University College London
Department of Computer Science
COMP0023: Networked Systems

Individual Coursework 1: Reliable Transport

Due: 3rd November 2023

By design, the service provided by packet-switched networks such as the Internet is flexible to users' needs. A key factor for achieving flexibility is the support for a variety of protocols at each layer of the network protocol stack. For example, applications requiring reliability (e.g., file transfer ones) can use transport protocols like TCP, while delay-sensitive applications (such as video calling) can build upon simpler protocols like UDP.

Flexibility also translates into customisation. There is no pre-defined selection of the transport protocols that must be used by end hosts. In fact, new transport protocols keep emerging to provide better service for different types of networks (e.g., data centers), under new network requirements (e.g., faster links), and/or for specific applications (e.g., video streaming).

Overview

In this coursework, we will focus on the design and implementation of the *server side* of a custom client-server protocol guaranteeing reliable packet delivery.

Suppose that you are employed by `DistroNet`, a small company specialised in efficiently distributing files through a technologically advanced private network. Following up on a recent upgrade of `DistroNet`'s network, you are asked to rethink the server-side application protocol used to deliver text files. You will build your protocol on top of UDP, so that you can carefully select primitives and mechanisms to use in order to ensure reliability and performance.

We assume that you have access to a laptop, a desktop or a UCL machine with decent Command Line Interface (CLI), such as bash, zsh or an equivalent Windows command prompt.

As in most real deployments, you don't have full control of the software clients that will access your server, so you cannot change the interface with them, including the format and semantics of the packets received and sent out by your server. Such an interface is defined by the following simple protocol.

Protocol. Your server has a single file that transfers to clients. When the server is launched, it has to wait for messages on a certain port. A file transfer starts when a client sends a GET message to the server. The server answers to the GET message by sending the content of the file to transfer, one line of the file per packet. After receiving each packet from the server, the client sends back cumulative ACKs, carrying the sequence number of the last message received in order. To complete the transfer, the server sends a FIN message, to which the client responds with ACK FIN; then, the server terminates the connection by sending a final ACK.

The messages in the protocol have the following format, where square brackets `[` and `]`, the colon symbol `:`, the pipe symbol `|`, and keywords `GET`, `FIN`, `ACK` are reserved characters.

- Server messages

```
[client-ID] sequence-number:fileline|checksum(fileline)
[client-ID] FIN
[client-ID] ACK
```

- Client messages

```
[client-ID] GET
[client-ID] ACK sequence-number
[client-ID] ACK FIN
```

Within this protocol, you will decide which messages are sent at what time, and how to react to feedback from the clients and the network.

Note that your server will receive additional messages, including the keyword ECN, that are not part of the above protocol. We detail the meaning of those messages later in this document.

Getting Started

To get started, download and unpack the coursework archive (i.e., tarball) for your coursework. If you are using Unix as OS, you can for example do so from the command line by executing the following commands:

```
$ wget www0.cs.ucl.ac.uk/staff/S.Vissicchio/comp0023/cw1/<portico-student-number>.tar.gz
$ tar xvzf <portico-student-number>.tar.gz
```

where `<portico-student-number>` has to be replaced with your Portico student number as it appear on Moodle.

Unpacking the tarball will create a directory containing all the files needed to complete this coursework. They include: (i) a skeleton of the server in a file `server.py`, that you can use as a basis to implement your server; (ii) a simplified version of the client side in a file `client.py`, modelling the combined actions of clients and network; (iii) the file `server_file.txt` that your server will transfer to connecting clients; (iv) a file `baseline-metrics.txt` and a directory `baseline-traces/` with the output of a baseline solution your server will be evaluated against; (v) a `README.txt` file with information on how to run the client and server skeletons.

The `client.py` file implements a simple but reasonable network model, simulating possible in-network packet losses plus the effect of a bottleneck link that queues packets sent by the server towards the client. The client discards packets received out-of-order: this differs from the case of typical Internet hosts that do buffer out-of-order packets. The files in `baseline-traces/` contain packet traces as collected by `client.py` when run in combination of the non-released baseline server. In each of those traces, packets are reported in the order in which they are sent and received by the client; blank lines indicate time intervals (of generally different length) where no packets where exchanged.

The provided skeletons can be used as-is to transfer the `server_file.txt` from the server to the client, but they do so without providing reliability guarantees. You will have to modify `server.py` in order to ensure correct, network-friendly and fast file transfers, as further detailed in the rest of this document.

