Raw Sockets in C (POSIX/Linux): A Detailed Implementation-Focused Guide

1. Raw Sockets vs. Standard Sockets: The Fundamental Shift

You're comfortable with `SOCK_STREAM` (TCP) and `SOCK_DGRAM` (UDP). Think of these as providing a "service" at the Transport Layer (Layer 4). You give them data, an address, and a port, and the OS handles the underlying IP details (Layer 3) and Ethernet details (Layer 2).

Raw Sockets (`SOCK_RAW`) fundamentally change the game. They operate primarily at the **Network Layer (Layer 3 - IP)**.

- **Loss of Abstraction:** You lose the convenience of the OS managing transport protocols. No automatic segmentation/reassembly (like TCP), no built-in port multiplexing (like TCP/UDP).
- **Gain Control:** You gain the ability to directly interact with IP or other protocols that sit directly on top of IP.
- **Responsibility:** You become responsible for tasks the OS previously handled, most notably constructing parts (or all) of the IP header itself.

Why is this necessary for your CLDP Assignment?

Your Custom Lightweight Discovery Protocol (CLDP) is specified to run *directly* over IP, identified by a custom IP protocol number (e.g., 253). It's *not* TCP (protocol 6) or UDP (protocol 17). Raw sockets are the standard mechanism in POSIX C to send and receive packets identified by such custom IP protocol numbers.

The Sudo Imperative: Because raw sockets provide low-level network access that can be used to bypass normal OS network stack processing and potentially interfere with network operations or security, creating them almost universally requires **root privileges**. Expect to run your compiled CLDP client and server using `sudo`. If you get a "Permission denied" or "Operation not permitted" error (`EPERM` or `EACCES` in `errno`), lack of `sudo` is the most likely cause.

2. Creating the Raw Socket Endpoint

The starting point looks familiar, but the parameters are key.

```c
// <sys/socket.h>, <netinet/in.h> needed
int protocol_number = 253; // Your custom protocol for CLDP
int raw_socket_fd;

raw_socket_fd = socket(AF_INET, SOCK_RAW, protocol_number);
// Check raw_socket_fd < 0 for errors (perror("socket failed"))
// Remember to check errno for EPERM/EACCES if it fails!
```

- **`AF_INET`**: Specifies the address family is IPv4.
- **`SOCK_RAW`**: This is the crucial type specifier, requesting raw IP access.

- **`protocol` (e.g., 253):** This integer is critically important for **filtering received packets**.
  - When you call `recvfrom` on this socket, the kernel will *primarily* deliver IP datagrams whose `protocol` field in their IP header matches this number (253 in this case).
  - **Implementation Detail:** While this is the main filtering mechanism, don’t rely on it exclusively. Network interfaces in promiscuous mode, other raw sockets, or OS quirks *might* occasionally deliver packets with different protocol numbers. **Always manually check the `protocol` field within the IP header of every received packet** to ensure it’s actually a CLDP packet you should process.
  - This `protocol` number *also* influences sending if you *don’t* use `IP_HDRINCL` (which we *will* use), as the kernel would insert this number into the IP header it generates.

3. Sending Data: Embracing Manual Packet Construction

With TCP/UDP, you `send()` data, and the kernel adds headers. With raw sockets, especially for custom protocols, you usually build the packet yourself.

3.1 The `IP_HDRINCL` Socket Option: Taking Control

This option dictates who builds the main IP header when you send data.

- **Default (`IP_HDRINCL` *not* set):** Kernel builds the IP header. You only `sendto()` the payload *after* the IP header (e.g., your CLDP header + CLDP payload). The kernel fills IP header fields (source IP, dest IP from `sendto`, protocol from `socket()`, checksum, etc.). **This is NOT suitable for CLDP** because you need your custom CLDP header immediately following the IP header, not a transport header the kernel might try to insert or expect.
- **`IP_HDRINCL` *set*:** YOU build the *entire* IP packet in your buffer, starting from the IP header itself. The kernel largely trusts the header you provide. **This is REQUIRED for your CLDP implementation.**

3.2 Enabling `IP_HDRINCL`

Use `setsockopt` *after* creating the socket:

```
// <sys/socket.h>, <netinet/ip.h> needed
int one = 1; // Option value to enable

int result = setsockopt(raw_socket_fd,
                        IPPROTO_IP,   // Level: Apply option at the IP
protocol level
                        IP_HDRINCL,   // Option Name: IP Header
Include
                        &one,         // Option Value: Pointer to '1'
(enable)
                        sizeof(one)); // Option Length
```

```
// Check result < 0 for errors (perror("setsockopt IP_HDRINCL
failed"))
```

- **`IPPROTO_IP`**: Specifies the option operates at the IP layer.
- **`IP_HDRINCL`**: The specific option to enable manual IP header construction.
- **`&one`**: A pointer to an integer value of 1, signifying "true" or "enable".

From this point on, any buffer you pass to `sendto()` on this `raw_socket_fd` *must* begin with a fully formed `struct iphdr`.

4. Deep Dive into Packet Construction (with `IP_HDRINCL`)

You need to prepare a contiguous block of memory (a `char` buffer) that represents the entire packet on the wire.

