Custom Low-Level Protocol for Battleship (Updated)

This document outlines the specification for a custom packet-based protocol designed for the Battleship game. It includes a header with sequence numbers, packet types, payload length, game-specific fields (implicitly via packet types and payload), and a checksum for data integrity, drawing principles from your "Transport Layer" lecture.

1. Packet Structure

We'll define a fixed-size header followed by a variable-size payload. All multi-byte numerical fields will be encoded in **big-endian** format (network byte order).

Field	Size (bytes)	Description
Header		
Sequence Number	4	A unique, incrementing number for each packet sent. Used to detect lost or out-of-order packets (similar to TCP's sequence numbers, Lecture p.20, 26).
Packet Type	1	Defines the purpose/content of the packet (see Packet Types below).
Payload Length	2	The length of the Payload section in bytes. A value of 0 means no payload.
Checksum	2	A 16-bit checksum calculated over the custom protocol's Header (excluding the checksum field itself, treated as zero during calculation) and the Payload.
Payload	Variable	The actual data being transmitted, its structure depends on the Packet Type. Max length: 65535 bytes.

Total Header Size: 4 (Seq) + 1 (Type) + 2 (Len) + 2 (Checksum) = **9 bytes**

Packet Types (Example Values):

These are suggestions; you can expand or modify them.

Type ID (Decimal)	Mnemonic	Direction	Payload Description
0	`CMD_PLACE`	Client -> Srv	Ship placement choice: "M" (manual) or "R" (random). (1 byte: 'M' or 'R')
1	`CMD_COORD`	Client -> Srv	Ship placement coordinate (e.g., "A1"). (ASCII string)
2	`CMD_ORIENT`	Client -> Srv	Ship placement orientation (e.g., "H"). (1 byte: 'H' or 'V')

3	`CMD_FIRE`	Client -> Srv	Fire coordinate (e.g., "B5"). (ASCII string)
4	`CMD_QUIT`	Client -> Srv	Player quits. (No payload)
5	`CMD_PLAY_AGAIN`	Client -> Srv	Player response to "play again?" (1 byte: 'Y' or 'N')
10	`MSG_WELCOME`	Server -> Cli	Welcome message. (UTF-8 string)
11	`MSG_INFO`	Server -> Cli	General information/prompt. (UTF-8 string)
12	`MSG_BOARD_OWN`	Server -> Cli	Player's own board state. (Serialized board string)
13	`MSG_BOARD_OPP`	Server -> Cli	Opponent's board state (public view). (Serialized board string)
14	`MSG_RESULT`	Server -> Cli	Result of a fire command (e.g., "HIT", "MISS", "SUNK Carrier"). (UTF-8 string)
15	`MSG_GAME_OVER`	Server -> Cli	Game over message. (UTF-8 string)
16	`MSG_WAITING`	Server -> Cli	Waiting for other player/in queue. (UTF-8 string)
17	`MSG_SPECTATOR`	Server -> Cli	Spectator specific messages / board states. (UTF-8 string)
20	`ACK`	Both	Acknowledgement for a received packet. Payload: Sequence number being ACKed (4 bytes). (Concept from TCP, Lecture p.19-20)
21	`NACK`	Both	Negative Acknowledgement for a corrupted/missing packet. Payload: Seq num (4 bytes).
22	`HEARTBEAT`	Both	Keep-alive packet. (No payload)
23	`ERR_MSG`	Server -> Cli	Error message from server (e.g. invalid input). (UTF-8 string)

Note on String Payloads: The `Payload Length` field is crucial. For strings, ensure this length matches the byte length of the encoded string (e.g., UTF-8).

2. Checksum Mechanism

We'll use the **Internet Checksum algorithm**, as described in your lecture notes (e.g., for IP/UDP on p.14). This involves a 16-bit one's complement sum of 16-bit words.