Stage 1: Warmup

As a starting point, you are given a skeleton for a server implementation – see file `server.py` in your coursework tarball. The skeleton implements the basics of the protocol used by `DistroNet`'s software clients. To do so, it sends and receives UDP packets, using a so-called network *socket*. We believe that the code is quite self-explanatory. If you however need more details about sockets in Python, you are suggested to consult the official Python documentation i.e., `https://docs.python.org/3/library/socket.html`.

Your first task is to understand the code you are provided with, and check its limitations. As part of this task, you will have to explain the content of the `warmup-task.txt` file included in your coursework tarball. This file contains the output of an experiment with the `server.py` and `client.py` files in your coursework tarball, where the client was given in input specific CLI parameters. It exemplifies a case where the file transfer did not succeed.

Warmup task. Familiarise with the server skeleton implemented in the file `server.py` contained in your coursework tarball. Then, explain how and why the server skeleton you are provided with led to the output in `warmup-task.txt`.

[10 marks]

- **Deliverable**. Submit on Moodle a text file named `warmup-sol.txt`. The submitted file must include a few sentences (i.e., three or four) in English, explaining why data received by the client differs from the one sent by the server.
- **Assumptions and constraints**. Be concise but precise in your explanation: clearly indicate which event prevented the file transfer to complete correctly, how you know such information, and why the server skeleton was not able to cope with such an event.

  The `warmup-sol.txt` file that you submit must be a plain text one (e.g., not a Word document or similar): it must be classified as "ASCII text" when running the command `file <filename>` on a Linux CLI.
- **Marking scheme**. Your answer will be marked according to its correctness, conciseness and completeness. For example, you will not get full marks if you don't correctly identify the event or the reason that caused the fail transfer to fail. You may also lose a few marks if your explanation is not clear, if parts of it are not technically correct, or if the submitted `warmup-sol.txt` doesn't comply with the above format.

Stage 2: Implementing the Server

Having understood where the server skeleton fails and why, you are now ready to implement a server that reliably transfers files.

Overall, your server must: (i) guarantee the correctness of every file transfer, irrespectively of network conditions and possible client errors; (ii) avoid overloading the network and sending unnecessary packets; (iii) speed up file transfer as much as possible. The former requirement is the main functional requirement for our application, while the latter two are performance-related.

Performance is the main reason why you are asked to implement the server side of a custom reliable protocol – e.g., instead of just relying on TCP. We focus on two performance optimisations.

First, your server should *fast-retransmit* packets when it receives three consecutive duplicate ACKs. Some `DistroNet`'s network devices are sometimes capable to detect packet loss: when they do so, they keep state and implement the logic to send a burst of duplicate ACKs, so that even TCP implementation can infer that a (specific) packet was lost. In your customised server, you should treat three duplicate ACKs as an explicit notification of a packet loss, which does *not* indicate congestion.

Second, in addition to fast-retransmit, your server should leverage another feature of `DistroNet`'s devices: *explicit congestion notification*, or ECN. When receiving too much traffic, `DistroNet`'s devices forward queue-exceeding packets back to senders instead of dropping them: the senders (e.g., your server) will then receive messages they previously sent, but starting with the string `ECN dropped`. When your server receives such ECN messages, it should slow down its sending rate, to avoid loading the network even more.

Main task. Design and implement a server that ensures reliable file transfer, implements fast retransmission, and avoids overloading the network by reducing its sending rate when receiving ECN packets.

[90 marks]

- **Deliverable**. Submit on Moodle one plain text file, named `server.py`, containing the Python3 source code of your server. It must be possible to run your server from command line, as follows:

  ```
  $ python3 server.py [-a <IP_address>] [-p <port_number>]
  ```

  No additional CLI parameters will be used when marking your submission.

- **Assumptions and constraints**. Your primary goal is to ensure reliable file transfers. Subordinate to that, you should aim at minimising the congestion in the network. That is, minimising the number of ECN packets has higher priority than speeding up the file transfer be kind to `DistroNet` routers!

  You can change each and every line of both `server.py` and `client.py`. However, keep in mind that:

  - you must not change the interface of the original `server.py`, namely its support for the command line parameters and prints as in the server version in your coursework tarball.
  - the `client.py` file in your coursework tarball is a version of the client that is functionally equivalent but not identical (e.g., has less checks) than the one we will use to mark your submission.
  - you will submit only your server implementation, not the client: it does make sense to change the code in `client.py` to log or add functionalities/tests to it, but always remember to keep your server compatible with the original version of the client.
  - the packets exchanged to transfer files from your server implementation must follow the syntax and semantics of the protocol described earlier in this document.

