file.
IP	The IP the server has to listen on.
PORT	The port the server has to listen on.
SERVERNAME	The servername a client might send using SNI. The name relates to the provided certificate and might be necessary for some QUIC implementations.

[1] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format`

DOWNLOADS directory.

Specific test cases (e.g., a simple handshake) will be configured using the *TESTCASE* variable. If your client does not support a testcase, it should terminate with the return code 127. This will be checked by the Interop Runner before executing tests. Required test cases can be found in Table 3 and for now only the `handshake` test must be supported.

The Interop Runner will start your client with the `run-client.sh` script and all variables set as environment variables. You can adapt the script to start your implementation and handle variables. You can either use them as environment variables in your implementation or use the script to transform them to command line parameters/a config file.

c) Write a build script (`build.sh`) that builds your server and client and combines all files required to execute your implementation into an `artifact.zip`. The `build.sh` script can be used to install all dependencies required to build your implementation. The `artifact.zip` needs to contain at least the scripts `setup-env.sh`, `run-server.sh`, and `run-client.sh`. This artifact will be used by the Interop Runner in the CI and by us during tests.

We offer a basic Debian Bookworm image you can use to build your implementations. If you require dependencies that are costly to install each time, you can either use publicly available containers (e.g., the official Go container) or create your own *Dockerfile* and we will build the image and provide it to you. For the latter, you need to create a branch in `https://gitlab.lrz.de/acn/quic-project/container`, commit all required files to create your image and start a merge request.

d) The `setup-env.sh` script is executed before each test by the Interop Runner. Therefore, if your implementation requires any dependencies installed or available on the system during runtime (e.g., the respective library) extend this script to install them.

e) Extend your client and server to support a simple handshake and download a file (see Table 3). The testcase is called *handshake*. The Interop Runner will create a random file in the directory provided to the server via the *WWW* variable. The server has to serve this file. The client will receive the specific request in the *REQUESTS* variable. After a successful QUIC handshake, the client needs to request this file using HTTP/3 and save the complete file to *DOWNLOADS*. After the successful download, the client should terminate with return code 0.

The Interop Runner will check for a successful QUIC handshake in a packet trace and compare the files.

Table 2: Client Environment Variables

Variable	Description
SSLKEYLOGFILE	It contains the path and name of the file used for the key log. The output is required to decrypt traces and verify tests. The file has to be in the NSS Key Log format.[1]
QLOGDIR	qlog results are not required but might help to debug your output. However, they have a negative impact on performance so you might want to deactivate it for some tests.
LOGS	It contains the path to a directory the client can use for its general logs. These will be uploaded as part of the results artifact.
TESTCASE	The name of the test case. You have to make sure a random string can be handled by your implementation.
DOWNLOADS	The directory is initially empty, and your client implementation is expected to store downloaded files into this directory. Served and downloaded files are compared to verify the test.
REQUESTS	A space separated list of requests a client should execute one by one. (e.g., https://server:4433/xyz)

[1] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/Key_Log_Format`

Problem 3 Implementation Tests **3 credits**

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

During the third phase, you need to work with your chosen implementations in more detail and extend your implemented client and server to pass all test cases listed in Table 3. You should already have successfully implemented the first test case and used the CI in the Problem 2 of the project. The new test cases are from now on available to be tested. Merge the quic branch of the template repository to merge the CI. The provided reference implementation supports all test cases and can be used to test your server and client individually.

If your selected library does not allow to implement one of the test cases, notify us and add a `README.md` file to your implementation in your Gitlab repository with an explanation and sufficient citations.

While all test cases can be implemented independently, we suggest the following order.

a) Implement the `retry` test case as stated in Table 3. To allow 1-RTT handshakes, the server needs to derive cryptography keys and keep state regarding the client after the initial packet. This could be abused by an attacker during a DoS attack. Sending large amounts of initial packets with spoofed source addresses can generate high load on the server and reduce its responsiveness. To prevent computational expensive cryptography and state on the server without knowledge whether clients are actually reachable and responsive, a server can send a retry packet. The client has to return the token to the server before a successful handshake can finish. However, this principle breaks the promise of 1-RTT handshakes and is therefore most likely only activated if the server is under attack.