Scope of Checksum: The checksum is calculated over your custom packet's header (with the checksum field itself temporarily zeroed) and its payload.
* *Clarification based on Lecture (p.11-13 regarding UDP Pseudo-Header):* The UDP/TCP checksums include a "pseudo-header" with IP addresses because they operate directly above

IP. For your custom application-layer protocol (which will run over TCP or UDP via Python sockets), the underlying transport (TCP/UDP) and network (IP) layers will handle their own checksums covering IP addresses, ports, etc. Your custom checksum only needs to protect your application-defined header and payload.

Checksum Generation (Sender):
1. Concatenate your custom packet's `Sequence Number` (4 bytes), `Packet Type` (1 byte), and `Payload Length` (2 bytes). This forms 7 bytes.
2. If a payload exists, append the `Payload` bytes to these 7 bytes.
3. The data to be checksummed is this sequence of bytes: `SeqNum (bytes) + PktType (byte) + PayloadLen (bytes) + Payload (bytes)`.
4. Initialize a 32-bit sum to 0.
5. Iterate through the data in 16-bit words (2 bytes at a time). Add each 16-bit word to the sum.
6. If the total length of the data is odd, pad it with a zero byte at the end *for the checksum calculation only* (this padding byte is not transmitted as part of the packet itself).
7. After summing all 16-bit words, "fold" any carry bits from the most significant 16 bits of the sum into the least significant 16 bits. This means while the sum is greater than `0xFFFF` (16 bits), take the upper 16 bits, add them to the lower 16 bits, and repeat.
   `while (sum >> 16): sum = (sum & 0xFFFF) + (sum >> 16)`
8. Take the one's complement of this 16-bit sum (flip all the bits). This is your checksum.
   `checksum = ~sum & 0xFFFF`
9. Place this 16-bit checksum into the `Checksum` field of your packet header (in big-endian order).

Checksum Verification (Receiver):
1. Extract the received `Checksum` value from the header.
2. Prepare the data for checksum calculation exactly as the sender did: `Received SeqNum (bytes) + Received PktType (byte) + Received PayloadLen (bytes) + Received Payload (bytes)`.
3. Calculate a new checksum on this received data using the *exact same method* as generation (steps 4-8 above, including padding if necessary for an odd length of these combined fields).
4. **Crucially, for verification using the Internet Checksum algorithm, you sum the received header fields (SeqNum, PktType, PayloadLen), the received payload, AND the received checksum value itself (treated as a 16-bit word).**
5. If the data (including the original checksum) is error-free, the 16-bit one's complement sum of all these 16-bit words (after folding carries) will be `0xFFFF` (all ones). If it's not `0xFFFF`, the packet is considered corrupted.
   * Alternatively, you can calculate the checksum on the header (with checksum field zeroed) and payload, and compare it to the received checksum value. If they match, it's valid. The "summing everything including checksum should result in 0xFFFF" is a common verification technique for the Internet Checksum.

3. Error Handling Policy

Your error handling policy is a key part of the protocol design.

Corrupted Packets (Checksum Mismatch):
* **Policy (as per Lecture p.20 for TCP):** Discard the packet. The receiver detects errors and might implicitly or explicitly signal the sender (e.g., TCP doesn't ACK corrupted data).
* **For your custom protocol:**
    * **Minimum:** Discard the packet and log the error.
    * **Enhancement (if implementing reliability):** If you build an ACK/NACK system (similar to TCP's ACKs, Lecture p.19-20, 35-36), you could send a `NACK` (Negative Acknowledgement) for the sequence number of the corrupted packet to request retransmission. This requires the sender to buffer sent packets.

Out-of-Sequence Packets:
* **Policy (drawing from TCP principles, Lecture p.20):** TCP uses sequence numbers to detect missing data and reorder packets.
* **For your custom protocol:**
    * **Simple:** Maintain an `expected_sequence_number`. If a received packet's sequence number doesn't match, discard it and log. This is simpler but can be inefficient if retransmissions are frequent.
    * **More Advanced (if implementing reliability):**
        * If packet `N` is expected, but packet `N+1` arrives: You could buffer `N+1` (for a short time/small window) and potentially send a `NACK` for `N` (or wait for sender timeout if you have a reliable sender).
        * If a duplicate packet (already processed sequence number) arrives: Discard it. If using ACKs, you might re-send the ACK for that sequence number as the previous ACK could have been lost.

