COM SCI 118 Fall 2023
Computer Network Fundamentals

Project 2: Reliable Data Transfer and Congestion Control
Due date: Dec 8th, 11:59 p.m. PT

1 Goal

In this project, we are going to design and implement a protocol that can provide the following 2 functionalities:

1. Reliable data transfer
2. Congestion control

We are hoping you will gain a better understanding towards the design principles and detailed workflow of transport layer protocol through implementing this project.

2 Working Environment

We will be hosting templates on https://github.com/AlanNPC/CS118-F23-Project-2 (currently under construction). Since C/C++ is a cross-platform language, you should be able to compile and run your code on any machine with a C/C++ compiler and BSD socket library installed. No high-level network-layer abstractions (like httplib, Boost.Asio or similar) are allowed in this project. You are allowed to use some high-level abstractions, including C++14 extensions, for parts that are not directly related to networking, such as string parsing. Note that you must program in **C/C++** for this project, rather than in Java as the textbook shows.

We have also provided skeleton code for this project; we highly recommend you to read through it before using it. **Note that although we provided the skeleton code in C, you may also submit a C++ solution if you prefer to. Also, the skeleton code is NOT a mandatory requirement. If you prefer to write your own solution from scratch, we will accept it as long as it is written in C/C++.**

This project is graded on Ubuntu 22.04 LTS (Jammy Jellyfish). You may develop the project on any compatible Unix environment (including macOS), but we highly recommend testing under the same Ubuntu environment. You may obtain a copy of Ubuntu 22.04 from its official website (https://releases.ubuntu.com/22.04/, 64-bit PC AMD64 desktop image) and install it in VirtualBox (https://www.virtualbox.org) or other similar VM platforms. We do not support Windows for Project 2 because of its different socket programming conventions and behaviors; however, you may use Windows Subsystems for Linux (WSL) (https://learn.microsoft.com/en-us/windows/wsl/) or a virtual machine. For macOS users, we also recommend testing your code on Ubuntu before submission.

3 Instructions

3.1 Project Framework

- You are required to write 2 programs, `server` and `client`. The end goal of this project is to successfully transfer a file from the client to the server, and hopefully efficiently.
- Your `client` program should be able to partition the file into smaller segments, and send the segments to the `server`. However, the link has a limited capacity, and it could be lossy. You would need to implement your own loss recovery and congestion control mechanisms to achieve reliable transfer of the file (more on this later).
- Your `server` program should be able to receive the segments from the `client`, and reassemble the segments to the complete file (this should be the same file as the one `client` sends).
- You may only use UDP sockets for communication. All protocol behaviors (e.g. error-checking, acknowledgments, re-transmissions, etc.) must be implemented at the application layer. Please note that the maximum packet size (header + payload) allowed in this project is **1200 Bytes**. If your packets exceed this limit, they will be automatically dropped by the proxy (next bullet point).
- We will use a proxy (`rdcc_proxy.py`) to simulate link behaviors, including packet loss and congestion. For this project, **all communications between the client and the server (regardless of direction) must go through this proxy; a violation of this rule will result in an automatic 0 for the project.** We have provided the `rdcc_proxy.py` code in the github repository for you to test your client and server implementation. Our proxy implementation is based on Python 3.7.3, but it should work for any python version later than 3.7.
- The topology for link simulation is as follows:
  - All modules run locally, including `rdcc_proxy.py`, `server` and `client`.
  - `client` and `server` should each use port `6001` and `6002` for packet reception, respectively. `rdcc_proxy.py` will listen on ports `5001` and `5002`.
  - `client` should always send packets (e.g., data packets) to port `5002`. `rdcc_proxy.py` will automatically forward these to port `6002` (`server`).
  - `server` should always send packets (e.g., ACK packets) to port `5001`. `rdcc_proxy.py` will automatically forward these to port `6001` (`client`).
- Another caveat worth mentioning is that you may not assume that the medium does not re-order packets. In fact, to simulate a more realistic network condition, packets that are sent at similar times are not guaranteed to arrive at the receiver in-order. Therefore, it is necessary that your design considers out-of-order packet delivery and still achieve reliable data transfer.

3.2 Reliable Data Transfer

Unreliable link:

- The proxy simulates an unreliable link. It will randomly drop packets.

You must implement mechanisms of your choice to ensure data integrity and reliability at the application layer. Consider applying techniques we have discussed in class, such as acknowledgments, sequence numbers, and timeouts.

Caveat: While you need to implement timeout for reliable data transfer, we do not recommend implementing a sophisticated RTO estimation mechanism like the one in TCP Congestion Control unless you are otherwise done with the project and willing to devote large amount of time in implementing and testing it for the leaderboard (more on this later) or for fun. Instead, since we have a fixed propagation delay of 1 second, using a reasonable fixed time-out should be sufficient.