  Additional requirements for your server implementation follow.

  - your server must rely on **no external library or software** other than standard libraries included in a basic Python3 distribution. We will indeed mark your submission in an environment with only a basic Python3 installation.
  - when the client and server are run on the same machine (i.e., no network delay), your server should pass each individual test released in the coursework tarball in at most 60 seconds;
  - your server is allowed to serve one client at the time – i.e., no need to support simultaneous transfers for multiple clients. However, after any file transfer is completed, the server must be able to serve requests from subsequent clients.

- **Marking scheme**. Your server will be marked by comparing its behaviour against a baseline solution, under different conditions. More precisely, we will compare statistics collected during file transfers performed with your server against the corresponding ones for the baseline solution. The file `baseline-metrics.txt` in your coursework tarball includes a subset of the tests we will use during marking.

  Referring to the test codes used in `baseline-metrics.txt`, marks will be assigned as follows.

  - reliability despite packet loss with no bottleneck, including A1, A2 and A3 tests [15 marks], and hidden ones [10 marks].
  - correct implementation of fast retransmit irrespectively of the bottleneck size, including B1, B2 and B3 tests [15 marks], and hidden ones [5 marks].
  - congestion minimisation for network bottlenecks of fixed size, including C1, C2 and C3 tests [15 marks], and hidden ones [15 marks].
  - congestion minimisation for network bottlenecks of variable size, including D1, D2, and D3 [9 marks] and hidden ones [6 marks].

  We will assign full marks if your server complies with the above constraints and produces statistics equal or better than the baseline solutions: no difference between `server_file.txt` and file received by the client, lower number of triggered ECN packets, faster transfers, and so on. For each non-hidden test, `baseline-metrics.txt` specifies the statistics that will be considered during marking. For example, only difference between sent and received files is considered in the first set of tests.

  Partial marks will be awarded depending on the misbehaviours exposed by our tests. As a rule of thumb, you will receive most of the marks if your server complies with the above constraints, guarantees reliable delivery but performs slightly worse than the baseline solution. Note that tests are independent on each other: you *don't need* to implement the perfect solution for one test before working on the following ones.

  We will use tests not revealed in advance to ensure that your code is not an overfit for the tests you have access for. Hidden tests can for example check that your server reliably transfers files different from the provided `server_file.txt`, or that it correctly deals with events similar to those in the released tests.

  The last set of tests, including D1, D2, and D3, focus on how to deal with bottlenecks whose size varies over time, potentially increasing or decreasing during a file transfer. We release no trace for these tests. This is because it is an important *part of the coursework* for you to think about how the server should behave in these scenarios. The COMP0023 material on reliability primitives and TCP can guide you through this process. In fact, we believe that this part of the coursework is an effective way for you to deepen your understanding on the mentioned material, critically revise it, and build upon it.

- **Additional notes**. We suggest that you solve the coursework progressively, addressing the above sets of tests one at the time. The first set of tests only evaluate reliability: to pass them, your goal is only to guarantee that the file received by the client is equal to `server_file.txt`. At this stage, you can therefore ignore any performance-related aspect, such as how quickly to send packets. Once you have a server guaranteeing reliability, you can progressively add performance-related features, starting with fast-retransmit, and then focusing on different types of network bottlenecks.

  We stress that the *baseline solution is not optimal*: your server can perform better than it! In fact, one challenge that you can take after finishing the coursework is to outperform the baseline solution, or even try to implement an optimal solution for the requirements in this coursework.

Academic Honesty

You are permitted to discuss the content of lectures and assigned readings with your classmates, but you are *not permitted* to share details of the assignment, show your code (in whole or in part) to any other student, or to contribute any lines of code to any other student's solution.

All code and answers that you submit must be the product of work done by you alone.

Copying of code from student to student is a serious infraction; it will result in automatic awarding of zero marks to all students involved, and is viewed by the UCL administration as cheating under the regulations concerning Examination Irregularities.

We use copying detection software that exhaustively compares code submitted by all students from this year's class and past years' classes, and produces color-coded copies of students' submissions, showing exactly which parts of pairs of submissions are highly similar. Do not copy code from anyone, either in the current year, or from a past year of the class.