Western Science

ASSIGNMENT 03

GO-BACK-N

Due Date:
November 14, 2024 23:55:59

EXPIRED

ACADEMIC DISHONESTY

Assignments will be run through a similarity checking software to check for code that looks very similar to that of other students. Sharing or copying code in any way is considered plagiarism (Academic dishonesty) and may result in a mark of 0 on the assignment and/or reported to the Dean's Office. Plagiarism is a serious offence. Work is to be done **individually.**

If you want to store a PDF version of this assignment, press Ctrl+p *on Windows or* Command+p *on Mac, the print window will appear. Then, select* Save As PDF *from the* **Destination** *dropdown. Then click* Save

This file was last modified on 2024-11-14 11:16 am ET

UPDATES and CHANGES

The following are the updates/changes made to this assignment AFTER it was posted:

No changes have been made!

Announcements and Important Notes

The following are the announcements and important notes for this assignment AFTER it was posted:

No announcements have been made!

TOP

1 Introduction

In this assignment, you will practice the following networking concepts:

- Implementing the Go-Back-N ARQ protocol.
- Managing packet transmission with timeouts and acknowledgments.
- Handling packet loss and retransmission.
- Sliding window protocol implementation.

You are required to implement a Go-Back-N protocol-based system for reliable data transmission. You will simulate packet loss and manage retransmissions, timeouts, and acknowledgments within a sliding window framework.

Notes

- You are required to adhere to the class structure, method descriptions, and naming conventions for both methods and instance variables as outlined in the sections below.
- Feel free to implement your own logic of the methods as long as they fulfill the requirements specified in the assignment.
- You can add additional methods or instance variables to the class if needed for your implementation.
- Logging is an essential part of this assignment. Make sure to log all events in the same format as the example below provided.
- When developing the methods, it is efficient to use try-except blocks to handle exceptions or errors that may occur in your code. Also make sure to prevent blocking calls by using timeouts when necessary.

2 Implementation

For this assignment, you must submit the implementation of the Go-Back-N sender and receiver. Below is a detailed explanation of the components you need to include:

TOP

GBN_sender Class

__init__(self, input_file, window_size, packet_len, nth_packet, send_queue, ack_queue, timeout_interval, logger)

Instance variables:

- `input_file`: The file containing the data to be transmitted.
- `window_size`: The size of the sliding window for sending packets.
- `packet_len`: The length of the packets to be sent (in bits).
- `nth_packet`: An integer value representing the interval at which packets should be dropped. (e.g., if nth_packet=9, every 9th packet sent by the sender will be dropped).
- `send_queue`: A queue to send packets to the receiver.
- `ack_queue`: A queue to receive acknowledgments from the receiver.
- `timeout_interval`: The timeout period for retransmitting packets if no acknowledgment is received.
- `logger`: A logging object to log events during transmission.

- `base`: The base of the sliding window, indicating the sequence number of the first unacknowledged packet. Initialized to `0`.
- `packets`: A list of all created packets to be sent. Initialized by calling the `prepare_packets` method.
- `acks_list`: A list that tracks which packets have been acknowledged. Initialized to a list of `False` values with the same length as the `packets` list.
- `packet_timers`: A list that keeps track of the timeout for each packet. Initialized to a list of `0` values with the same length as the `packets` list.
- `dropped_list`: A list to track which packets have already been dropped (if using simulated packet loss). Initialized to an empty list.

This method initializes the sender object with the specified parameters, including the `input_file`, `window_size`, `packet_len`, `nth_packet`, `send_queue`, `ack_queue`, `timeout_interval`, and `logger`. It also initializes the `base` to **0**, which indicates the first unacknowledged packet.

The `packets` list is intialized by calling the `prepare_packets` method to read data from the input file and split it into packets. The packet is a string of bits. The `acks_list` is a list of booleans to track which packets have been acknowledged. The `packet_timers` list is used to keep track of the timeout for each packet. It is worth noting that the `acks_list` and `packet_timers` lists have the same length as the `packets` list. Finally, the `dropped_list` is used to store the sequence numbers of packets that have been dropped.

prepare_packets(self)

This method does the following:

- Reads the data from the `input_file`.
- Converts the data into binary representation by encoding each character into its corresponding ASCII code point, padded to 8 bits. For example, the character `'A'` is represented as `'01000001'`. (You can use the `ord` and `format` functions in Python to achieve this).