Conceptual Layout:

```
|---------------------- Buffer (`char packet_buffer[MAX_SIZE]`)
----------------------|
|
|
+-----------------+-------------------
+------------------------------------------------+
| IP Header       | CLDP Header       | Payload (CLDP Data)
|
| (struct iphdr)  | (cldp_header_t)   | (e.g., hostname string, time
info, etc.)      |
+-----------------+-------------------
+------------------------------------------------+
^                 ^                   ^
|                 |                   |
ip_header_ptr     cldp_header_ptr     payload_ptr
(buffer + 0)      (buffer + ip_hdr_len) (buffer + ip_hdr_len +
cldp_hdr_len)
```

4.1 The IP Header (`struct iphdr`)

Defined in `<netinet/ip.h>`, this structure maps directly onto the IPv4 header format. You need to fill its fields meticulously.

```
// <netinet/ip.h>, <stdint.h> needed

// Example: Pointing to the IP header part of your buffer
struct iphdr *ip_hdr = (struct iphdr *)packet_buffer;

// --- Filling Key Fields ---

// Version (4 bits): Always 4 for IPv4
ip_hdr->version = 4;

// Internet Header Length (IHL) (4 bits): Length in 32-bit words.
// Minimum and usual value is 5 (5 * 4 = 20 bytes) if no IP options
are used.
ip_hdr->ihl = 5;

// Type of Service (TOS) (8 bits): Usually 0 unless you need specific
QoS.
ip_hdr->tos = 0;

// Total Length (16 bits): **CRITICAL**. Size of ENTIRE packet (IP Hdr
+ CLDP Hdr + Payload) in bytes.
// MUST be in Network Byte Order.
uint16_t total_packet_size = sizeof(struct iphdr) +
sizeof(cldp_header_t) + actual_payload_size;
ip_hdr->tot_len = htons(total_packet_size); // Use htons!

// Identification (16 bits): Used for packet fragmentation/reassembly.
// Can be a simple counter or random value for non-fragmented packets.
// MUST be in Network Byte Order.
ip_hdr->id = htons(some_packet_id_counter++); // Use htons!

// Fragment Offset + Flags (16 bits): For fragmentation.
// For simple, non-fragmented packets (most likely your case), set to
0.
// Or set the "Don't Fragment" (DF) flag if needed: htons(0x4000)
ip_hdr->frag_off = 0; // Usually 0, careful with byte order if setting
flags.

// Time To Live (TTL) (8 bits): Limits packet lifetime (hop count).
// A common value like 64 is usually fine for local networks.
ip_hdr->ttl = 64;

// Protocol (8 bits): **CRITICAL**. Identifies the next-level
protocol.
// Set this to your custom CLDP protocol number.
ip_hdr->protocol = CLDP_PROTOCOL; // e.g., 253

// Header Checksum (16 bits): **CRITICAL**. Error detection for the IP
header *only*.
// MUST be calculated *after* all other IP header fields are set.
// The checksum field itself MUST be 0 *during* the calculation.
// MUST be in Network Byte Order (though calculation result often is).
ip_hdr->check = 0; // Set to 0 before calculating checksum
// ... (Fill other fields: saddr, daddr) ...
ip_hdr->check = ip_checksum((unsigned short *)ip_hdr, sizeof(struct
iphdr)); // Calculate checksum


// Source IP Address (saddr) (32 bits): Your machine's source IP.
// MUST be in Network Byte Order.
// Use inet_addr() or inet_pton(). Hardcoding ok for testing.
// Finding programmatically: getifaddrs() is robust but complex.
ip_hdr->saddr = inet_addr("192.168.1.100"); // Replace with actual
source IP

// Destination IP Address (daddr) (32 bits): Target machine IP or
broadcast.
// MUST be in Network Byte Order.
ip_hdr->daddr = inet_addr("192.168.1.255"); // Example: Broadcast
```

Implementation Notes on IP Header Fields:

- **Bitfields (`version`, `ihl`):** Defined using C bitfields. Their layout in memory depends on the system's endianness. While direct assignment (`ip_hdr->version = 4;`) usually works on common platforms like Linux x86, be aware of potential portability issues if targeting obscure architectures.
- **`tot_len`:** Underestimating this value can cause packet truncation. Overestimating might cause issues if it exceeds the actual buffer data sent. Get it right!
- **`saddr`:** The kernel doesn't usually verify this when `IP_HDRINCL` is set (allowing spoofing, though often restricted by network hardware/ISPs). However, for CLDP, use a *real* source IP of the sending machine so replies can come back correctly.
- **Finding Source IP:** For robust applications, `getifaddrs()` iterates through network interfaces and their addresses. For the assignment, you might:
  - Hardcode it based on your test environment.
  - Require it as a command-line argument.
  - (Advanced) Use `ioctl` with `SIOCGIFADDR` on a specific interface name (e.g., "eth0").

4.2 Byte Order: The Network Programmer's Bane

- **Host Byte Order:** How your CPU stores multi-byte numbers (e.g., `uint16_t`, `uint32_t`). Often Little-Endian on x86/x64 (least significant byte first).
- **Network Byte Order:** The standard for TCP/IP headers. Always Big-Endian (most significant byte first).

You MUST convert:

- From **Host** order **To Network** order for multi-byte fields you place *into* IP or CLDP headers before sending (`htons`, `htonl`).
- From **Network** order **To Host** order for multi-byte fields you read *from* received IP or CLDP headers before using them in your C logic (`ntohs`, `ntohl`).

```
// <netinet/in.h> or <arpa/inet.h> needed

// Preparing to send:
uint16_t host_total_len = 500;
uint32_t host_tx_id = 12345;
ip_hdr->tot_len = htons(host_total_len);
cldp_hdr->transaction_id = htonl(host_tx_id);
```

```c
// After receiving:
uint16_t received_total_len_net = ip_hdr->tot_len; // Read directly from buffer (Network order)
uint16_t usable_total_len_host = ntohs(received_total_len_net); // Convert for C logic
printf("Total length from header: %u\n", usable_total_len_host);
```

Common Pitfall: Forgetting byte order conversion leads to misinterpreted lengths, IDs, ports (if used), and invalid multi-byte checksum calculations. Use them religiously for `uint16_t` and `uint32_t` header fields. `uint8_t` fields don't need conversion.