For your report, you MUST state your chosen policy clearly. A simple "discard on checksum failure, discard if out of order (and no ACK/NACK)" is a valid starting policy for this assignment if full TCP-like reliability is not required.

4. Integrating into Your Python Code

This requires using `socket.sendall()` and `socket.recv()` with byte strings, and the `struct` module for packing/unpacking header fields.

Key Python Modules:
* `struct`: For packing (Python data types to bytes) and unpacking (bytes to Python data types). Use `!` for network (big-endian) order.
* `socket`: For network communication.

General Steps:

A. Define Constants and Helper Functions (Updated Checksum):

```python
import struct
import socket # For socket.htons, socket.ntohs if needed, though struct handles endianness

# --- Packet Field Sizes ---
SEQ_NUM_SIZE = 4
PACKET_TYPE_SIZE = 1
PAYLOAD_LEN_SIZE = 2
CHECKSUM_SIZE = 2
HEADER_SIZE = SEQ_NUM_SIZE + PACKET_TYPE_SIZE + PAYLOAD_LEN_SIZE +
CHECKSUM_SIZE

# --- Packet Type IDs (example) ---
PACKET_TYPE_CMD_FIRE = 3
PACKET_TYPE_MSG_RESULT = 14
# ... add all your defined types

# Sequence number management (per connection)
# client_sequence_number = 0 # Client-side
# server_expected_sequence_numbers = {} # Server-side: map client_id to
expected_seq_num
# server_send_sequence_numbers = {} # Server-side: map client_id to next_seq_num_to_send

def internet_checksum(data_bytes):
    """
    Calculates the Internet Checksum (16-bit one's complement sum).
    'data_bytes' should be the header (with checksum field zeroed) + payload.
    """
    s = 0
    # Iterate over data in 16-bit words
    for i in range(0, len(data_bytes), 2):
        word = data_bytes[i] << 8 # High byte
        if (i + 1) < len(data_bytes):
            word += data_bytes[i+1] # Low byte
        s += word
        s = (s & 0xffff) + (s >> 16) # Fold carry

    # One more fold to catch any carry from the previous fold
    s = (s & 0xffff) + (s >> 16)
```

```
    # Return one's complement
    return ~s & 0xffff

def create_packet(seq_num, packet_type, payload=b''):
    """Creates a packet with header and payload, including checksum."""
    payload_len = len(payload)

    # 1. Pack header fields for checksum calculation (checksum field is conceptually zero here)
    #    and concatenate with payload.
    #    ! = network (big-endian)
    #    I = unsigned int (4 bytes) for seq_num
    #    B = unsigned char (1 byte) for packet_type
    #    H = unsigned short (2 bytes) for payload_len

    # Data for checksum: SeqNum, PktType, PayloadLen, Payload
    # We'll construct this byte string manually for clarity with the checksum algorithm

    seq_bytes = struct.pack("!I", seq_num)
    type_byte = struct.pack("!B", packet_type)
    len_bytes = struct.pack("!H", payload_len)

    data_to_checksum = seq_bytes + type_byte + len_bytes + payload

    # 2. Calculate checksum
    checksum = internet_checksum(data_to_checksum)

    # 3. Construct full header with calculated checksum
    full_header = struct.pack("!IBHH", seq_num, packet_type, payload_len, checksum)
    return full_header + payload

def unpack_and_verify_packet(packet_bytes):
    """
    Unpacks a received packet and verifies its checksum.
    Returns (seq_num, packet_type, payload, is_valid)
    """
    if len(packet_bytes) < HEADER_SIZE:
        # Packet is too short to contain a full header.
        print("[UNPACK ERROR] Packet too short for header.")
        return None, None, None, False

    header = packet_bytes[:HEADER_SIZE]
    payload = packet_bytes[HEADER_SIZE:]
```

```
    # Unpack all header fields including the received checksum
    seq_num, packet_type, payload_len, received_checksum = struct.unpack("!IBHH", header)

    # Verify payload length
    if payload_len != len(payload):
        print(f"[UNPACK ERROR] Payload length mismatch: header says {payload_len}, actual is
{len(payload)}.")
        return seq_num, packet_type, payload, False # Return unpacked values for logging if
needed

    # Data for checksum verification: SeqNum, PktType, PayloadLen, Payload
    # This is the data over which the original checksum was computed.
    seq_bytes_recv = struct.pack("!I", seq_num)
    type_byte_recv = struct.pack("!B", packet_type)
    len_bytes_recv = struct.pack("!H", payload_len)

    data_that_was_checksummed = seq_bytes_recv + type_byte_recv + len_bytes_recv +
payload

    # Recalculate checksum on the received data (excluding the received_checksum itself)
    calculated_checksum = internet_checksum(data_that_was_checksummed)

    is_valid = (calculated_checksum == received_checksum)

    if not is_valid:
        print(f"[CHECKSUM FAIL] Received: {received_checksum:04x}, Calculated:
{calculated_checksum:04x}")

    return seq_num, packet_type, payload, is_valid
```