TOP

- The `packet_len` specifies the number of bits in each packet. It includes the number of bits for the data and the sequence number. The sequence number is `16` bits long. The remaining bits are used for the data. For example, if `packet_len` is `20`, the first `4` bits are used for data, and the last `16` bits are used for the sequence number.
- Divides the data into chunks and appends the sequence number to each to create packets of length `packet_len`. The last packet may have fewer bits if the data does not fill the packet completely, but the sequence number is always included and padded to `16` bits.
- Returns a list of the created packets.

send_packets(self)

This method sends all packets within the sliding window starting from the `base`. The method first logs an informational message for each packet will be sent, using the `logger` object (format: `f"sending packet {x}"`, where `x` is the sequence number of the packet being sent).

To send a packet, the method enqueues the packet in the `send_queue`. If the packet is the `nth_packet` and not has been dropped, it is not enqueued, and appended to the `dropped_list` instead and the logger logs a message indicating that the packet has been dropped (format: `f"packet {x} dropped"`, where `x` is the sequence number of the packet dropped). The method also starts a timer for each packet to track the timeout.

Note:

- If the packet is dropped, the packet itself is not added to the `dropped_list`. Instead, the sequence number of the packet is added to the list.
- When starting the timer for a packet, the timer is set to the current time (`time.time()`).

send_next_packet(self)

This method is called upon receiving an acknowledgment to send the next packet in the sliding window.

The method first increments the `base` to move the sliding window forward. Then, it gets the last packet in the sliding window. For this last packet, the method does the same process and checkings as in the `send_packets()` method for sending it.

check_timers(self)

This method checks if any packets within the sliding window have exceeded their timeout interval. If a timeout occurs, the `logger` logs a message indicating that the packet has timed out (format: `f"packet {x} timed out"`, where `x` is the sequence number of the packet that timed out), and returns `True`. Otherwise, it returns `False`.

receive_acks(self)

This method continuously listens for acknowledgments from the receiver by dequeuing acknowledgments from the `ack_queue`. Acknowledgments are integers representing the sequence number of the packet that has been acknowledged. For example, if an acknowledgment is `5`, it means that the packet with sequence number `5` has been received by the receiver.

TOP

The method updates the `acks_list` to mark the corresponding packet as acknowledged by setting it to `True`. Then, the logger logs an informational message indicating that the packet has been acknowledged (format: `f"ack {x} received"`, where `x` is the sequence number of the acknowledged packet). Finally, it calls `send_next_packet()` to send the next unsent packet in the sliding window.

If a packet was acknowledged before and the acknowledgment is received again, the logger logs a message indicating that the acknowledgment is ignored (format: `f"ack {x} received, Ignoring"`, where `x` is the sequence number of the acknowledged packet).

run(self)

This is the main method that starts the sender's operation. It begins by calling `send_packets()` to transmit the packets in the initial sliding window. Then, it starts a thread to handle the reception of acknowledgments. While the base has not reached the total number of packets, it continuously checks for timeouts using `check_timers()`. If a timeout occurs, the method retransmits the packets in the sliding window. Once all packets have been sent and acknowledged, the sender enqueues a `None` in the `send_queue` to notify the receiver that the transmission is complete.

GBN_receiver Class

__init__(self, output_file, send_queue, ack_queue, logger)

Instance variables:

- `output_file`: The file where the received data will be written.
- `send_queue`: A queue to receive packets from the sender.
- `ack_queue`: A queue to send acknowledgments back to the sender.
- `logger`: A logging mechanism to log events during reception.
- `packet_list`: A list to store the received packets.
- `expected_seq_num`: The sequence number of the next expected packet.

process_packet(self, packet)

This method processes each packet received from the sender. It extracts the sequence number from the packet and checks if it matches the `expected_seq_num`. If the packet is in order, the data is appended to `packet_list`, and an acknowledgment is sent back to the sender. The logger then logs a message indicating that the packet is received (format: `f"packet {x} received"` where `x` is the sequence number of the packet received), and the method returns `True`. If the packet is out of order, the receiver resends the acknowledgment for the last successfully received packet, and the logger logs a message indicating that the packet is out of order (format: `f"packet {x} received out of order"` where `x` is the sequence number of the packet received), and the method returns `False`.

write_to_file(self)

TOP

This method writes the received data to the output file once all packets have been received from `packet_list`. The data is converted from binary format back into characters and written to the file specified by `output_file`.

run(self)

This is the main method for the receiver. It continuously listens for packets from the `send_queue` until a `None` packet is received, which signals the end of the transmission. Each received packet is processed by calling `process_packet()`, and once all packets have been received, the data is written to the output file using `write_to_file()`.

3 Example

Suppose you have a sender and receiver with the following parameters:

- Window size: `4`
- Packet length: `32`
- Timeout interval: `1`
- Nth packet: `4`

The data to be transmitted is **`"hello world"`**. The string data is converted to binary format using ASCII encoding and padded to `8-bits`.