4.3 IP Header Checksum Calculation

The IP protocol requires a checksum calculated *only* over the IP header bytes.

```c
// Standard implementation (place this utility function in your code)
unsigned short ip_checksum(unsigned short *buf, int len) {
    unsigned long sum = 0;
    // Sum up 16-bit words
    while (len > 1) {
        sum += *buf++;
        len -= 2;
    }
    // Add leftover byte if any (len == 1)
    if (len == 1) {
        sum += *(unsigned char *)buf;
    }
    // Fold 32-bit sum to 16 bits: add carrier to result
    sum = (sum >> 16) + (sum & 0xFFFF); // Add high 16 bits to low 16 bits
    sum += (sum >> 16);                // Add carry from previous addition
    // Return one's complement of sum
    return (unsigned short)(~sum);
}

// Usage (after filling all other ip_hdr fields, ensuring ip_hdr->check is 0):
ip_hdr->check = 0; // MUST be zero before calculation
ip_hdr->saddr = inet_addr(src_ip_str);
ip_hdr->daddr = inet_addr(dst_ip_str);
// ... other fields filled ...
ip_hdr->check = ip_checksum((unsigned short *)ip_hdr, ip_hdr->ihl * 4);
```

Implementation Details:

- The checksum calculation treats the header as a sequence of 16-bit integers.
- The `check` field itself must be zeroed *before* computing the sum.

- The result is the one's complement of the sum.
- Receivers perform the same calculation on the incoming header (including the received checksum). If the result is `0xFFFF` (all ones), the header is likely intact. Our `ip_checksum` function does the complement at the end, so a receiver can just call it on the received header; if the result is `0`, it's valid.

4.4 The CLDP Header (Your Definition)

Based on the assignment: minimum 8 bytes, Message Type, Payload Length, Transaction ID, Reserved.

```
// <stdint.h> needed

#pragma pack(push, 1) // Crucial: Prevent compiler padding
typedef struct {
    uint8_t msg_type;       // CLDP_MSG_HELLO, CLDP_MSG_QUERY,
CLDP_MSG_RESPONSE
    uint8_t reserved;       // Keep 0 for now
    uint16_t payload_len;   // Length of data FOLLOWING this header
(Network Byte Order!)
    uint32_t transaction_id; // Unique ID for request/response
matching (Network Byte Order!)
    // Add more fields if needed, but respects minimum 8 bytes
} cldp_header_t;
#pragma pack(pop) // Restore previous packing setting

#define CLDP_MSG_HELLO    0x01
#define CLDP_MSG_QUERY    0x02
#define CLDP_MSG_RESPONSE 0x03

// --- Filling CLDP Header ---
// Assume 'packet_buffer' exists, 'ip_hdr' points to its start
// Point to where CLDP header should begin
cldp_header_t *cldp_hdr = (cldp_header_t *)(packet_buffer +
sizeof(struct iphdr)); // Assumes IHL=5

cldp_hdr->msg_type = CLDP_MSG_QUERY;
cldp_hdr->reserved = 0;

// Length of the actual data (e.g., hostname string) that will follow
this header
uint16_t actual_payload_size = strlen(query_data) + 1;
cldp_hdr->payload_len = htons(actual_payload_size); // Use htons!

uint32_t my_transaction_id = get_unique_id(); // Generate or track
this
cldp_hdr->transaction_id = htonl(my_transaction_id); // Use htonl!
```

`#pragma pack(1)`: This compiler directive is vital. It tells the compiler *not* to insert padding bytes between struct members to align them on word boundaries. Network protocols require exact layouts; padding would break compatibility between different machines or compilers. Always bracket your network header struct definitions with `#pragma pack(push, 1)` and `#pragma pack(pop)`.

4.5 The Payload

This is the actual data relevant to your CLDP message, following the CLDP header.

- **HELLO:** Might have 0 payload length (`actual_payload_size = 0`).
- **QUERY:** Payload could indicate *what* metadata is requested.
  - Simple approach: A fixed string like "HOSTNAME TIME". Server parses this.
  - Better: Define bitmasks or codes (e.g., `0x01` for hostname, `0x02` for time) included in the payload.
- **RESPONSE:** Contains the requested data.
  - Need a defined format. If sending multiple items (hostname, time), how are they delimited? Null terminators? Prepending lengths for each piece? Keep it simple for the assignment: maybe just send one piece of data per RESPONSE, or concatenate null-terminated strings.

```c
// --- Placing Payload Data ---
// Assume 'cldp_hdr' points to the CLDP header in 'packet_buffer'
char *payload_ptr = (char *)(packet_buffer + sizeof(struct iphdr) +
sizeof(cldp_header_t));

// Example: Copying hostname into payload for a RESPONSE
char hostname_buffer[256];
gethostname(hostname_buffer, sizeof(hostname_buffer)); // Error check
needed
int hostname_len = strlen(hostname_buffer); // Don't include null
terminator in payload_len? Or do? BE CONSISTENT.
                                            // Let's assume we *do*
include null term for strings.
hostname_len++;

// Ensure you calculated space for this in total_packet_size!
memcpy(payload_ptr, hostname_buffer, hostname_len);

// Make sure cldp_hdr->payload_len was set correctly
(htons(hostname_len))
// Make sure ip_hdr->tot_len included hostname_len
```

Consistency is Key: Decide *exactly* what your payload format is for each message type and document it (`cldp_spec.pdf`). Does `payload_len` include null terminators for strings? How are multiple data items separated? The receiver must parse based on these exact rules.

5. Sending the Assembled Packet

Use `sendto()`. Even though the destination IP is *inside* your crafted IP header (because you set `IP_HDRINCL`), the kernel still needs a destination `struct sockaddr_in` to figure out Layer 2 details (like the destination MAC address via ARP).

```
// <sys/socket.h>, <netinet/in.h>, <arpa/inet.h> needed

// Assume:
// - raw_socket_fd is ready (socket created, IP_HDRINCL set)
// - packet_buffer contains the full IP/CLDP/Payload packet
// - total_packet_size is the correct total length
// - dest_ip_str holds the destination IP ("192.168.1.10" or
"192.168.1.255")

struct sockaddr_in dest_addr;
memset(&dest_addr, 0, sizeof(dest_addr));
dest_addr.sin_family = AF_INET;
// Port is ignored for raw IP, set to 0. Needed by struct definition.
dest_addr.sin_port = 0;
// Convert destination IP string to network byte order binary format
if (inet_pton(AF_INET, dest_ip_str, &dest_addr.sin_addr) <= 0) {
    perror("inet_pton failed for destination");
    // Handle error
}

ssize_t bytes_sent = sendto(raw_socket_fd,
                            packet_buffer,      // Your fully crafted
packet
                            total_packet_size,  // Total length to
send
                            0,                  // Flags (usually 0)
                            (struct sockaddr *)&dest_addr, // Dest
address for L2 routing
                            sizeof(dest_addr));

// Check bytes_sent:
// if (bytes_sent < 0) { perror("sendto failed"); }
// else if (bytes_sent != total_packet_size) { // Log warning,
unlikely but possible }
```