3.3 Congestion Control

Congested Link:

- The proxy maintains a separate queue for packets in each direction between client-server communication (a queue for client to server, and another for server to client).
- Incoming packets will be placed into the queue. Packets in a queue will be forwarded at a limited rate on a FIFO (First-In-First-Out) basis.
- Each queue has a limited size. If the queue is full, new incoming packets will be dropped.
- The queue sizes and the forwarding rates are dynamic. They can change due to various factors, simulating real-world links.

You must implement strategies of your choice to handle congestion and ensure efficient data transfer despite these challenges.

3.4 Testing your `server` and `client` locally

*Please note that the details of the command line operations are subject to change. The following commands demonstrates the overall conceptual procedure for testing:

1. compile your code
2. start the link simulator: `python3 rdcc_proxy.py [arguments]`
3. start your server: `./server`
4. start your server: `./client [file-to-send]`
5. check file correctness
6. check file transfer throughput

4 Grading Criteria

Your code will be first checked by a software anti-plagiarism tool. If any plagiarism is found in your work, we are obliged to report the case to the Dean's Office. Your code will be graded based on several testing rubrics, including correctness and performance. Following is the general criteria and more details can be found when the autograder is released on Gradescope.

4.1 Reliable Data Transfer (50% of project score)

For this part, **we only check the correctness of your code**. Your server should always save the uploaded file to `output.txt`, and automatically exit after the file transfer is completed. You may get full score if your `client` can always correctly upload a file to your `server` (i.e., the file saved by the server is identical to the orginal one). We may test your code under different packet loss rates and patterns.

4.2 Congestion Control (50% of project score)

For this part, **we focus on the performance of your file transfer**. Your solution's performance is measured by the time between file transfer initiation (`client` starts) and termination (`server` exits). Your solution should first meet all requirements described in Section 4.1. If your solution does not pass the file correctness check (i.e. the files are different, which suggests reliable data transfer is not achieved), you will not receive full credit for the performance section.

We also ask you to write a concise `report.txt` file that clearly describes your congestion control design. This `report.txt` file should ideally be written within several simple sentences, no longer than half a page (Bullet points are also encouraged here). The purpose of this `report.txt` is for you to outline your general design as well as to list the techniques used. We may cross check your code with the design in your `report.txt`. If your solution implementation fundamentally differs from your design, points may be taken off.

We will set up two benchmarks for the performance part. If your solution passes the file correctness check, and performs better than benchmark A, you are guaranteed a 25/50 for the performance section; additionally, if your solution outperforms benchmark B, you are guaranteed a 45/50 for the performance section. Another note is that to get 45/50 for this section, your solution will have to implement some form of AIMD rule. Please refrain from implementing unfair rules to speed up (e.g. please do **NOT** use MIMD); if your solution uses apparently unfair rules to speed up (e.g. MIMD) and outperforms benchmark B, we cannot guarantee your 45/50 for the performance section.

We will also set up a leaderboard for the performance. The leaderboard will consist of the top 15 teams and the performance attained by the 15 teams. The leaderboard will be locked at exactly the due time, and **late submissions will not considered for leaderboard** (so please do not submit late just to optimize for leaderboard). You will be guaranteed 100/100 and extra credits (up to 10% of project score) if your team's solution makes the leaderboard by the time it locks, and the specific amount of extra credit will be evaluated based on your leaderboard performance and your solution approach.

Please note that we may test your code under different `rdcc_proxy` setting (e.g., token rate and queue size) in determining your performance. So we advise against hard-coding a fixed, fine-tuned scheme specific to the default token rates and queue sizes, as they will likely be detrimental to your final performance.

4.3 Hints & Possible Approaches for Congestion Control

4.3.1 Towards Benchmark A

One possible strategy to attain benchmark A is to implement a AIMD batch stop-and-wait scheme with dynamic batch size. To be more specific, one may first implement stop-and-wait for a batch size of N (i.e. Sender send N packets, and wait for ACKs to come back before transmitting the next batch. Receiver does not buffer packets, and only acks in-order packets. On packet loss or

out-of-order ACKs, sender retransmits the entire batch). Then, the next step is to introduce the AIMD rule with cumulative ACKs to dynamically adjust batch size (i.e. additive-increase batch size when there is no loss, and multiplicative-decrease batch size upon loss). This allows the batch size to dynamically adapt to the possible remaining queue at the proxy, and therefore alleviates overflowing the queue to avoid congestion; you may also need to slide the batch sending window if the first k packets are received to further optimize this scheme. A note in this scheme is that since batch stop-and-wait transmits the entire batch at similar starting points, which may further worsen the out-of-order transmission. One way to mitigate this issue is to introduce some amount of delay (say, 100ms) between sending each packets - this way, the packets with smaller sequence numbers will have some amount of "head start" as compared to the packets with larger sequence numbers in the batch and hence the packets with smaller sequence numbers have a higher possibility to arrive earlier than those with larger sequence numbers.