So the binary representation of the string data is ('-' is used to separate each character):

- **`'hello'`**: `01101000-01100101-01101100-01101100-01101111`
- `' '`: `00100000`
- **`'world'`**: `01110111-01101111-01110010-01101100-01100100`

The total binary data is:

`01101000011001010110110001101100011011110010000001110111011011110111001001101100011001000`

The sequence numbers of `16-bits` are appended to each packet. Since the packet length is `32-bits`, the packet will contain `16-bits` of **data** and `16-bits` of **sequence number**.

We found that the total number of packets is `6`. The packets are as follows:

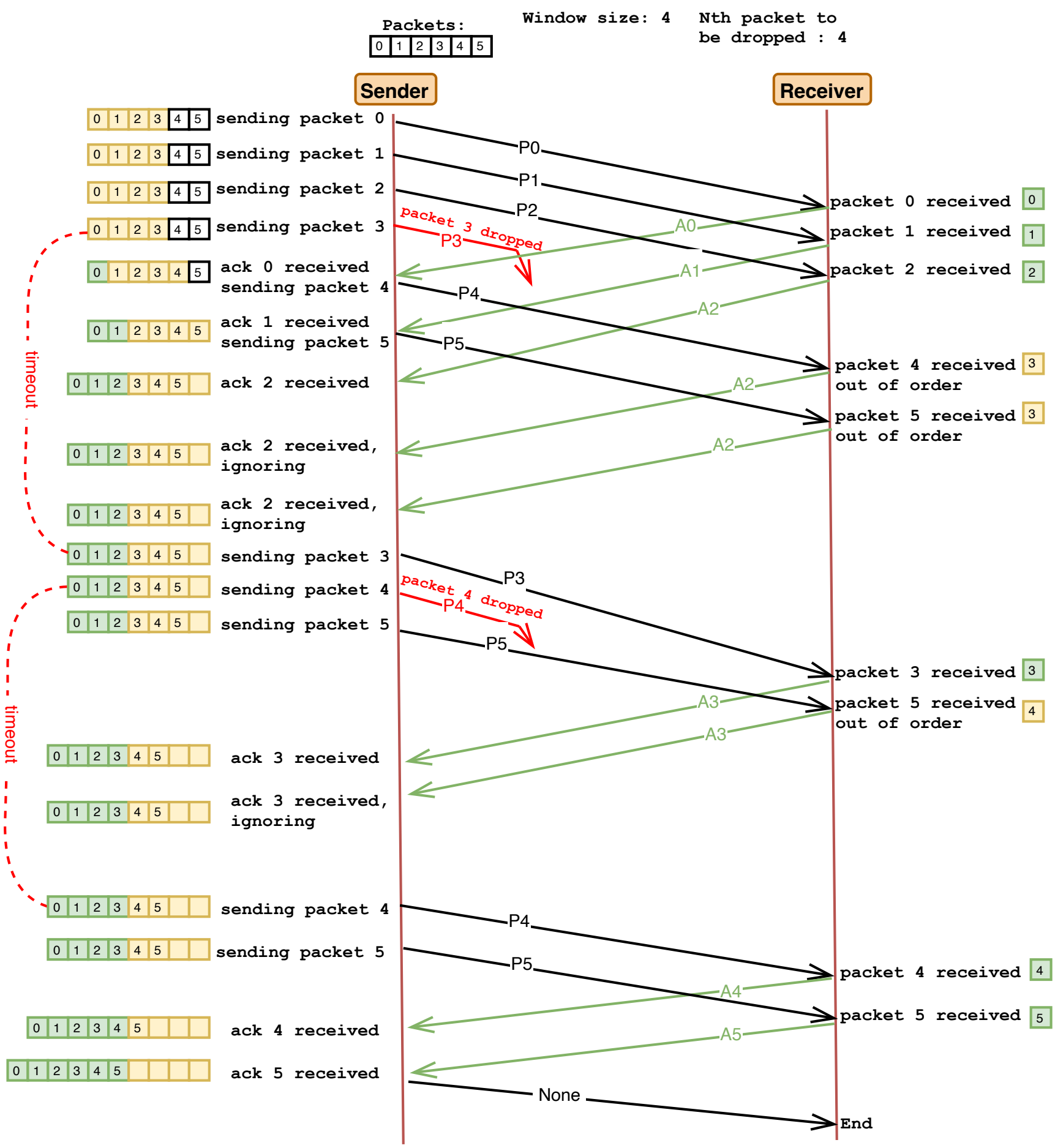
- Packet 0: **`01101000011001010000000000000000`**
- Packet 1: **`01101100011011000000000000000001`**
- Packet 2: **`01101111001000000000000000000010`**
- Packet 3: **`01110111011011110000000000000011`**
- Packet 4: **`01110010011011000000000000000100`**
- Packet 5: **`01100100000000000000101`**