Implementation Idea - Broadcast: To send a `QUERY` to all nodes, use the appropriate broadcast address for your subnet (e.g., `192.168.1.255` if your network is `192.168.1.0/24`). You'll likely need `setsockopt` with `SO_BROADCAST` enabled on the socket *before* sending to a broadcast address.

```
int broadcast_enable = 1;
int result = setsockopt(raw_socket_fd, SOL_SOCKET, SO_BROADCAST,
&broadcast_enable, sizeof(broadcast_enable));
// Check result < 0
```

6. Receiving Packets

The server (and the client listening for responses) uses `recvfrom` in a loop.

```
// <sys/socket.h>, <netinet/in.h>, <arpa/inet.h>, <stdio.h>,
<string.h> needed

#define RECV_BUF_SIZE 65535 // Max possible IP packet size

char recv_buffer[RECV_BUF_SIZE];
struct sockaddr_in sender_addr; // To store who sent the packet
socklen_t sender_addr_len = sizeof(sender_addr);
ssize_t bytes_received;

// In a loop:
memset(recv_buffer, 0, RECV_BUF_SIZE); // Clear buffer before
receiving
sender_addr_len = sizeof(sender_addr); // Reset addr len each time

bytes_received = recvfrom(raw_socket_fd,
                          recv_buffer,          // Buffer for incoming
packet
                          RECV_BUF_SIZE,        // Max bytes to read
into buffer
                          0,                    // Flags (usually 0)
                          (struct sockaddr *)&sender_addr, // Fill
with sender's info
                          &sender_addr_len);    // Pass addr struct
size, updated by kernel

// Check bytes_received:
// if (bytes_received < 0) {
//     if (errno == EINTR) continue; // Interrupted by signal, try
again
//     perror("recvfrom failed");
//     break; // Exit loop on fatal error
// }

// --- Packet is now in recv_buffer, bytes_received holds its size ---
// --- Proceed to PARSE the packet ---
// You can get the sender's IP like this:
// char sender_ip_str[INET_ADDRSTRLEN];
// inet_ntop(AF_INET, &sender_addr.sin_addr, sender_ip_str,
INET_ADDRSTRLEN);
// printf("Received %zd bytes from %s\n", bytes_received,
sender_ip_str);
```

- `recvfrom` blocks until a packet matching the socket's protocol (253) arrives (or an error occurs).
- `recv_buffer` contains the *entire* IP datagram, starting with the `struct iphdr`.

- `sender_addr` is filled by the kernel using the source IP from the received packet's IP header. This is crucial for the server to know where to send the `RESPONSE`.

7. Parsing the Received Packet: Validation is Key

This mirrors packet construction but involves checks at each stage.

```
// Assume recv_buffer holds the packet, bytes_received has the length

// 1. Access IP Header
if (bytes_received < sizeof(struct iphdr)) { /* Handle error: Too
small */ }
struct iphdr *ip_hdr = (struct iphdr *)recv_buffer;

// 2. Validate IP Header
if (ip_hdr->version != 4) { /* Skip: Not IPv4 */ }
if (ip_hdr->protocol != CLDP_PROTOCOL) { /* Skip: Not our protocol
(IMPORTANT!) */ }
unsigned int ip_hdr_len = ip_hdr->ihl * 4;
if (ip_hdr_len < sizeof(struct iphdr) || ip_hdr_len > bytes_received)
{ /* Handle error: Invalid IHL or short packet */ }

// 3. Verify IP Checksum (Highly Recommended)
uint16_t received_checksum = ip_hdr->check;
ip_hdr->check = 0; // Zero out field in buffer for calculation
uint16_t calculated_checksum = ip_checksum((unsigned short *)ip_hdr,
ip_hdr_len);
ip_hdr->check = received_checksum; // Restore buffer content (optional
but good practice)
if (received_checksum != calculated_checksum) { /* Handle error:
Checksum mismatch, discard packet */ }

// 4. Access CLDP Header
if (bytes_received < ip_hdr_len + sizeof(cldp_header_t)) { /* Handle
error: Too small for CLDP header */ }
cldp_header_t *cldp_hdr = (cldp_header_t *)(recv_buffer + ip_hdr_len);

// 5. Extract CLDP Fields (Use Network-to-Host conversion!)
uint8_t msg_type = cldp_hdr->msg_type;
uint16_t payload_len_net = cldp_hdr->payload_len; // Network order
from buffer
uint16_t payload_len_host = ntohs(payload_len_net); // Host order for
logic
uint32_t transaction_id_net = cldp_hdr->transaction_id;
uint32_t transaction_id_host = ntohl(transaction_id_net);

// 6. Access Payload
char *payload_ptr = recv_buffer + ip_hdr_len + sizeof(cldp_header_t);

// 7. Validate Lengths Further
```

```
uint16_t total_len_from_ip_host = ntohs(ip_hdr->tot_len);
// Check if CLDP payload length makes sense given total lengths
if (ip_hdr_len + sizeof(cldp_header_t) + payload_len_host >
total_len_from_ip_host) { /* Handle error: Inconsistent lengths (CLDP
claims more than IP) */ }
if (ip_hdr_len + sizeof(cldp_header_t) + payload_len_host >
bytes_received) { /* Handle error: Inconsistent lengths (CLDP claims
more than received) */ }

// Determine safe length to read from payload
size_t safe_payload_read_len = payload_len_host;
// Optional stricter check: Clip payload length if it exceeds buffer
boundaries based on bytes_received
size_t max_possible_payload = (bytes_received > ip_hdr_len +
sizeof(cldp_header_t)) ? (bytes_received - ip_hdr_len -
sizeof(cldp_header_t)) : 0;
if (safe_payload_read_len > max_possible_payload) {
    // Warning: Payload length in header exceeds actual received data.
Truncating.
    safe_payload_read_len = max_possible_payload;
}


// 8. Process based on msg_type using payload_ptr and
safe_payload_read_len
// if (msg_type == CLDP_MSG_QUERY) { /* Parse query from payload */ }
// else if (msg_type == CLDP_MSG_RESPONSE) { /* Parse response from
payload */ }
// Remember to handle potential null termination within
safe_payload_read_len if expecting strings.
```