4.3.2 Towards Benchmark B

To further improve the performance towards outperforming benchmark B, a possible strategy is to modify the AIMD batch stop-and-wait scheme to behave more like a selective-repeat scheme. While you may or may not need to add separate timers for each packet in the sending window depending on your implementation, you will need to buffer received packets on the receiver's end to avoid wasting bandwidth on unnecessary retransmissions (e.g. out-of-order packets that arrive at receiver, but discarded by the AIMD batch stop-and-wait scheme). A good optimization here is to leverage duplicate ACKs as we have learnt in the class. Upon getting duplicate ACKs, the sending window size should decrease by a factor of your choice, and upon time outs, the sending window size should be reset to the initial window size.

The intuition behind this scheme is that selective-repeat can pipeline packets, instead of waiting for the entire round-trip time of a batch and then deciding whether to send the next batch or not. This more fine-grained approach not only helps to make your solution more robust with out-of-order packets, but also utilize the idle bandwidth in the batch stop-and-wait scheme, hence achieving a better performance.

Again, while it is also acceptable to use alternative designs other than the above scheme, **we do require some form of AIMD implementation to get 95/100**. Please refrain from using unfair rules for optimization (e.g. do **NOT** implement MIMD to speed up your performance).

4.3.3 Towards Further Performance

Additionally, if you wish to optimize your solution's performance even further, one possible mechanism to consider is the slow-start mechanism. Upon startup or timeout, instead of using the additive increase, you may consider using an exponential slow start mechanism until a certain threshold like we learnt in class. This should in theory further improve your performance.

4.3.4 Disclaimer

We would like to clarify that the schemes listed above are just recommendations on approaching the congestion control performance requirements, so you are **not required to** follow either scheme if you do not wish to. In addition, while we recognize that a good implementation of these two schemes will outperform the benchmarks, your solutions' exact performances still largely depend on your implementation details, so we cannot guarantee that your solution will outperform the benchmarks just because you are following the top level ideas we provided here. However, our hope

is that these two schemes can provide you a good starting point to optimize your congestion control performance, and eventually attain an ideal result.

It is also worthwhile to mention that these two schemes are **NOT** necessarily built on top of each other. While implementing stop-and-wait and then extending to selective-repeat is feasible, it may not be the most efficient route for the project if you are confident about implementing the selective-repeat scheme directly. Again, we would like to stress that these are just recommendations, you are more than welcome to devise creative ways or alternative means of congestion control if you want to - as long as your solution passes the file correctness check and outperform the benchmarks, you will be given credit.

5 Summary & Overall Roadmap

This section is intended as a brief summary of the project, and does **NOT** include all the details. For detailed description for each of the referenced items here, please refer to the previous sections.

Below is a brief score break-down along our recommendations in trying to achieve the corresponding score. While we cannot guarantee your solution will achieve the corresponding score if your solution uses the same high level idea as our recommendation, our hope is that these recommendations will set a good starting point for you to be close to the benchmarks, if not out performing them.

- **Correctness score (50/100)**: Our recommendation is to implement sequence numbers and retransmission mechanism to achieve reliable data transfer.
- **Performance Benchmark A (75/100)**: Our recommendation is to implement an AIMD batch stop-and-wait mechanism. You will likely need to implement cumulative ACK and variable for AIMD window control. In addition, you may also consider adding a small delay between each packet to mitigate out-of-order packet transmission.
- **Performance Benchmark B (95/100)**: Our recommendation is to implement an AIMD selective-repeat mechanism. You will likely need to implement fast retransmit mechanism upon duplicate ack and receiver buffer.
- **Further Performance (up to 110/100)**: Our recommendation is to implement a slow-start mechanism on top of your existing solution. Other than this, you may apply any knowledge you have learnt in class or use your creativity to devise smart ways to improve performance to get higher up on the leaderboard. Good luck :)

6 Project Submission

The project is due at 11:59 p.m. on Friday, Dec. 8th on Gradescope. Late submission is allowed by Monday, Dec. 11th (10% deduction on Saturday, 20% deduction on Sunday, 40% deduction on Monday, and no credit afterwards).

You need to submit a zip archive that contains all the source code, a `Makefile` that can compile your server program, and a brief `report.txt` file that clearly describes your congestion control design (Please try to keep it within one paragraph/no longer than half a page; bullet points are also welcome). The `Makefile` should be located in the root directory within your submission. The `make` command will compile the program and **produce two executable named `server` and `client`**.

As mentioned, your programs will be graded by an autograder on Gradescope. You will be able to see part of the test result after submission. We don't limit the number of submission attempts, and the final grading is determined by your *last* submission.