To test whether your implementation is able to use the retry mechanism, your server should send a retry packet after the first initial packet. The client needs to act upon the retry packet and initiate a successful handshake.

b) Implement the `transfer` test case as stated in Table 3. During the handshake, your server is expected to accept handshakes and serve all files in the *WWW* directory using HTTP/3. The client needs to establish a single connection and send all *REQUESTS* using this connection. The testcase is successful if only a single handshake is visible and all files match.

c) Implement the `multihandshake` test case as stated in Table 3. During the handshake, your server is expected to accept handshakes and serve all files in the *WWW* directory using HTTP/3. The client needs to establish a new connection for each request in *REQUESTS*. The testcase is successful if all files match and a handshake is visible in the trace for each file.

d) Implement the `follow` test case as stated in Table 3. The server receives two files to serve from the Interop Runner. The client is only provided with the URL to the first file. After downloading the first file, the client needs to parse the content of the file and construct a second request to download the second file. The testcase is successful if both files match.

e) Implement the `chacha20` test case as stated in Table 3. The server and client should only use the `TLS_CHACHA20_POLY1305_SHA256` cipher as defined by IANA and be able to successfully finish a handshake. Afterwards, the client has to download all files.

f) Implement the `transportparameter` test case as stated in Table 3. As covered during the first phase of the project, QUIC specifies 17 transport parameters. These can be set by client and server individually. To transmit parameters during the handshake, a new TLS extension was specified. During this test, your server needs to set one of these parameters in the TLS extension. The value is *initial_max_streams_bidi* and has to be set to a value <11. You might want to test the effect of some of these parameters during the last phase of the project.

The handshake has to succeed and all files need to be downloaded by the client.

g) Implement the `resumption` test case as stated in Table 3. To resume connections, clients can use a session ticket from the server. During this test, multiple steps are required. Firstly, the client should connect normally to the server and request the first file. Additionally, it should wait for a session ticket it can use in later handshakes. Therefore, the server needs to send a session ticket to the client. After the client downloaded the first file and received a session ticket, it needs to close the first connection, establish a new connection using the session ticket and download all remaining files. Even with the session ticket, the client is not allowed to send 0-RTT data.

h) Implement the `zerortt` test case as stated in Table 3. In contrast to the resumption test, the client should use a session ticket and request following files with 0-RTT data during this test. The overall process of the test is the same as for the resumption test but 0-RTT data needs to be visible in traces.

Table 3: Test cases

Testcase	Description
`handshake`	The client requests a single file and the server should serve the file. The test is successful if there is exactly one QUIC handshake and no retries within the packet trace. Additionally, the downloaded file must be equal to the file served by the server.
`transfer`	The client needs to send multiple requests and download all files using a single connection. All files have to match and only a single handshake should be visible to pass the test.
`multihandshake`	The client needs to send multiple requests and download all files using new connections for each request. All files have to match and for each file, a handshake needs to be visible to pass the test.
`follow`	The client requests a single file from the server, which serves two files. While the first file contains the path of the second file, the second file contains random data. The client only receives one request but has to download both files by parsing the content of the first file and constructing a second request by replacing the path with the one retrieved.
`chacha20`	In this test, client and server are expected to offer **only** `TLS_CHACHA20_POLY1305_SHA256` as a cipher suite. The client then downloads the files.
`retry`	Tests that the server can generate a Retry, and that the client can act upon it (i.e. use the Token provided in the Retry packet in the Initial packet). Only a single handshake should be visible.
`resumption`	Tests QUIC session resumption (**without** 0-RTT). The client is expected to establish a connection and download the first file (first value in the *REQUESTS* variable). The server is expected to provide the client with a session ticket that allows it to resume the connection. After downloading the first file, the client has to close the first connection, establish a resumed connection using the session ticket, and use this connection to download the remaining file(s).
`zerortt`	Tests QUIC 0-RTT. The client is expected to establish a connection and download the first file. The server is expected to provide the client with a session ticket that allows the client to establish a 0-RTT connection on the next connection attempt. After downloading the first file, the client has to close the first connection, establish and request the remaining file(s) in 0-RTT.
`transportparameter`	Tests whether the server is able to set an *initial_max_streams_bidi* value of < 11 during the handshake. The client has to download all files with a single connection.

Problem 4 Performance Optimization 2 credits

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

The goal of the fourth period is to analyze the performance of your implementation, mainly the goodput. Therefore, we want to execute your implementation on two different servers in our testbed (see Figure 2) and measure the goodput in different scenarios. You have to prepare and test two measurements in the CI. We will use your artifacts, run all measurements and publish results on the Interop website.[2] Each measurement will be executed multiple times to cover result deviations due to external effects and allow to analyze the stability of your implementation.