Parsing Philosophy: Be paranoid. Assume the incoming packet might be malformed, corrupted, or not even intended for you (despite socket filtering). Validate lengths, checksums, and protocol numbers at each step before accessing data further into the buffer.

8. Assignment-Specific Implementation Ideas & Techniques

- **Server: Handling QUERY & Sending RESPONSE:**
  - In the `recvfrom` loop, after successfully parsing a `CLDP_MSG_QUERY`:
    - Extract the sender's IP from the `sender_addr` filled by `recvfrom`. This is your destination IP for the response.
    - Parse the query payload (using `payload_ptr` and `safe_payload_read_len`) to see what's requested (e.g., "HOSTNAME", "TIME").
    - Fetch the requested data:
      - `gethostname(buf, size)`: `<unistd.h>`. Straightforward.

      - `gettimeofday(struct timeval *tv, NULL)`: `<sys/time.h>`. Format `tv->tv_sec`, `tv->tv_usec` into a string or binary representation for the payload.
      - `sysinfo(struct sysinfo *info)`: `<sys/sysinfo.h>`. Access `info->loads[0]` (1-min avg). Remember this is scaled; you might send the raw `unsigned long` or format it as `load / 65536.0`.
      - `/proc/stat` parsing (CPU%): More complex. Open `/proc/stat`, read the first line (`cpu ...`), parse the numeric values (user, nice, system, idle, etc.). Store them. Wait a short interval (e.g., 1 sec). Read/parse again. Calculate the *difference* in idle time and total time over the interval. `CPU Usage % = 100.0 * (1.0 - (delta_idle / delta_total))`. Requires file I/O and careful string parsing (`sscanf` or `strtok`).
    - Craft the `RESPONSE` packet buffer: Fill IP header (your IP as `saddr`, sender's IP as `daddr`, new ID, protocol 253, recalc checksum), CLDP header (`msg_type=RESPONSE`, match `transaction_id` from query, calculate `payload_len`), copy metadata into payload.
    - `sendto` the response back to the sender's address.
- **Server: Sending Periodic HELLO:**
  - Outside the main `recvfrom` blocking call, or in a separate thread/process (more complex), use a timer.
  - Simple way: In the main loop, after `recvfrom` (or if it times out using `SO_RCVTIMEO` socket option), check if enough time has passed (e.g., 10 seconds using `gettimeofday` to track last HELLO time).
  - If time, craft a `HELLO` packet (IP header with broadcast dest? CLDP header `msg_type=HELLO`, `payload_len=0`).
  - `sendto` the `HELLO` packet (remember `SO_BROADCAST` option if using broadcast address).
- **Client: Sending QUERY & Receiving RESPONSE:**
  - Craft the `QUERY` packet (IP header with server/broadcast dest, CLDP header `msg_type=QUERY`, generate unique `transaction_id`, potentially add payload specifying requested data).
  - `sendto` the query.
  - Enter a `recvfrom` loop to wait for `RESPONSE` packets.
  - Use a timeout on `recvfrom` (`setsockopt` with `SO_RCVTIMEO`) so the client doesn't wait forever if no servers respond.
  - Parse incoming packets, specifically looking for `CLDP_MSG_RESPONSE`.
  - (Optional) Check if the `transaction_id` in the response matches the one sent in the query.
  - Extract and display the metadata from the response payload.

- **Transaction IDs:** Generate unique IDs for `QUERY` messages (e.g., increment a counter, use `rand()`). When the server sends a `RESPONSE`, it should copy the `transaction_id` from the `QUERY` it's responding to. This allows the client to match responses to its requests, especially if it sends multiple queries.
- **Debugging with `tcpdump`:** This is invaluable. `sudo tcpdump -i <interface_name> -nvv -X ip proto 253`
  - `-i <interface_name>`: Listen on a specific interface (e.g., `eth0`, `wlan0`, or `any`).
  - `-n`: Don't resolve hostnames (show raw IPs).
  - `-vv`: Very verbose output (shows more IP header details).
  - `-X`: Show packet content in hex and ASCII.
  - `ip proto 253`: Filter *only* for your CLDP packets. Examine the hex output carefully to verify your header fields (byte order!), lengths, and checksums match what you intended.

This detailed walkthrough, focusing on the implementation steps and potential pitfalls, should provide a solid foundation for tackling the CLDP assignment using raw sockets. Remember to be meticulous, test incrementally, and leverage debugging tools like `tcpdump`.