B. Modifying client.py (Illustrative Snippets):

- Remove rfile and wfile. Use s.sendall() and s.recv().
- Maintain client_send_sequence_number (increment before sending each new packet).
- Maintain client_expected_sequence_number_from_server.

```
# client.py (Illustrative Snippets)
# global client_send_sequence_number = 0 # Initialize
# global client_expected_sequence_number_from_server = 0 # Initialize

def send_packet_to_server(sock, packet_type, payload_data=b''):
    global client_send_sequence_number # Use a global or pass/return if part of a class
    packet = create_packet(client_send_sequence_number, packet_type, payload_data)
    try:
```

```
        sock.sendall(packet)
        print(f"[DEBUG CLIENT SENT] Seq: {client_send_sequence_number}, Type:
{packet_type}, Len: {len(payload_data)}")
        client_send_sequence_number += 1
        # If implementing ACKs/retransmissions, add to an "unacknowledged packets" buffer
here.
    except socket.error as e:
        print(f"[CLIENT SEND ERROR] {e}")
        # Handle error, maybe close connection or set a flag

def receive_packet_from_server(sock):
    global client_expected_sequence_number_from_server # Use global or pass/return
    try:
        # 1. Read the fixed-size header first to determine payload length
        header_bytes = recv_all(sock, HEADER_SIZE) # Use recv_all helper
        if not header_bytes:
            print("[CLIENT] Server closed connection (failed to read header).")
            return None, None, None, False

        # Temporarily unpack just payload_len to know how much more to read
        # We can't fully validate yet as we don't have the full packet for checksum
        _dummy_seq, _dummy_type, payload_len_from_header, _dummy_checksum =
struct.unpack("!IBHH", header_bytes)

        # 2. Read the payload based on payload_length
        payload_bytes = b''
        if payload_len_from_header > 0:
            payload_bytes = recv_all(sock, payload_len_from_header) # Use recv_all helper
            if not payload_bytes or len(payload_bytes) != payload_len_from_header:
                print(f"[CLIENT ERROR] Incomplete or failed payload read. Expected
{payload_len_from_header}, got {len(payload_bytes) if payload_bytes else 0}")
                return None, None, None, False # Critical error in receiving payload

        full_packet_bytes = header_bytes + payload_bytes

        # 3. Unpack and validate the entire packet
        seq_num, packet_type, payload, is_checksum_valid =
unpack_and_verify_packet(full_packet_bytes)

        # Log received packet details
        print(f"[DEBUG CLIENT RECV] Raw: {full_packet_bytes.hex()}")
        print(f"[DEBUG CLIENT RECV] Seq: {seq_num}, Type: {packet_type}, PayloadLen:
{len(payload) if payload else 0}, ChecksumValid: {is_checksum_valid}")
```

```
        if not is_checksum_valid:
            print(f"[CLIENT WARNING] Corrupted packet received from server. Seq: {seq_num if
seq_num is not None else 'N/A'}. Discarding.")
            # Optionally send NACK here if implementing that feature
            return seq_num, packet_type, payload, False # Checksum failed

        # Sequence Number Check (basic)
        if seq_num != client_expected_sequence_number_from_server:
            print(f"[CLIENT WARNING] Out-of-order packet from server. Expected:
{client_expected_sequence_number_from_server}, Got: {seq_num}. Discarding.")
            # More advanced: buffer or send NACK for
client_expected_sequence_number_from_server
            return seq_num, packet_type, payload, False # Out of order, but checksum was ok

        client_expected_sequence_number_from_server += 1
        # Optionally send ACK here if implementing reliability
        return seq_num, packet_type, payload, True # Packet is valid and in order

    except socket.timeout:
        print("[CLIENT RECV TIMEOUT]")
        return None, None, None, False
    except ConnectionResetError:
        print("[CLIENT ERROR] Connection to server was reset.")
        return None, None, None, False
    except struct.error as e: # From struct.unpack if header_bytes was malformed before full
unpack
        print(f"[CLIENT ERROR] Packet unpacking error: {e}")
        return None, None, None, False
    except Exception as e:
        print(f"[CLIENT ERROR] Unexpected error receiving from server: {e}")
        return None, None, None, False

# Helper function to ensure all bytes are received
def recv_all(sock, n_bytes):
    data = bytearray()
    while len(data) < n_bytes:
        try:
            packet_chunk = sock.recv(n_bytes - len(data)) # Renamed to avoid conflict
            if not packet_chunk: # Connection closed
                return None
            data.extend(packet_chunk)
        except socket.timeout: # Handle timeout specifically if socket is non-blocking or has
```

