Assignment 2 - Implementing Reliable Transport

Network Centric Computing - Spring '25

Lead TAs: Ayesha Mirza, Danish, Eman, Faheem, Khawaja Gul, Muiz, Namwar

Due Date: 11:59pm, 26th March, 2025

Contents

Disclaimer on Plagiarism and AI Tools

Academic integrity is paramount. Any form of plagiarism, including but not limited to copying code from online sources, unauthorized collaboration, or using AI-assisted tools such as GitHub Copilot, ChatGPT, or similar, is strictly prohibited.

All submissions will be analyzed using `MOSS` and other plagiarism detection tools. If plagiarism or unauthorized AI-assisted work is detected, the case will be forwarded to the **Disciplinary Committee (DC)** for review and appropriate action, which may include academic penalties or further disciplinary measures.

"AI wrote this for me" is not a valid justification. Submissions should reflect **your own** understanding, problem-solving ability, and effort. While AI tools can be useful for learning, relying on them to generate solutions undermines the purpose of the assignment and will be treated as academic misconduct.

Ensure that all work submitted is your **own** and adheres to academic integrity policies. If you have any doubts about what constitutes plagiarism or unauthorized AI use, seek clarification before submission.

1 Assignment Overview

In this assignment, you'll implement a reliable transport protocol. You will be writing logic for a message sender and a message receiver that communicate over UDP (an unreliable protocol by default), but reliably (which is your task). **The assignment must be done individually, without the use of AI models, and you are required to use Python.**

1.1 Queries

You should post any assignment-related queries **ONLY** on Slack. Please avoid emailing the course staff with any assignment-related queries. This will ensure that everyone benefits from the answers to your queries.

1.2 Grade Distribution

This assignment is worth **12.5%** of your final grade.

It will be graded out of 35 points. The distribution of points is as follows:

- **BasicFunctionalityTest:** Connections are correctly set up, and data messages can be exchanged between clients. (5 points)
- **PacketLossTest:** Packets are reliably received, even when there are losses inside the network. (5 points)
- **OutOfOrderPacketsTest:** Packets are received in order by the end host even when the packets are reordered by the network. (5 points)
- **DuplicatePacketsTest:** Your code handles receiving duplicate packets from the network (5 points)
- **WindowSizeTest:** You set and adapt the sending window correctly, and your implementation is resilient against packet corruption. This should follow the `Selective Repeat Algorithm` only. Anyone who passes the less rigorous version will be awarded **5 points** while those who pass both the less and more rigorous will be awarded **10 points**. (10 points).

  Manual checking will also be done to ensure your implementation conforms to the implementation guidelines. Failure to follow them will result in 0 marks for this test case. **The test cases will be ran 3 times and you will be awarded average of all 3 runs.**

- **Code Quality:** Following good code practices as described in section 1.5 (5 points)

Disclaimer

Please note that your assignments will be tested on Docker containers with a Ubuntu image as part of GitHub Workflows. As such, it is recommended that you run your code on it at least once before submitting it. It is the student's responsibility to ensure that their code works correctly on Docker containers and GitHub Workflows. Additionally, when the code is pushed to GitHub, they should verify the output of the autograder workflow to confirm that there are no errors.

"It works on my machine" is not a valid response. If your code works on some machines but fails elsewhere, it is likely that the code has not been properly written, tested, or designed for compatibility. Well-thought-out, thoroughly tested code should perform consistently across environments, especially within Docker containers, and it is **YOUR** responsibility to ensure that.

1.3 Submission Instructions

You will submit your assignment via GitHub. Follow these instructions carefully:

1.4 Submitting the Assignment

1. Ensure all your code is pushed to GitHub before the deadline. Pushing your work to GitHub correctly is your responsibility.
   - If you do not push your final submission to GitHub, your assignment will **NOT** be graded.
2. Only the submission on the `main` branch will be graded.
3. No separate submission on LMS is required; your latest pushed commit will be considered for grading.
4. The staff will evaluate the code using the test suite and automated scripts.
5. Ensure that your latest work is visible on GitHub before the deadline.
6. Navigate to your repository URL and verify that your latest commit is present in the `main` branch before submission.
7. The time of your last commit will determine whether your grace days will be used or not, depending on whether your last commit was before or after the deadline.
8. Late days will only count if you push a new commit to GitHub after the deadline. Running a workflow does **not** count as using your grace days, as it is just like running test cases.
9. The staff will fetch the latest workflow output from each repository to extract the marks.
10. When you push new code to your GitHub repository, you should **not** rerun an existing job. That will use an older version of your code. Instead, you should run the workflow from scratch using the `Run Workflow` button in the Actions tab.
11. You do not need to commit something to run the workflow. You can manually trigger the workflow run without making a new commit, as demonstrated in the Git tutorial, granted that the workflow will then run on your last committed code that is available in your remote GitHub repository.