Here is a sample program for raw sockets :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>      // close(), getpid()
#include <errno.h>       // errno
#include <time.h>        // time() for seeding rand()

// Networking includes
#include <sys/socket.h>  // socket(), setsockopt(), recvfrom(),
sendto()
#include <netinet/in.h>  // sockaddr_in, IPPROTO_IP, htons(),
htonl(), ntohs(), ntohl()
#include <netinet/ip.h>  // struct iphdr, IP_HDRINCL
#include <arpa/inet.h>   // inet_addr(), inet_ntop()

// --- Configuration ---
#define SPP_PROTOCOL 254    // Custom IP Protocol Number (use an
unused one)
#define BUF_MAX 65535       // Maximum possible IP packet size

// --- Custom Protocol Definition (SPP - Simple Packet Protocol) ---
#define SPP_MSG_PING 0x01
#define SPP_MSG_PONG 0x02

// Ensure no padding is added by the compiler
#pragma pack(push, 1)
```

```c
typedef struct {
    uint8_t  msg_type;      // PING or PONG
    uint8_t  reserved;      // Reserved for future use (must be 0)
    uint16_t payload_len;   // Length of data following this header
(Network Byte Order!)
    uint32_t sequence;      // Sequence number (Network Byte Order!)
} spp_header_t;
#pragma pack(pop)

// --- Utility Functions ---

// Simple error handling wrapper
void die(const char *message) {
    perror(message);
    exit(EXIT_FAILURE);
}

// Calculate IP header checksum
// (Standard algorithm)
unsigned short ip_checksum(unsigned short *buf, int len) {
    unsigned long sum = 0;
    while (len > 1) {
        sum += *buf++;
        len -= 2;
    }
    if (len == 1) { // Odd byte
        sum += *(unsigned char *)buf;
    }
    // Fold carry bits
    sum = (sum >> 16) + (sum & 0xFFFF); // Add high 16 to low 16
    sum += (sum >> 16);                 // Add carry again
    return (unsigned short)(~sum);      // Return 1's complement
}

// --- Packet Parsing Function (Receiver Side) ---
void parse_spp_packet(int sock_fd, const char *local_ip_str, char
*buffer, ssize_t len, struct sockaddr_in *sender_addr) {
    // 1. Access IP Header
    if (len < sizeof(struct iphdr)) {
        fprintf(stderr, "Received packet too small for IP header (%zd
bytes)\n", len);
        return;
    }
    struct iphdr *ip_hdr = (struct iphdr *)buffer;

    // 2. Validate IP Header Basics
    if (ip_hdr->version != 4) {
        fprintf(stderr, "Received non-IPv4 packet (version %u),
skipping.\n", ip_hdr->version);
        return;
    }
    // **Crucially, double-check the protocol even though the socket
filters**
    if (ip_hdr->protocol != SPP_PROTOCOL) {
        fprintf(stderr, "Received packet with protocol %u, expected
%d. Skipping.\n",
                ip_hdr->protocol, SPP_PROTOCOL);
        return;
    }

    // Calculate exact IP header length (IHL is in 32-bit words)
    unsigned int ip_hdr_len = ip_hdr->ihl * 4;
    if (ip_hdr_len < sizeof(struct iphdr) || ip_hdr_len > len) {
        fprintf(stderr, "Invalid IP header length: %u bytes (received
total %zd)\n", ip_hdr_len, len);
        return;
    }

    // 3. Verify IP Checksum (Recommended)
    uint16_t received_checksum = ip_hdr->check;
    ip_hdr->check = 0; // Temporarily zero out checksum field in
buffer for recalculation
    uint16_t calculated_checksum = ip_checksum((unsigned short
*)ip_hdr, ip_hdr_len);
    ip_hdr->check = received_checksum; // Restore original checksum in
buffer

    if (received_checksum != calculated_checksum) {
        fprintf(stderr, "IP header checksum mismatch! (Received=0x%x,
Calculated=0x%x). Discarding packet.\n",
                ntohs(received_checksum), ntohs(calculated_checksum));
        return;
    }

    // 4. Access SPP Header (immediately follows IP header)
    if (len < ip_hdr_len + sizeof(spp_header_t)) {
        fprintf(stderr, "Received packet too small for SPP header (%zd
bytes, need %lu)\n",
                len, (unsigned long)(ip_hdr_len +
sizeof(spp_header_t)));
        return;
    }
    spp_header_t *spp_hdr = (spp_header_t *)(buffer + ip_hdr_len);

    // 5. Extract SPP Fields (Convert from Network to Host byte order)
    uint8_t msg_type = spp_hdr->msg_type;
    uint16_t payload_len_host = ntohs(spp_hdr->payload_len);
    uint32_t sequence_host = ntohl(spp_hdr->sequence);

    // 6. Access Payload (immediately follows SPP header)
    char *payload_ptr = buffer + ip_hdr_len + sizeof(spp_header_t);

    // 7. Validate Lengths consistency
    uint16_t total_len_from_ip_host = ntohs(ip_hdr->tot_len);
    uint16_t expected_len_from_spp = ip_hdr_len + sizeof(spp_header_t)
+ payload_len_host;

    if (total_len_from_ip_host != expected_len_from_spp) {
         fprintf(stderr, "Warning: IP total length (%u) doesn't match
SPP calculated length (%u)\n",
                 total_len_from_ip_host, expected_len_from_spp);
        // Decide how to handle: trust IP header length? SPP length?
Smallest?
        // For simplicity, we might trust the SPP header for payload
length, but ensure
        // we don't read beyond the actual received bytes (`len`).
    }
     if (expected_len_from_spp > len) {
         fprintf(stderr, "Error: SPP header indicates length (%u)
greater than received bytes (%zd). Discarding.\n",
                 expected_len_from_spp, len);
         return;
     }

    // Determine safe length to read from payload
    size_t safe_payload_len = payload_len_host;
    size_t max_possible_payload = len - ip_hdr_len -
sizeof(spp_header_t);
    if (safe_payload_len > max_possible_payload) {
        fprintf(stderr, "Warning: Payload length in header (%u)
exceeds actual data (%zu). Truncating read.\n",
                payload_len_host, max_possible_payload);
        safe_payload_len = max_possible_payload;
    }

