Robust UDP Challenge

It is well known that UDP is an unreliable transport layer protocol. This lab aims to practice implementing a robust file transmission protocol using UDP. The objective is to send 1000 files from a client to a server over a lossy link. You (and your team members) have to implement both the server and the client. We do not have a spec for the protocol. You can decide how to transmit the files by yourself. The only limitation is that you must transmit the files over UDP. Once you have completed your implementations, you can upload your compiled server and client to our challenge server for evaluation. Good luck, and have fun!

The Challenge Server

The challenge server can be accessed via nc using the command:

nc inp.zoolab.org 10560

You have to solve the Proof-of-Work (PoW) challenge first. The PoW solver is available here [view].

The server then allows you to upload two binary files encoded in base64. The first one is for the server, and the second one is for the client. You ***must*** compile your programs on a Linux machine (both x86_64 and arm64 dockers are fine) and link the program with -static option. This is because your binary will be invoked on the challenge server, but no standard dynamic libraries are available on the challenge server.

We recommend you interact with our challenge server using **pwntools**. If you do not have it, install it by following the instructions here.

Once your installation is successful, you can submit your binaries using our prepared script submit.py (view | download). The usage of this script is as follows:

[TOKEN=team-token] ./submit.py /path/to/your/server /path/to/your/client

The script solves the PoW challenge from the challenge server, uploads the two binaries to the server, runs the server and the client on the challenge server, and reports results from the server. An optional environment variable, TOKEN, is used to submit your score to the scoreboard.

Note that the challenge server runs your program by passing several arguments to your programs. For the server, it is

/server <path-to-store-files> <total-number-of-files> <port>

^^^^^^^

Your program

For the client, it is

/client <path-to-read-files> <total-number-of-files> <port> <server-ip-address>

^^^^^^^

Your program

Suppose the files are stored in the **/files** directory (only on the client side), and there is a total of *N* files. Each filename is a six-digit numeric starting from zero to *N-1*. The default setting of the challenge server generates 1000 files (named from 000000 to 000999) of different sizes randomly.

Your client program should read files from the **<path-to-read-files>** and send the files to your server using UDP. Your server program should store received files in the **<path-to-store-files>** directory.

The challenge server checks the transmitted files right after your client program terminates. It then reports how many files have been correctly transmitted over the lossy link. The current settings for the lossy link emulated by **netem** is **limit 1000 delay 100ms 50ms loss 40% corrupt 10% duplicate 5% rate 10Mbit.**

Sample Files

To better illustrate how the challenge server works, we implement a **UDP echo server** and a **UDP ping client** to demonstrate how to transmit the binary executables to the challenge server and test the programs. The source codes for the two programs are available here:

You can upload any executables you like to the server and see how it behaves. Here we simply use our implemented sample files for an illustration.

- **UDP echo server**: view | download
  On receipt of a UDP packet, the UDP echo server sends it back to its sender.
- **UDP ping client**: view | download
  The UDP ping client sends a packet to a target UDP server. A sequence number and a timestamp are embedded into a packet. The echo-backed packet from the echo server can then be used to measure the round-trip time.

You can compile and generate the executables using the following commands:

gcc -static -o udpechosrv udpechosrv.c

gcc -static -o udpping udpping.c

You can then submit the two binary executables to the challenge server using the submit.py script: (Suppose all the files are placed in the same directory)

python3 submit.py ./udpechosrv ./udpping

If everything is correct. You should see messages like those in the below screenshot. Here are some interesting observations.

- The screenshot shows that the submit.py script spent about 3s to solve the PoW.
- It then uploads the two files to the challenge server.
- The challenge server shows its network settings and starts running the server and the client.
- The screenshot also shows that the measured round-trip time varies from 100-

250ms, and many packets are lost (only four are received).

- After the client is terminated, the challenge server checks the files on the server side. Since we did not perform any file transmission, all the reported numbers were zero.

You have to send 1000 files from the client to the server. Note that we have a memory usage limitation, which is currently set to 36MB. You have to ensure that everything (including the challenge server, your client/server programs, and spaces allocated by your programs) runs well. The challenge server kills your programs if the total memory usage exceeds the constraint.

The challenge server supports both Linux x86-64 and aarch64 executables. You can compile the executables in a Linux container running on a Windows or a Mac OS. However, the aarch64 executables are run with an emulator, and the performance might be dropped a little bit.

Demonstration

- [20%] You can run **udpechosrv** and **udpping** in your development environment and submitted for server verification.
- [48-90%] Your implementation can work on our challenge server and transmit at least 10 files successfully. We record the performance of your implementation. The final score is (tentatively) evaluated based on the success rate and the rank on the scoreboard. The minimum requirement to get 60% pts is to correctly transmit at least 1% of the files (that is, success rate > 0.01). Note that the challenge server *kills* your server and client programs after running it for about 600s. It means you have to finish file transmission before your programs are killed.
    - You can get **50 pts** if your implementation can transmit at least 1% of the files.
    - You can get another **20 * success rate pts** based on your implementation
    - The rest of scores are based on your performance on the live scoreboard. We plan to classify all the unique identities on the leaderboard into four classes. The top class gets all **20pts**, the second class gets **15 pts**, the third class gets **10 pts**, and the last one gets **5 pts**.

  Please note that ***you must*** demonstrate your implementation to a TA for recording your score. If you would like to know how other teams perform, a scoreboard is also available to see the success rate of all participating teams.

  The default display name on the scoreboard is your student ID. ***You have one chance to change your display name on the scoreboard.*** Please use our API to perform the change.

- Retrieve your current name:
  curl 'https://robustudp.zoolab.org/myname?token=YOUR-TOKEN'
- Change your display name:
  curl -d 'token=YOUR-TOKEN&name=YOUR-NAME' https://robustudp.zoolab.org/myname
  The token will be available in the classroom.