1.5 Code Quality

Your code will be evaluated using automated tools as well as a manual review based on the following criteria:

- **Maintainability, Modularity, and Complexity:** Your code will be analyzed using tools like `Radon` and `Pylint` to assess maintainability, modularity, and cyclomatic complexity. The grading will follow the detailed breakdown provided in Section 6.
- **Naming Conventions:** You are expected to follow `PEP 8` naming conventions. You can find the standards here. We will specifically look at:
  - **Variables, Functions, and Methods:** Use lowercase letters and separate words with underscores. For example, `my_variable`, `calculate_sum()`.
  - **Constants:** Use uppercase letters and separate words with underscores. For example, `MAX_SIZE`, `PI`.
  - **Classes:** Use CamelCase (capitalize the first letter of each word without underscores). For example, `MyClass`, `CarModel`.
- **Code Documentation and Comments:** Your code should be well-documented, following the comment ratio guidelines. The comment percentage will be evaluated as described in Section 6, ensuring that your code is understandable to others.

- **Code Modularity and Reusability:** Your implementation should not be monolithic; instead, break down your code into smaller reusable functions. This will be assessed as part of the modularity score in the automated quality check.
- **Readability and Formatting:** Your code should be readable with meaningful variable and function names. Additionally, proper indentation and spacing should be maintained to improve readability.

The final quality score will be calculated using automated scripts. Ensure that your code adheres to these best practices, as they are essential for writing maintainable and efficient software.

2 GitHub Setup and Assignment Submission

The assignment will be managed and graded via GitHub. Follow the steps below to set up your repository and submit your work.

Important Warning

The `.github` folder is a protected path. Modifying any files within this folder, including workflows and configuration files, is strictly prohibited, and GitHub classrooms will flag such repositories for us. Any unauthorized changes to this folder will result in your assignment being discarded. Ensure that you do not tamper with the contents of the `.github` directory.

2.1 Working on the Assignment

1. Implement the required functionality by modifying the provided files (**reliable_transport.py** should be modified only).
2. Regularly commit your changes using git with the following commands or by Github Desktop:

```
git add .
git commit -m "Implemented feature XYZ"
git push origin main
```

Ensure that your latest work is always pushed to the repository before the deadline.

Important Guidelines

- You are only allowed to use the imports explicitly mentioned in the assignment instructions. **Any additional imports will result in an automatic score of `0`.** Ensure that your submission strictly follows the allowed imports list.
- This time, **no extra lines** should be present in the output files. The order of the output does not matter, but the content must strictly adhere to the expected format.

2.2 Running Tests on GitHub Workflows

GitHub Actions is set up to test your code. To check your test results:

1. Navigate to your repository on GitHub.
2. Click on the **Actions** tab.
3. Pick the **PA2 Autograding Runner** on the left side.

4. Press the **Run Workflow** button on the right side.
5. Refresh your page (You may need to do this a few times)
6. Look for the latest workflow run.
7. Click on it to view detailed logs of the test results, including your marks, by toggling open the step **Run TestHarness**.
8. You can also view your marks in the workflow artifact `run_output.txt`, which you can find at the bottom of **Summary** tab on the left after clicking on a workflow run.

If your tests fail, fix the issues locally and push your changes again.

3 Introduction

Since the Network layer does not make any guarantees regarding the delivery of IP packets, a reliable transport protocol is needed to ensure reliable delivery of packets. You used such a reliable protocol—TCP—in programming assignment 1. This freed you, the application developer, from having to worry about making your chat application resilient against network layer losses, packet corruption, packet delays, etc. In this assignment, you will be writing the transport logic that makes this possible.

First, let's revise some of the issues that a reliable protocol needs to deal with and their remedies.

- **Packet Loss:** Packets may be lost on the Internet. Packets may get lost on the network, by e.g. a router dropping incoming packets if it does not have sufficient available buffer to store it.

  Remedy: A reliable protocol's receiver sends acknowledgments for received packets back to the sender. In case of no acknowledgment within a timeout value, the sender retransmits the packet.

- **Packet Corruption:** Factors such as noise/interference on the links can cause packets to be corrupted.

  Remedy: The sender appends a checksum value to each packet to detect corruption. Upon receiving a packet, a receiver computes the packet's checksum value and compares it to the checksum value included in the packet. If they do not match, the receiver will discard the packet and not send an acknowledgment for it.

- **Reordering of Packets:** There is a possibility that packets do not reach the receiver in the same order as they were sent.

  Remedy: The receiver uses the sequence of packets to find the correct order of incoming packets.

- **Duplicate Packets:** Dealing with the packet losses can introduce another problem—packet duplication. Packet retransmissions can cause a receiver to receive the same packet more than once.