    // 8. Process the Message
    char sender_ip_str[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &(sender_addr->sin_addr), sender_ip_str,
INET_ADDRSTRLEN);

    printf("=== Received SPP Packet ===\n");
    printf("  From IP: %s\n", sender_ip_str);
    printf("  IP Total Len: %u, IP Hdr Len: %u\n",
total_len_from_ip_host, ip_hdr_len);
    printf("  SPP Type: 0x%02x (%s)\n", msg_type, (msg_type ==
SPP_MSG_PING) ? "PING" : (msg_type == SPP_MSG_PONG ? "PONG" :
"Unknown"));
    printf("  SPP Seq: %u\n", sequence_host);
    printf("  SPP Payload Len (from header): %u\n", payload_len_host);
    printf("  SPP Safe Payload Len (usable): %zu\n",
safe_payload_len);

    if (safe_payload_len > 0) {
        // Print payload safely (treat as potential string, ensure
null termination)
        char temp_payload[safe_payload_len + 1];
        memcpy(temp_payload, payload_ptr, safe_payload_len);
        temp_payload[safe_payload_len] = '\0'; // Null terminate for
safety
        printf("  Payload: \"%s\"\n", temp_payload);
    } else {
        printf("  Payload: (empty)\n");
    }
     printf("===========================\n\n");


    // --- Action: If PING received, send PONG back ---
    if (msg_type == SPP_MSG_PING && local_ip_str != NULL) {
        printf("Received PING, sending PONG back to %s...\n",
sender_ip_str);

        char pong_packet[BUF_MAX];
        memset(pong_packet, 0, BUF_MAX);

        // Pointers for constructing PONG
        struct iphdr *pong_ip_hdr = (struct iphdr *)pong_packet;
        spp_header_t *pong_spp_hdr = (spp_header_t *)(pong_packet +
sizeof(struct iphdr));
        char *pong_payload_ptr = pong_packet + sizeof(struct iphdr) +
sizeof(spp_header_t);

        // Prepare PONG payload
        const char *pong_msg = "PONG_DATA";
        uint16_t pong_payload_len = strlen(pong_msg) + 1; // Include
null terminator
        memcpy(pong_payload_ptr, pong_msg, pong_payload_len);

        // Fill PONG SPP Header
        pong_spp_hdr->msg_type = SPP_MSG_PONG;
        pong_spp_hdr->reserved = 0;
        pong_spp_hdr->payload_len = htons(pong_payload_len); //
Network Byte Order
        pong_spp_hdr->sequence = htonl(sequence_host);      // Echo
sequence number (Network Byte Order)

        // Calculate total PONG packet length
        uint16_t pong_total_len = sizeof(struct iphdr) +
sizeof(spp_header_t) + pong_payload_len;
```

```
        // Fill PONG IP Header (Use received PING's source as
destination)
        pong_ip_hdr->version = 4;
        pong_ip_hdr->ihl = 5; // No options
        pong_ip_hdr->tos = 0;
        pong_ip_hdr->tot_len = htons(pong_total_len);       // Network
Byte Order
        pong_ip_hdr->id = htons(getpid() & 0xFFFF);         // Simple
ID (Network Byte Order)
        pong_ip_hdr->frag_off = 0;                          // No
fragmentation
        pong_ip_hdr->ttl = 64;
        pong_ip_hdr->protocol = SPP_PROTOCOL;
        pong_ip_hdr->check = 0; // Checksum calculated below
        pong_ip_hdr->saddr = inet_addr(local_ip_str);       // Our IP
as source
        pong_ip_hdr->daddr = ip_hdr->saddr;                 // PING
sender's IP as destination

        // Calculate PONG IP Checksum
        pong_ip_hdr->check = ip_checksum((unsigned short
*)pong_ip_hdr, sizeof(struct iphdr));

        // Prepare destination address structure for sendto()
        struct sockaddr_in pong_dest_addr;
        memset(&pong_dest_addr, 0, sizeof(pong_dest_addr));
        pong_dest_addr.sin_family = AF_INET;
        pong_dest_addr.sin_port = 0; // Port ignored for raw IP
        pong_dest_addr.sin_addr.s_addr = pong_ip_hdr->daddr; // Use
the destination from crafted header

        // Send the PONG packet
        ssize_t bytes_sent = sendto(sock_fd, pong_packet,
pong_total_len, 0,
                                    (struct sockaddr
*)&pong_dest_addr, sizeof(pong_dest_addr));

        if (bytes_sent < 0) {
            perror("sendto (PONG) failed");
        } else if (bytes_sent != pong_total_len) {
            fprintf(stderr, "Warning: sendto (PONG) sent %zd bytes,
expected %u\n", bytes_sent, pong_total_len);
        } else {
            printf("PONG sent successfully (%zd bytes).\n\n",
bytes_sent);
        }
    }
}
```

```
// --- Main Function ---
int main(int argc, char *argv[]) {

    // --- Argument Parsing ---
    if (argc < 2) {
        fprintf(stderr, "Usage:\n");
        fprintf(stderr, "  %s send <source_ip> <dest_ip>\n", argv[0]);
        fprintf(stderr, "  %s receive <local_ip_for_pong_source>\n",
argv[0]);
        exit(EXIT_FAILURE);
    }

    char *mode = argv[1];
    int send_mode = (strcmp(mode, "send") == 0);
    int receive_mode = (strcmp(mode, "receive") == 0);

    if (!send_mode && !receive_mode) {
        fprintf(stderr, "Invalid mode: %s. Use 'send' or 'receive'.\
n", mode);
        exit(EXIT_FAILURE);
    }

    char *source_ip_str = NULL;
    char *dest_ip_str = NULL;
    char *local_ip_str = NULL; // For receiver sending PONGs

    if (send_mode) {
        if (argc != 4) {
            fprintf(stderr, "Usage: %s send <source_ip> <dest_ip>\n",
argv[0]);
            exit(EXIT_FAILURE);
        }
        source_ip_str = argv[2];
        dest_ip_str = argv[3];
        printf("Mode: Send | Source: %s | Dest: %s\n", source_ip_str,
dest_ip_str);
    } else { // receive_mode
        if (argc != 3) {
             fprintf(stderr, "Usage: %s receive
<local_ip_for_pong_source>\n", argv[0]);
             exit(EXIT_FAILURE);
        }
        local_ip_str = argv[2];
        printf("Mode: Receive | Listening for SPP proto %d | PONG
Source IP: %s\n", SPP_PROTOCOL, local_ip_str);
    }


    // Seed random number generator (for packet IDs)
    srand(time(NULL));