During the measurement, `ifstat`[3] is used to measure the interface usage (see Listing 2). The output will be provided to you as part of the artifacts. Additionally, `tcpdump` is disabled during measurements.

You will use measurement outcomes from the CI and from our testbed to analyze your performance and report your results. We provide a report template in `./quic-project/quic-project4/Problem4.md`. However, you need to add some visualization of your results, interpretation and creativity regarding potential optimizations.

To be able to participate with your implementation in the measurements, you need to configure different measurement test cases in your client and server. The full list of measurements can be seen in Table 4.

The measurements will be available in your CI as well, to allow testing. However, your final evaluations should be based on results from our testbed runs. The CI executes the code in docker containers and tests implementations on localhost. Note, that the corresponding jobs are not executed automatically in the CI and require manual triggering via the Gitlab web interface.

Add your findings, a visualization of results and explanations to your report. The report is directly built by a CI job. Make sure the result is properly formatted. We will use the artifact of this stage to grade the last problem.

```
HH:MM:SS    Kbps in     Kbps out
15:57:29   1.14e+06    1.14e+06
15:57:30   587079.6    587079.6
15:57:31   497029.0    497029.0
15:57:32   387665.7    387665.7
15:57:33   516711.3    516711.3
15:57:34   417625.5    417625.5
15:57:35   486301.0    486301.0
15:57:36   502921.3    502921.3
15:57:37   481484.7    481484.7
15:57:38   205523.5    205523.5
15:57:39   544422.6    544422.6
15:57:40   17875.71    17875.71
```

Listing 2: Example `ifstat` output.

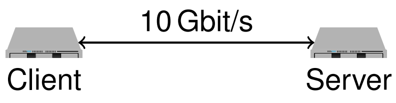


Figure 2: Testbed topology

Table 4: Measurement test cases

Testcase	Description
`goodput`	During this measurement, the client needs to connect to the server and download all files set by the *REQUESTS* variable. The client and server **are not allowed to** write qlogs. The *QLOGDIR* variable will not be given to your implementations. Note that the file size can be drastically larger compared to the `handshake` test case.
`optimize`	During the second measurement, the client needs to connect to the server and download all files set by the *REQUESTS* variable.You are free to optimize your implementation individually. Note that the file size can be drastically larger compared to the `handshake` test case.

[2]Since this requires some manual interaction from our side, it will not be executed daily and you should start early.
[3]`https://man7.org/linux/man-pages/man8/ifstat.8.html`

a) The first measurement, `goodput`, will test the goodput of your implementation. During this measurement, you need to deactivate qlog and reduce logging to important messages. The *QLOGDIR* environment variable will not be set by the Interop Runner and cannot be used by your implementation. Make sure, that qlog is disabled during this measurement, since it does impact the performance.

Your server and client need to support a simple handshake. Afterwards, the client needs to download given files, similar to the `transfer` testcase. To make sure the measurement takes several seconds, the file download will be larger compared to the `handshake` test case.

After implementing the test case, use results from the CI and Interop website to analyze your implementation, mainly the `ifstat` output. Visualize (e.g., with Matplotlib and Jupyter) and describe your findings.

b) During the `optimize` measurement you are free to optimize whatever you want and think might be useful. You can try different transport parameters, UDP settings, congestion algorithms or any other mechanism, based on your experiences with the used library. Furthermore, you could try to split the file into chunks, download them in parallel and reassemble them on the client side. You have to explain your idea(s) in your report and evaluate the outcome.

Use results from the CI and the Interop website to analyze your implementation, mainly the `ifstat` output. Visualize (e.g., with Matplotlib and Jupyter) and describe your findings.

Problem 5 Feedback on the project 0.5 credits

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

The `.quic-project/quic-project4/Problem4.md` template contains questions on the project and additional feedback questions. You would help us a lot if you could give us feedback about the project so we can improve this project in the future.

Include your answer into `./quic-project/quic-project4/Problem4.md` of the previous subproblem. Do not forget to commit and push. The report is directly built by a CI job. Make sure the result is properly formatted. We will use the artifact of this stage to grade the last problem.

References

[1] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[2] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC Transport Protocol: Design and Internet-scale Deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 183–196, 2017.