  Remedy: The receiver sends an acknowledgment for the duplicate packet and does nothing else as it already has a copy of the duplicate packet.

For more details, revisit class lectures on reliable transport and congestion control. In this assignment, we will be implementing a reliable transport protocol ourselves to develop a better understanding and appreciation of reliable protocols like TCP.

4 Provided Code

4.1 Everything at a glance

This assignment is based on programming assignment 1. However, instead of using a TCP socket to communicate, `server.py` and `client.py` use a class called `ReliableSocket` implemented in `reliable_socket.py`. `ReliableSocket` uses UDP, an unreliable protocol, for sending/receiving messages.

- Whenever `ReliableSocket` has to send a message to a destination, it uses a `ReliableMessageSender` (defined in `reliable_transport.py`) instance to reliably send that message to the destination.
- For each incoming message, `ReliableSocket` initializes a `ReliableMessageReceiver` (defined in `reliable_transport.py`) instance to reliably receive the message from the sender.

Your task is to implement the `ReliableMessageSender` and `ReliableMessageReceiver` classes in `reliable_transport.py`. You should not modify any other file except `reliable_transport.py`.

Implementation can be done on any platform e.g., Windows, macOS, or Linux. Testing should be done on Docker containers or on Github workflows.

4.2 ReliableMessageSender

This class reliably delivers a message to a receiver. You have to implement the `send_message` and `on_packet_received` methods. You can use `self.send(packet)` method to send a packet to the receiver.

Method Descriptions:

- `__init__(self, ..., window_size: int)`

  This is the constructor of the class where you can define any class attributes. `window_size` is the size of your message transport window (the number of in-flight packets during message transmission). Ignore other arguments; they are passed to the parent class. You should immediately return from this function and not block.

- `send_message(self, message: str)`

  This method reliably sends the passed message to the receiver. This method does not need to spawn a new thread and return immediately; it can block indefinitely until the message is completely received by the receiver. You can send a packet to the receiver by calling `self.send(...)`. Detailed implementation details on how to reliably send a message are provided in section 5.2.1.

- `on_packet_received(self, packet: str)`

  This method is invoked whenever a packet is received from the receiver. Ideally, only ACK packets should be received here. You would have to use a way to communicate these packets to the `send_message` method. One way is to use a queue: you can enqueue packets to it in this method, and dequeue them in `send_message`. You can also use the timeout argument of a queue's **dequeue method** to implement timeouts in this assignment. You should immediately return from this method and not block.

4.3 ReliableMessageReceiver

This class reliably receives a message from a sender. You have to implement the `on_packet_received` method. You can use `self.send(packet)` to send a packet back to the sender, and will have to call `self.on_message_completed(message)` when the **complete** message is received (where `message` is the original message reconstructed from its individual packets).

Method Descriptions:

- `__init__(self, ...)`

  This is the constructor of the class where you can define any class attributes to maintain state. You should immediately return from this function and not block.

- `on_packet_received(self, packet: str)`

This method is invoked whenever a packet is received from the sender. You have to inspect the packet and determine what to do. You should immediately return from this method and not block. You can either ignore the packet, or send a corresponding ACK packet back to the sender by calling `self.send(packet)`. If you determine that the sender has completely sent the message, call `self.on_message_completed(message)` with the completed message as its argument. Details on the role of a reliable receiver are provided in section 5.2.2.

5 Implementation Instructions

5.1 Packets

You'll need four types of packets in this assignment: `"start"`, `"end"`, `"data"`, and `"ack"`. All packets will also have a sequence number and a checksum. If the packet is a data packet, it will also contain the message content. The packet format you have to follow is:

```
"<packet_type>|<seq_num>|<msg_content>|<checksum>"
```

An example start packet would be:

```
"start|4212||1947410104"
```

Note that `<msg_content>` is empty as this is a *control* packet, not a *data* packet (which contains at least a part of the message to be sent). A common example of *control* packets you've studied in class is ACK packets.

Moreover, for the message `"request_users_list"`, with a chunk size of 10 bytes (recall that one character is encoded in 1 byte), the *data* packets would be:

```
"data|4213|request_us|2826007388"
"data|4214|ers_list|993936753"
```

You can use `util.make_packet(...)` to make a packet. It accepts `packet_type`, `seq_num`, and `msg_content` as arguments and returns a packet in the correct format. The returned packet also contains the checksum (so you don't have to worry about calculating checksums yourself). The following are some examples of using this function:

```
data_packet = util.make_packet("data", 2, "request_us")
start_packet = util.make_packet("start", 1)
ack_packet = util.make_packet("ack", 2)
end_packet = util.make_packet("end", 3)
```

To validate the checksum, you can pass your packet to `util.validate_checksum(...)`, which returns true/false accordingly. You can also use `util.parse_packet(...)` to parse a packet into its individual components (packet type, sequence number, data, and checksum).