After the sender and the receiver start operating as described in the previous section, the log of events will be as follows:

TOP

```
2024-09-18 12:17:14,712 - 6 packets created, Window size: 4, Packet length: 32, Nth packet to be dropped: 4, Timeout interval: 1
2024-09-18 12:17:14,713 - Sender: sending packet 0
2024-09-18 12:17:14,713 - Sender: sending packet 1
2024-09-18 12:17:14,713 - Sender: sending packet 2
2024-09-18 12:17:14,713 - Sender: sending packet 3
2024-09-18 12:17:14,713 - Sender: packet 3 dropped
2024-09-18 12:17:14,714 - Receiver: packet 0 received
2024-09-18 12:17:14,714 - Receiver: packet 1 received
2024-09-18 12:17:14,734 - Sender: ack 0 received
2024-09-18 12:17:14,734 - Receiver: packet 2 received
```

If we plot these events, it will look like the following diagram:



TOP

4 Attachments

1 The `go_back_n.py` file should contain the following code structure:

```
class GBN_sender:
        def __init__(self, input_file, window_size, packet_len, nth_packet,
send_queue, ack_queue, timeout_interval, logger):
        pass


    def prepare_packets(self):
        pass


    def send_packets(self):
        pass


    def send_next_packet(self):
        pass


    def check_timers(self):
        pass


    def receive_acks(self):
        pass


    def run(self):
        pass


class GBN_receiver:
    def __init__(self, output_file, send_queue, ack_queue, logger):
        pass


    def process_packet(self, packet):
        pass


    def write_to_file(self):
        pass


    def run(self):
        pass
```

2 To test your `go_back_n.py` implementation, you can use file `test_gbn.py` that contains the following code:

TOP

```
from go_back_n import GBN_sender, GBN_receiver
import threading, queue, logging


log_file = 'simulation.log'
in_file = 'input_test.txt'
out_file = 'output_test.txt'
with open(in_file, 'w') as f: f.write("Hello World")


window_size = 4
packet_len = 32
nth_packet = 4
timeout_interval = 1


logger = logging.getLogger()
logger.setLevel(logging.INFO)
file_handler = logging.FileHandler(log_file, 'w')
file_handler.setFormatter(logging.Formatter('%(asctime)s - %(message)s'))
logger.addHandler(file_handler)


send_queue, ack_queue = queue.Queue(), queue.Queue()
sender = GBN_sender(input_file = in_file, window_size = window_size, packet_len
= packet_len, nth_packet = nth_packet, send_queue = send_queue, ack_queue =
ack_queue, timeout_interval = timeout_interval, logger = logger)
receiver = GBN_receiver(output_file = out_file, send_queue = send_queue,
ack_queue = ack_queue, logger = logger)


sender_thread = threading.Thread(target=sender.run)
sender_thread.start()
receiver.run()
sender_thread.join()


with open(in_file, 'r') as f1, open(out_file, 'r') as f2: sent, received =
f1.read(), f2.read()
if sent == received: print("Data transmitted successfully!")
```

- Submission

TOP

REMEMBER!

- You have 4 coupons (for the entire semester) that will be automatically applied when you submit late.
- It is the student's responsibility to ensure the work was submitted and posted in GradeScope.
- Any assignment not submitted correctly will not be graded.
- Submissions through the email will not be accepted under any circumstances.
- Please check this page back whenever an announcement is posted regarding this assignment.
- Marks will be deducted if you submit anything other than the required Python files.
- Submit the assignment on time. Late submissions will penalty unless you have enough coupons to be applied (automatically).
- You may re-submit your code as many times as you like. Gradescope uses your last submission for grading by default. There are no penalties for re-submitting. However, re-submissions that come in after the due date will be considered late.
- Again, assignments will be run through similarity checking software to check for code similarity. Sharing or copying code in any way is considered plagiarism (Academic dishonesty) and may result in a mark of 0 on the assignment and/or be reported to the Dean's Office. Plagiarism is a serious offense. Work is to be done individually.

1. You must submit one file only to the Assignment submission page on Gradescope. The required file is as follows:

   `go_back_n.py`

2. Make sure that you get the confirmation email after submission.

3. The autograder takes around **80 seconds** to run and test your submission. if your submission takes longer than that, it means that you have an infinite loop or blocking call in your code.

4. The submission will be using Gradescope Assignment-03 page .

TOP