Programming Assignment 3 for CSE232

Simulation of Go back n protcol

Palak Bhardwaj (2022344)
Yashovardhan Singhal (2022591)

8 November 2024

Abstract :

This report presents an implementation of the Go-Back-N (GBN) protocol using datagram sockets to simulate a reliable data link layer over the unreliable transport service provided by the User Datagram Protocol (UDP). The main objective of this assignment was to deepen the understanding of data link protocols by simulating Go-Back-N, a sliding window protocol that ensures reliable data transfer between two network entities.

The simulation consists of two primary data link entities, DL_Entity_1 and DL_Entity_2, which represent the sender and receiver, respectively. Packets are generated at random time intervals by both network entities, with the time between two successive packets being uniformly distributed between chosen values T1 and T2. Once generated, these packets are enqueued in an outgoing buffer and transmitted as frames according to the GBN protocol.

The implementation follows the basic principles of the Go-Back-N protocol. DL_Entity_1 transmits frames from the outgoing queue and waits for acknowledgment from DL_Entity_2. If an acknowledgment is not received or a frame is dropped due to simulated errors, the sender retransmits all unacknowledged frames starting from the last acknowledged frame. The receiver validates the received frames based on the sequence number and sends acknowledgment for successfully received frames. The frame structure includes essential components such as sequence numbers and acknowledgment numbers, and the sequence numbering scheme uses modulo-8 with a transmission window size of 7 (N=8) and a receiver window size of 1.

- `socket` : This module provides functions for creating and managing network connections using protocols like TCP and UDP.
- `time` : Used for various time-related functions, such as adding delays or capturing timestamps.
- `random` : Used to generate random values, which in this case, is used to simulate packet generation at random intervals.
- `threading` : Allows the program to run multiple threads concurrently, enabling parallel execution of the network and data link layer simulations.
- `queue` : A thread-safe FIFO queue used for managing frames between the network and data link layers.
- `SERVER_ADDRESS` : The address of the receiver, specified as a tuple `(hostname, port)`. Here, it's set to the local machine (`localhost`) and port `12000`.
- `WINDOW_SIZE` : The size of the sliding window in the Go-Back-N protocol, set to 8. This means the sender can send up to 8 frames without waiting for an acknowledgment.
- `TOTAL_FRAMES` : The total number of frames that need to be transmitted. In this case, 20 frames.
- `T1`, `T2` : These are the minimum and maximum time intervals (in seconds) between each packet generation, used to simulate random intervals.
- `packet_queue` : A queue to hold the packets generated by the network layer, which are to be sent by the data link layer.
- `client_socket` : A UDP socket created using `socket.AF_INET` (IPv4) and `socket.SOCK_DGRAM` (UDP) for sending data to the receiver.
- `base` : Keeps track of the starting frame number in the sender's sliding window. Initially set to 1.
- `next_frame_to_send` : Points to the next frame to send. Initially set to the value of `base`.
- `no_tr` : The count of total transmissions (including retransmissions) made by the sender.
- `stop_sending` : A flag indicating when to stop the sending process. Initially set to `False`.
- `first_send_times` : A dictionary to store the first send time for each frame.
- `first_receive_times_at_sender` : A dictionary to store the timestamps when the sender receives an acknowledgment for each frame.
- `transmission_count` : A dictionary to track how many times each frame has been transmitted.
- `network_layer()` : Simulates the network layer, which generates packets at random intervals and places them in the `packet_queue` for transmission by the data link layer.
- `time.sleep(random.uniform(T1, T2))` : Introduces a random delay between packet generation to simulate network delay.
- `packet_queue.put(f"Frame {frame_number}".encode())` : Adds the generated frame to the queue for sending.
- `frame_number += 1` : Increments the frame number after each packet is generated.
- `data_link_layer()` : Simulates the data link layer that implements the Go-Back-N protocol to send packets.
- `while packet_queue.qsize() < 8` : Ensures the sender waits until there are at least 8 frames in the queue before sending them.
- `while not packet_queue.empty() and next_frame_to_send < base + WINDOW_SIZE` : This loop sends packets while there is space in the window and the queue is not empty.
- `client_socket.sendto(packet, SERVER_ADDRESS)` : Sends the packet to the receiver using the UDP socket.
- `if next_frame_to_send not in first_send_times` : Records the time when a frame is first sent.
- `next_frame_to_send += 1` : Increments the pointer for the next frame to send.
- `client_socket.settimeout(5)` : Sets a timeout of 5 seconds for waiting for an acknowledgment from the receiver.
- `ack_packet, _ = client_socket.recvfrom(1024)` : Waits for an acknowledgment packet from the receiver.

— `ack_num = int(ack_data[3])` : Extracts the acknowledgment number from the decoded packet.
— `if ack_num == base` : If the acknowledgment matches the base, the sender slides the window by incrementing `base`.
— `elif ack_num > base` : If the acknowledgment is for a frame beyond the base, retransmit frames starting from the base.
— `except socket.timeout` : If no acknowledgment is received within the timeout period, retransmit the frames in the window.
— `stop_sending = True` : Sets the flag to stop the sending process once all frames are transmitted or a timeout occurs.