5.2 Implementing Reliable Transport

A message transmission will start from a `start` packet, and end with an `end` packet. You will be sending packets using a **sliding window mechanism**, specifically the **Selective Repeat** implementation of it. For every received packet, the receiver will send a **cumulative ACK** with the sequence number it expects to receive next.

You might find these resources for understanding Selective Repeat Protocol helpful:

- Selective Repeat Slides
- Selective Repeat Animation

5.2.1 Sender Logic

1. Break down the message into `util.CHUNK_SIZE` sized chunks.
2. Choose a random sequence number to start the communication from.
3. Reliably send a start packet. (i.e. wait for its ACK and resend the packet if the ACK is not received within `util.TIME_OUT` seconds.)
4. Send out a window (window size is passed to you in the constructor of `ReliableMessageSender`) of data packets and wait for ACKs to slide the window appropriately.
5. If you receive no ACKs for `util.TIME_OUT` seconds, resend all the packets in the current window
6. Once all the data chunks have been reliably sent, reliably send an end packet.
7. As the above points imply, you only need to be sending a window of `data` packets. Sending `start` and `end` packets in a window is not allowed and will result in your test cases failing.

5.2.2 Receiver Logic

1. When you receive a packet, validate its `checksum` and ignore it if it is corrupted.
2. Inspect the `packet_type` and `sequence_number`.
3. If the packet type is `"start"`, prepare to store incoming chunks of data in some data structure and send an ACK back to the sender with the received packet's `sequence_number + 1`. This is essentially the sequence number of your next expected packet.
4. If the packet type is `"data"`, store it in an appropriate data structure, but only if it is not a duplicate of a previously stored packet.
5. After storing the packet, send a **cumulative ACK** with a sequence number equal to `(highest contiguous sequence number received + 1)`.
6. For example, if the highest sequence number received **in order** so far is 10, the ACK sent should be for sequence number 11.
7. Since packets may arrive out of order, always send an ACK based on the highest contiguous sequence number received, not just the most recently received packet's sequence number.
8. This ensures that the receiver only acknowledges up to the last sequentially received packet, preventing gaps in the data stream.
9. If the packet type is `"end"`, assemble all the stored chunks into a message, call `self.on_message_received(message)` with the completed message, and send an ACK with the received packet's `sequence_number + 1`.

6 Testing

There are two ways to test your implementation of the assignment. The first is to use the GitHub autograder, which will be used to grade your assignment. The second is to test your implementation yourself.

To test your submission, we will run the following command in your assignment directory

```
$ python3 TestHarness.py
```

7 Starter Code

You have been provided with the following files in the Code directory:

```
PA2
├── docker-compose.yml
├── README.md
├── Dockerfile
├── Code
│   ├── client.py
│   ├── server.py
│   ├── util.py
│   ├── reliable_socket.py
│   ├── reliable_transport.py
│   ├── TestHarness.py
│   ├── ReviewCodeQuality.py
│   ├── requirements.txt
│   └── Tests
│       ├── BasicTest.py
│       ├── BasicFunctionalityTest.py
│       ├── DuplicatePacketsTest.py
│       ├── OutOfOrderPacketsTest.py
│       ├── PacketLossTest.py
│       └── WindowSizeTest.py
└── Manual
    └── PA2.pdf
```

You will write code in the following files only:

- `reliable_transport.py`: This file contains the implementation of your UDP-based reliable transport protocol.

8 Running the Code on Docker

A **Dockerfile** is included for consistent testing. Use the following steps:

8.1 Build and Run Container

To build and run the container, use the following command:

```
docker compose run --rm netcen-spring-2025
# or
docker-compose run --rm netcen-spring-2025
```

What this command does:

- `run`: Executes the `netcen-spring-2025` service defined in the `docker-compose.yml` file as a temporary, one-off container.
- `-rm`: Automatically removes the container after the task completes, ensuring no leftover containers clutter the system.
- This command starts the container and runs the service or command specified for `netcen-spring-2025` in `docker-compose.yml`.

8.2 Access Code

- The code is mounted at `/home/netcen_pa2` inside the container.
- This allows you to access and work with your files directly within the container.

9 Tips on Getting Started

Here is some advice for this assignment.

1. Start early. Read the handout carefully.
2. Google is your friend. Google can give you answers to many questions faster than any TA can. Questions ranging from Bad File Descriptors to string parsing to iterating over lists; these can be answered in less than a minute with a few simple keywords being searched for.
3. Use the helper functions provided in `util.py`.
4. Your server code **must not crash** at any point unless explicitly being told to do so. Make sure you have done exceptional handling to preclude failures.

Good Luck, and Happy Coding!