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

Advanced Computer Networking

IN2097, Winter Semester 2021/22

Project - Optimizing QUIC (4 credits)

Last updated: December 2, 2021 at 2:44pm

The goal of this project is the development of a QUIC server and client and optimize the throughput of connections. Therefore, you need to familiarize yourself with the new QUIC protocol [2, 1] and use an existing QUIC 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/854881
- de: http://go.tum.de/246064

Contact

For questions regarding this project, please contact acn@net.in.tum.de and add `[QUIC]` to the mail subject.

Participation

Due to resource constraints we can only offer a limited number of 15 places in this project. If you want to participate, please send your "application" to the address mentioned above. This can just be an informal mail which must include the following information:

- your matriculation number
- your Gitlab User ID (see below).

Without this information your application is not considered. The deadline for the application is **November 16, 2021, 4:00 PM CET**. Afterwards, we will notify you. In case more than 15 students want to participate, we will distribute the places randomly.

Prof. Dr.-Ing. Georg Carle
acn@net.in.tum.de

Benedikt Jaeger, Johannes Zirngibl
acn@net.in.tum.de

Infrastructure

Access to Repositories

For this project we will host all material repositories as well as your working directories on the LRZ Gitlab. By default you already have an account which needs to be activated by logging in. To give you access to the repositories, please send us your User ID, which you can find as follows:

- Visit https://gitlab.lrz.de/-/profile
- Main settings → User ID

Once we have your User ID, we will give you 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. 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. It is initialized with the following files

```
your_repository
├── .gitlab-ci.yml
├── docs
│   └── problem1.md
├── README.md
├── build.sh
├── setup-env.sh
├── run-client.sh
└── run-server.sh
```

files	
`.gitlab-ci.yml`	configuration file for the CI
`problem1.md`	your solution for the questions in problem 1
`README.md`	add some meta information about your project
`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. For each test case, the implementations are configured using a set of environment variables listed in Table 1 and 2.

Implementations need to be added to `implementations.json` to be tested. Each implementation is epxected 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 in `acn/` that supports the test cases for the current phase of the project. You can use the Interop Runner locally to test your implementation as well. Therefore, you need to install all requirements, update the `implementations.json` and execute the runner. 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 Bullseye) 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_bullseye`: is image is based on the default `debian bullseye` and contains some additional packages like `wget, make, cmake, curl, zip,...`
- `interop_runner`: is also based on `debian bullseye` 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 Bullseye container by specifying this in your CI file:

```
image: gitlab.lrz.de:5005/acn/quic-project/container/debian_bullseye
```

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.

Problem 1 QUIC and your Implementation 2 credits

The deadline for this problem is **November 30, 2021, 4:00 PM CET**.

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. We suggest selecting one of the following implementations:

- lsquic (`https://github.com/litespeedtech/lsquic`)
- quic-go (`https://github.com/lucas-clemente/quic-go`)
- aioquic (`https://github.com/aiortc/aioquic`)

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**.

Your QUIC project repository contains a file `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.

a) Who introduced QUIC initially and when was the first IETF draft published? What does QUIC stand for?

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

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

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

e) 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.

f) 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?

g) Which QUIC versions are supported by your selected library? If multiple exist, how can you configure them.

h) Which congestion control algorithms are implemented in your selected library? If multiple exist, how can you configure them.

Problem 2 Setup — 2 credits

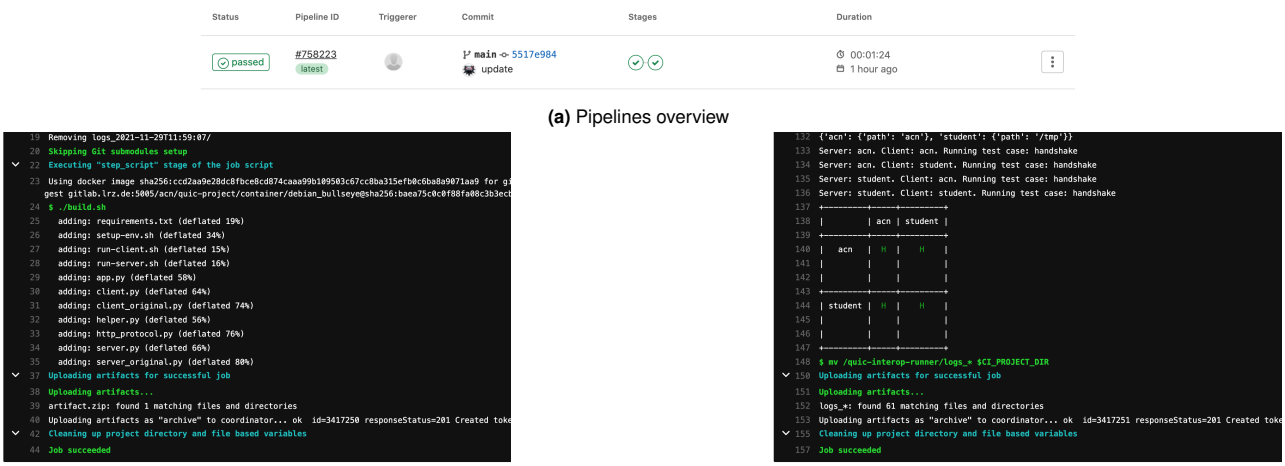
The deadline for this problem is **December 21, 2021, 4:00 PM CET**.

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 our test runner and the Gitlab CI. You will use the 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.

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 will push a general CI file (see Listing 1) to your repository with two stages, a build and a test phase. You should now modify this file unless necessary. For example, when you want to use another image in the `build` stage. The other

- `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 `artifacts.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.



(a) Pipelines overview

(b) Build stage

(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
  image: "gitlab.lrz.de:5005/acn/quic-project/container/debian_bullseye"
  script:
    - ./build.sh
  artifacts:
    paths:
      - artifact.zip

run_test:
  stage: test
  image: "gitlab.lrz.de:5005/acn/quic-project/container/interop_runner"
  needs:
    - job: build
      artifacts: true
  script:
    - unzip artifact.zip -d /tmp
    - (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 result code 127 and print "exited with 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 *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 result code 127 and print "exited with code 127". This

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 negativ 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.

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

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 `artifacts.zip`. The `build.sh` script can be used to install all dependencies required to build your implementation. The `artifacts.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 Bullseye image you can use to build your implementations. If you require dependencies that are costly to install each time, you can either use publicly avaialble containers (e.g., the offical 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 using Python) 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 clients should terminate with return code 0 and print "client exited with 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 negativ 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.
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://127.0.0.2:445/xyz)

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

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.

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.