— `socket` : This module is used for creating and managing network connections, here specifically for the UDP protocol, allowing the server to receive and send messages to the client.
— `random` : This module is used to generate random values. It's used to simulate random delay and packet drop behavior in this server.
— `time` : Provides functions to measure time, such as sleeping for random delays and tracking the first time a frame is received.
— `SERVER_ADDRESS` : Specifies the IP address and port number for the server to listen on. Here, the server listens on the local machine (`localhost`) and port `12000`.
— `BUFFER_SIZE` : Defines the size of the buffer used to receive data from the client, here set to `1024` bytes.
— `drop_probability` : The probability of dropping an acknowledgment message, set to `0.3` (30
— `T3`, `T4` : These are the minimum (`T3` = 0.2 seconds) and maximum (`T4` = 0.5 seconds) delays that will be introduced before sending the acknowledgment to simulate network delay.
— `server_socket` : A UDP socket object created using `socket.AF_INET` (IPv4) and `socket.SOCK_DGRAM` (UDP). This socket listens for incoming messages from clients.
— `server_socket.bind(SERVER_ADDRESS)` : Binds the server socket to the specified address and port, so the server can listen for incoming packets on that address.
— `first_receive_times` : A dictionary that stores the first time each frame is received. The frame number is the key, and the receive timestamp is the value.
— `while True` : The server runs an infinite loop to continuously listen for incoming packets from clients.
— `packet, client_address = server_socket.recvfrom(BUFFER_SIZE)` : Waits for and receives a packet from the client. It also captures the client's address so the server can send a response back.
— `frame_number = int(packet.decode().split()[1])` : Decodes the received packet to extract the frame number. It assumes the packet is a string where the frame number is the second word.
— `print(f"Received: Frame {frame_number}")` : Prints out the received frame number to the console for logging.
— `if frame_number not in first_receive_times` : Checks if the frame number has been received before. If not, it records the current timestamp as the first receive time for that frame.
— `first_receive_times[frame_number] = time.time()` : Stores the timestamp of the first time the frame is received in the `first_receive_times` dictionary.
— `delay = random.uniform(T3, T4)` : Generates a random delay between `T3` and `T4` seconds to simulate queuing and propagation delay.
— `print(f"Introducing delay of {delay:.3f} seconds")` : Prints the delay introduced before sending the acknowledgment.
— `time.sleep(delay)` : The server pauses for the randomly generated delay to simulate network conditions before responding to the client.
— `if random.random() > drop_probability` : Simulates the possibility of dropping the acknowledgment based on the configured drop probability.
— `recv_time = first_receive_times[frame_number]` : Retrieves the timestamp of when the frame was first received.
— `ack_message = f"Acknowledgment for Frame {frame_number} {recv_time}"` : Constructs the acknowledgment message containing the frame number and the timestamp when it was first received.
— `server_socket.sendto(ack_message.encode(), client_address)` : Sends the acknowledgment message back to the client with the corresponding timestamp.
— `print(f"Sent: {ack_message}")` : Prints the acknowledgment message sent to the client for logging.
— `else` : If the drop condition is met (i.e., a random number is less than or equal to the drop probability), the acknowledgment is not sent.

— `print(f"Dropped: No acknowledgment for Frame {frame_number}")` : Prints a message indicating that no acknowledgment was sent due to the simulated drop.

Simulating Network Entities as Processes

To simulate a network entity as a process that produces packets at random time instants, we can use a combination of random time delays and thread-based parallelism to model the behavior. The network entity, in this case, can be simulated as a producer process that generates packets at random intervals. This can be achieved using the following approach :

— **Random Time Generation :** We can use the `random` module to generate random delays between packet generation events. These delays can be modeled using the `random.uniform(min, max)` function, where `min` and `max` represent the minimum and maximum delays (in seconds) before the next packet is produced.

— **Packet Creation :** The packet can be created in the form of a message or data structure. For instance, it could be a tuple containing data such as the packet's frame number, timestamp, or other metadata.

— **Packet Sending :** Once a packet is created, it can be sent over the network using a socket (e.g., `socket.sendto()`) or added to a queue to be consumed later by another network entity.

— **Threads for Parallelism :** A separate thread can be used for the packet generation process to ensure that the packet producer operates independently from other processes in the system. This allows the network entity to generate packets at random time intervals without blocking the rest of the system.

In this setup, the network entity will continuously produce packets at random intervals and send them over a socket or add them to a queue for further processing.

Similarly, a network entity that consumes incoming packets can be simulated by using a queue data structure where packets are stored temporarily. The consumer process can consume these packets at a different rate, potentially introducing additional processing delays or actions.

— **Queue for Incoming Packets :** A `queue.Queue` can be used to store incoming packets. The queue will be thread-safe, ensuring that packets can be safely added and removed from it by different processes (i.e., producer and consumer threads).

— **Packet Consumption :** A consumer thread can repeatedly check the queue for incoming packets and process them as they arrive. Once a packet is consumed, the consumer can either process the packet, acknowledge it, or forward it to another entity.

— **Random Processing Delays :** The consumer thread may also introduce random delays to simulate varying processing times for packets, akin to network latency or server-side delays.

Are Threads an Option ?

Yes, threads are a viable option for simulating both packet production and consumption. By using threads, we can model the network entity as two concurrent processes : one that generates packets at random intervals (producer thread) and one that consumes them from the queue (consumer thread). Threads allow the system to run these operations concurrently without blocking the main process, enabling realistic simulations of network behavior.

In Python, the `threading` module provides a way to create and manage threads. Each thread can operate independently, allowing for asynchronous packet generation and consumption. This parallelism is especially useful when simulating complex network systems where multiple tasks (e.g., producing, sending, receiving, and processing packets) need to happen concurrently.

However, when using threads, it is essential to ensure proper synchronization to prevent race conditions or conflicts when accessing shared resources (such as the queue). Tools like locks and semaphores may be used to manage access to critical sections of code in a thread-safe manner.