timeout set
        print("[RECV_ALL TIMEOUT]")
        return None # Or raise custom exception
      except socket.error as e: # Other socket errors
        print(f"[RECV_ALL SOCKET ERROR] {e}")
        return None
  return bytes(data)

In your main client loop:
seq, p_type, payload, is_ok = receive_packet_from_server(s)
if is_ok:
Process packet based on p_type
if p_type == PACKET_TYPE_MSG_INFO:
print(payload.decode('utf-8'))
elif seq is None and p_type is None and not is_ok:
This indicates a connection or critical receive error
running = False # Example: stop client
else:
Packet was corrupted or out of order, already logged
pass

C. Modifying server.py (Illustrative Snippets):

- The server needs to manage sequence numbers *per client connection*.
- A dictionary mapping client_socket or a unique client ID to {'next_send_seq': 0, 'expected_recv_seq': 0} is necessary.

server.py (Illustrative Snippets)
player_sessions = {} # Key: client_socket, Value: {'send_seq': 0, 'expect_seq': 0, 'addr': addr, ...}

def init_player_session(client_sock, addr):
  global player_sessions
  player_sessions[client_sock] = {'send_seq': 0, 'expect_seq': 0, 'addr': addr}
  print(f"[SERVER] Initialized session for {addr}")

def send_packet_to_client(client_sock, packet_type, payload_data=b''):
  global player_sessions
  if client_sock not in player_sessions:
    print(f"[SERVER ERROR] No session found for {client_sock.getpeername()} to send packet.")
    return

