This document outlines the design for a two-module system (Sender and Receiver) to simulate the Stop-and-Wait ARQ protocol as shown in the lab diagram (Figure 5).

1. Common Module: stop_and_wait.h

- **Sub-Modules:** Packet Definition, Constants.
- **FSM:** Not applicable (this is a header file).
- **Algorithm:**
  1. Define constants: PORT, HOST, PAYLOAD_SIZE, TIMEOUT_SEC (e.g., 2 seconds).
  2. Define packet types: TYPE_DATA (1) and TYPE_ACK (2).
  3. Define a single, unified Packet struct. This struct will be used for *both* data and ACKs.
- **Code (Functions / Subroutines):**
  - struct Packet:
    - int type: TYPE_DATA or TYPE_ACK.
    - int seq_num: For DATA, this is 0 or 1. For ACKs, this is the ack_num (0 or 1), which is the sequence number of the *next* packet the receiver expects.
    - char data[PAYLOAD_SIZE]: The data payload.
- **Documentation:** A common header to share the packet structure and constants between the sender and receiver.

2. Module: Sender

- **Sub-Modules:** Socket Setup, Timer Setup, Main Send Loop, ACK Wait Loop.
- **FSM (Finite State Machine):**
  - The sender has one main variable: next_seq_to_send (starts at 0).
  - **State 1: WAIT_TO_SEND** (e.g., for Packet 0)
    - **Event:** Req (Program is ready to send).
    - **Action:** Create Packet(TYPE_DATA, seq_num=0). sendto(packet).
    - **Transition:** Move to WAIT_FOR_ACK_1.
  - **State 2: WAIT_FOR_ACK** (e.g., WAIT_FOR_ACK_1)
    - **Event 1:** aArr (ACK arrives).
      - Check if ack_packet.seq_num == 1.
      - **If YES (Correct ACK):** Stop timer, set next_seq_to_send = 1.
      - **Transition:** Move to WAIT_TO_SEND (for Packet 1).
      - **If NO (Wrong/Duplicate ACK):** Ignore ACK. Stay in WAIT_FOR_ACK_1.
    - **Event 2:** T-Out (Timer expires).
      - **Action:** Retransmit Packet(TYPE_DATA, seq_num=0). Restart timer.
      - **Transition:** Stay in WAIT_FOR_ACK_1.
- **Algorithm:**
  1. Create a **UDP** socket.
  2. Set a **receive timeout** on the socket using setsockopt(SO_RCVTIMEO). This will be our timer. recvfrom() will fail with a "timeout" error if no ACK arrives.
  3. Initialize next_seq_to_send = 0.
  4. Create two sample data payloads (e.g., "Packet 0 Data", "Packet 1 Data").

5. Loop 3-4 times to simulate the scenarios.
6. Inside the loop:
    - Create the data packet (pkt.type = TYPE_DATA, pkt.seq_num = next_seq_to_send).
    - sendto(packet).
    - Enter a retransmission_loop (a while(true) loop).
    - Call recvfrom() to wait for an ACK.
    - **If recvfrom() succeeds:**
        - An ACK packet arrived. Check its seq_num.
        - If ack.seq_num == (next_seq_to_send + 1) % 2: This is the correct ACK. printf("ACK OK."). break the retransmission_loop.
        - If not: printf("Wrong ACK, ignoring."). Stay in the loop.
    - **If recvfrom() fails (timeout):**
        - printf("Timeout! Resending Packet %d\n", next_seq_to_send).
        - sendto(packet) again. Stay in the retransmission_loop.
7. After the retransmission loop breaks, flip the sequence number: next_seq_to_send = (next_seq_to_send + 1) % 2.

- **Code (Functions / Subroutines):**
    - main(): Contains all logic.
    - die(string): Helper function for errors.
    - **Syscalls:** socket(AF_INET, SOCK_DGRAM, 0), setsockopt(), sendto(), recvfrom(), close().
- **Documentation:** Simulates the sender, which sends data and waits for an ACK. It retransmits on timeout.

3. Module: Receiver

- **Sub-Modules:** Socket Setup, Main Receive Loop, Loss Simulation.
- **FSM (Finite State Machine):**
    - The receiver has one main variable: expected_seq_num (starts at 0).
    - **State: WAIT_FOR_PACKET** (e.g., WAIT_FOR_PACKET_0)
        - **Event:** pArr (Packet arrives).
        - **Action:** Check packet.seq_num.
        - **If seq_num == expected_seq_num (Correct Packet):**
            - printf("Packet %d OK.").
            - Create ACK(type=TYPE_ACK, seq_num=(expected_seq_num + 1) % 2).
            - sendto(ACK).
            - Update state: expected_seq_num = (expected_seq_num + 1) % 2.
        - **If seq_num != expected_seq_num (Duplicate Packet):**
            - printf("Duplicate Packet %d, discarding.").
            - Create ACK(type=TYPE_ACK, seq_num=expected_seq_num). (Resend the *last* good ACK).
            - sendto(ACK).
            - Stay in the same state.

- **Algorithm:**
  1. Create a **UDP** socket.
  2. bind() the socket to the PORT.
  3. Initialize expected_seq_num = 0.
  4. Initialize packet_receive_counter = 0.
  5. Enter a while(true) loop to receive packets.
  6. Call recvfrom().
  7. packet_receive_counter++.
  8. **Simulate Packet 1 Loss (Scenario 2):**
     - if (packet_receive_counter == 2 && packet.seq_num == 1):
     - printf("SIMULATING PACKET LOSS (Packet 1)\n").
     - continue; (Drop the packet, send no ACK).
  9. **Check for duplicate/correct packet:**
     - if (packet.seq_num == expected_seq_num):
       - Handle correct packet (as per FSM).
       - ack_to_send = (expected_seq_num + 1) % 2.
       - expected_seq_num = (expected_seq_num + 1) % 2.
       - **Simulate ACK 1 Loss (Scenario 3):**
         - if (packet_receive_counter == 4) (This is the Packet 0 that arrives after the successful Packet 1 retransmission).
         - printf("SIMULATING ACK LOSS (ACK 1)\n").
         - continue; (Drop the ACK, don't send it).
       - sendto(ack_packet).
     - else (Duplicate packet):
       - Handle duplicate packet (as per FSM).
       - ack_to_send = expected_seq_num.
       - sendto(ack_packet).
- **Code (Functions / Subroutines):**
  - main(): Contains all logic.
  - die(string): Helper function for errors.
  - **Syscalls:** socket(), bind(), recvfrom(), sendto(), close().
- **Documentation:** Simulates the receiver, which ACKs correct packets, discards duplicates, and resends the last ACK for duplicates. It also simulates packet and ACK loss.