  session = player_sessions[client_sock]
  seq_num_to_send = session['send_seq']

```
    packet = create_packet(seq_num_to_send, packet_type, payload_data)

    try:
        client_sock.sendall(packet)
        print(f"[DEBUG SERVER SENT to {session['addr']}] Seq: {seq_num_to_send}, Type:
{packet_type}, Len: {len(payload_data)}")
        session['send_seq'] += 1
        # Add to unacked_packets for this client if implementing ACKs
    except socket.error as e:
        print(f"[SERVER SEND ERROR to {session['addr']}] {e}")
        cleanup_client_session(client_sock) # Example cleanup

def receive_packet_from_client(client_sock):
    global player_sessions
    if client_sock not in player_sessions:
        # This might happen if a client connects but session isn't fully set up,
        # or if trying to receive before init. For robustness, could init here,
        # but ideally init_player_session is called upon accepting connection.
        print(f"[SERVER ERROR] No session for {client_sock.getpeername()} to receive packet.")
        # A basic init might be risky if proper handshake/setup is needed first.
        # Consider how to handle this based on your server's connection logic.
        # For now, let's assume it's an error state.
        return None, None, None, False


    session = player_sessions[client_sock]
    expected_seq = session['expect_seq']

    try:
        header_bytes = recv_all(client_sock, HEADER_SIZE) # Use recv_all helper
        if not header_bytes:
            print(f"[SERVER INFO] Client {session['addr']} closed connection (failed to read
header).")
            cleanup_client_session(client_sock)
            return None, None, None, False

        _dummy_seq, _dummy_type, payload_len_from_header, _dummy_checksum =
struct.unpack("!IBHH", header_bytes)

        payload_bytes = b''
        if payload_len_from_header > 0:
            payload_bytes = recv_all(client_sock, payload_len_from_header)
            if not payload_bytes or len(payload_bytes) != payload_len_from_header:
```

```
            print(f"[SERVER ERROR from {session['addr']}] Incomplete or failed payload read.
Expected {payload_len_from_header}, got {len(payload_bytes) if payload_bytes else 0}")
            # Potentially close connection or mark client as problematic
            return None, None, None, False

        full_packet_bytes = header_bytes + payload_bytes
        seq_num, packet_type, payload, is_checksum_valid =
unpack_and_verify_packet(full_packet_bytes)

        print(f"[DEBUG SERVER RECV from {session['addr']}] Raw: {full_packet_bytes.hex()}")
        print(f"[DEBUG SERVER RECV from {session['addr']}] Seq: {seq_num}, Type:
{packet_type}, PayloadLen: {len(payload) if payload else 0}, ChecksumValid:
{is_checksum_valid}")

        if not is_checksum_valid:
          print(f"[SERVER WARNING] Corrupted packet from {session['addr']}. Seq: {seq_num if
seq_num is not None else 'N/A'}. Discarding.")
          return seq_num, packet_type, payload, False # Checksum failed

        if seq_num != expected_seq:
          print(f"[SERVER WARNING] Out-of-order packet from {session['addr']}. Expected:
{expected_seq}, Got: {seq_num}. Discarding.")
          return seq_num, packet_type, payload, False # Out of order

        session['expect_seq'] += 1
        # Optionally send ACK
        return seq_num, packet_type, payload, True

    except socket.timeout:
        print(f"[SERVER RECV TIMEOUT from {session['addr']}]")
        return None, None, None, False
    except ConnectionResetError:
        print(f"[SERVER INFO] Client {session['addr']} reset connection.")
        cleanup_client_session(client_sock)
        return None, None, None, False
    except struct.error as e:
        print(f"[SERVER ERROR from {session['addr']}] Packet unpacking error: {e}")
        return None, None, None, False
    except Exception as e:
        print(f"[SERVER ERROR from {session['addr']}] Unexpected error receiving: {e}")
        cleanup_client_session(client_sock)
        return None, None, None, False
```

```
def cleanup_client_session(client_sock):
    global player_sessions
    session_info = player_sessions.get(client_sock) # Get session info before deleting
    addr = session_info.get('addr', 'unknown client') if session_info else 'unknown client'

    print(f"[SERVER INFO] Cleaning up session for {addr}.")
    if client_sock in player_sessions:
        del player_sessions[client_sock]
    try:
        client_sock.close()
    except socket.error: # Or be more specific e.g. OSError if socket already closed
        pass
    # Additional cleanup: Notify opponent if in a game, remove from game queues, etc.

# When accepting a new connection:
# conn, addr = server_socket.accept()
# init_player_session(conn, addr) # Crucial step
# Then start a thread or handler for this 'conn'
```

5. Statistical Demonstration (Optional but Recommended)

To demonstrate your checksum:

1. **Introduce Errors:** In your client or server, *after* receiving bytes via recv_all() but *before* calling unpack_and_verify_packet(), add a function to occasionally corrupt the received byte string.
   ```
   import random
   def corrupt_packet_bytes_for_test(packet_bytes, corruption_rate=0.1,
   num_bits_to_flip=1):
       """Artificially corrupts a byte string with a certain probability."""
       if random.random() < corruption_rate:
           byte_list = bytearray(packet_bytes) # Use bytearray for mutability
           if not byte_list: return packet_bytes # Should not happen if packet_bytes is not
   empty

           original_hex = packet_bytes.hex() # For logging

           for _ in range(num_bits_to_flip):
               if not byte_list: break
               byte_index = random.randint(0, len(byte_list) - 1)
               bit_index = random.randint(0, 7)
               byte_list[byte_index] ^= (1 << bit_index) # Flip a random bit
   ```

```
            corrupted_hex = bytes(byte_list).hex()
            print(f"[DEBUG CORRUPT] Artificially corrupted packet from {original_hex} to
        {corrupted_hex}")
            return bytes(byte_list)
        return packet_bytes
```

2. **Track Statistics:**
   - Total packets sent/received.
   - Number of packets artificially corrupted.
   - Number of packets detected as corrupt by your checksum.

```
# Example stats dictionary
# stats = {
#    'packets_received_raw': 0,
#    'artificially_corrupted': 0,
#    'detected_corruptions_on_artificial_errors': 0,
#    'detected_natural_corruptions': 0, # Checksum failed on non-artificially-corrupted
#    'successfully_processed_packets': 0,
#    'out_of_sequence_packets': 0
# }

# In your receiving loop (client or server) after recv_all():
# raw_bytes_from_socket = recv_all(...)
# if raw_bytes_from_socket:
#    stats['packets_received_raw'] += 1
#    bytes_for_unpacking = corrupt_packet_bytes_for_test(raw_bytes_from_socket)
#    was_artificially_corrupted = (bytes_for_unpacking != raw_bytes_from_socket)
#    if was_artificially_corrupted:
#       stats['artificially_corrupted'] += 1
#
#    seq, p_type, payload, is_checksum_valid_and_in_order =
process_received_packet(bytes_for_unpacking, expected_seq_num)
#    # process_received_packet would call unpack_and_verify_packet and then check
sequence.
#    # It would return a more comprehensive status or specific error codes.
#
#    # For simplicity, let's assume unpack_and_verify_packet is called directly:
#    seq, p_type, payload, is_checksum_valid =
unpack_and_verify_packet(bytes_for_unpacking)
#
#    if not is_checksum_valid:
#       if was_artificially_corrupted:
#          stats['detected_corruptions_on_artificial_errors'] += 1
#       else:
#          stats['detected_natural_corruptions'] += 1
#    elif seq != expected_seq_num: # Checksum is valid, but out of order
```

```
#       stats['out_of_sequence_packets'] += 1
#       # Handle out_of_sequence based on policy (e.g. discard, NACK)
#    else: # Checksum valid and in order
#       stats['successfully_processed_packets'] += 1
#       expected_seq_num +=1 # Update expected sequence number
#       # ... process the payload ...
```

Important Considerations:

- **Blocking Sockets and Timeouts:** The recv_all helper assumes blocking sockets. If you set socket timeouts (sock.settimeout(seconds)), recv() can raise socket.timeout, which recv_all should handle gracefully (as shown in the updated recv_all).
- **Thread Safety:** If your server uses threads for multiple clients, ensure player_sessions and sequence number access is thread-safe. Accessing and modifying dictionary entries for a specific client socket (key) from the thread dedicated to that client is generally safe after the initial insertion of the key. If multiple threads could modify the same client's session data concurrently (which shouldn't typically happen if one thread handles one client), locks would be needed.
- **ACK/NACK and Reliability:** Implementing a full ACK/NACK system for reliable delivery (like TCP's mechanisms described on Lecture p.19-20, 33-36) adds significant complexity (timers for retransmission, buffering unACKed packets, managing sliding windows). For this assignment, clearly state the level of reliability your protocol aims for.