    // --- Socket Creation ---
    // AF_INET: IPv4 Internet protocols
    // SOCK_RAW: Raw network protocol access
    // SPP_PROTOCOL: Our custom protocol number - kernel will filter
incoming packets
    int sock_fd = socket(AF_INET, SOCK_RAW, SPP_PROTOCOL);
    if (sock_fd < 0) {
        // ** Check for permission error specifically **
        if (errno == EPERM || errno == EACCES) {
             fprintf(stderr,"socket() failed: Permission denied. Did
you forget sudo?\n");
        }
        die("socket() failed");
    }
    printf("Raw socket created (fd = %d) for protocol %d.\n", sock_fd,
SPP_PROTOCOL);


    // --- Socket Options ---
    if (send_mode) {
        // ** Enable IP_HDRINCL **
        // Tell the kernel that we will provide the IP header
ourselves
        int enable = 1;
        if (setsockopt(sock_fd, IPPROTO_IP, IP_HDRINCL, &enable,
sizeof(enable)) < 0) {
            die("setsockopt(IP_HDRINCL) failed");
        }
        printf("IP_HDRINCL option enabled.\n");

         // Optional: Enable broadcasting if sending to a broadcast
address
        // if (inet_addr(dest_ip_str) == INADDR_BROADCAST || ...) { //
Check if dest is broadcast
        //     if (setsockopt(sock_fd, SOL_SOCKET, SO_BROADCAST,
&enable, sizeof(enable)) < 0) {
        //         die("setsockopt(SO_BROADCAST) failed");
        //     }
        //     printf("SO_BROADCAST option enabled.\n");
        // }
    }

    //
======================================================================
==
    // --- Sending Logic ---
    //
======================================================================
==
    if (send_mode) {
        char packet[BUF_MAX];
        memset(packet, 0, BUF_MAX);

        // Pointers to different parts of the packet buffer
        struct iphdr *ip_hdr = (struct iphdr *)packet;
        // SPP header starts right after the IP header (assuming
ihl=5, 20 bytes)
        spp_header_t *spp_hdr = (spp_header_t *)(packet +
sizeof(struct iphdr));
        // Payload starts right after the SPP header
        char *payload_ptr = packet + sizeof(struct iphdr) +
sizeof(spp_header_t);

        // 1. Prepare Payload Data
        const char *ping_msg = "PING_DATA";
        // Include null terminator in length for string payload
        uint16_t payload_len = strlen(ping_msg) + 1;
        memcpy(payload_ptr, ping_msg, payload_len);

        // 2. Fill SPP Header
        spp_hdr->msg_type = SPP_MSG_PING;
        spp_hdr->reserved = 0;
        spp_hdr->payload_len = htons(payload_len); // Network Byte
Order!
        spp_hdr->sequence = htonl(12345);          // Example sequence
(Network Byte Order!)

        // 3. Calculate Total Packet Length
        uint16_t total_packet_len = sizeof(struct iphdr) +
sizeof(spp_header_t) + payload_len;

        // 4. Fill IP Header (IP_HDRINCL is enabled)
        ip_hdr->version = 4;            // IPv4
        ip_hdr->ihl = 5;                // Header length in 32-bit
words (5 * 4 = 20 bytes, no options)
        ip_hdr->tos = 0;                // Type of Service (usually 0)
        ip_hdr->tot_len = htons(total_packet_len); // Total length
(Network Byte Order!)
        ip_hdr->id = htons(rand() % 0xFFFF); // Packet ID (random)
(Network Byte Order!)
        ip_hdr->frag_off = 0;           // Fragmentation flags/offset
(0 for simple packets)
        ip_hdr->ttl = 64;               // Time To Live
        ip_hdr->protocol = SPP_PROTOCOL;// Our custom protocol number
        ip_hdr->check = 0;              // Checksum placeholder
(calculated below)
        // Source IP Address (Convert from string to network byte
order binary)
```

```
        if (inet_pton(AF_INET, source_ip_str, &(ip_hdr->saddr)) != 1)
{
             fprintf(stderr, "Invalid source IP address: %s\n",
source_ip_str); close(sock_fd); exit(EXIT_FAILURE);
        }
        // Destination IP Address
        if (inet_pton(AF_INET, dest_ip_str, &(ip_hdr->daddr)) != 1) {
             fprintf(stderr, "Invalid destination IP address: %s\n",
dest_ip_str); close(sock_fd); exit(EXIT_FAILURE);
        }

        // 5. Calculate IP Header Checksum (MUST be done after all
other IP fields are set)
        ip_hdr->check = ip_checksum((unsigned short *)ip_hdr,
sizeof(struct iphdr)); // ihl * 4 would also work


        // 6. Prepare Destination Address Structure for sendto()
        // Even with IP_HDRINCL, sendto needs this for routing (e.g.,
finding MAC via ARP)
        struct sockaddr_in dest_addr;
        memset(&dest_addr, 0, sizeof(dest_addr));
        dest_addr.sin_family = AF_INET;
        dest_addr.sin_port = 0; // Port is ignored for raw IP sockets
but required by struct
        dest_addr.sin_addr.s_addr = ip_hdr->daddr; // Use the
destination IP we put in the header

        printf("Constructed PING packet (%u bytes total). Sending to
%s...\n", total_packet_len, dest_ip_str);

        // 7. Send the Packet!
        ssize_t bytes_sent = sendto(sock_fd,
                                    packet,             // Buffer with
the FULL packet
                                    total_packet_len,   // Total size
to send
                                    0,                  // Flags
(usually 0)
                                    (struct sockaddr *)&dest_addr, //
Destination for routing
                                    sizeof(dest_addr)); // Size of
address structure

        if (bytes_sent < 0) {
            die("sendto failed");
        } else if (bytes_sent != total_packet_len) {
            fprintf(stderr, "Warning: sendto sent %zd bytes, but
expected %u bytes\n", bytes_sent, total_packet_len);
        } else {
            printf("Successfully sent %zd bytes (PING).\n",
bytes_sent);
        }

    }
    //
========================================================================
==
    // --- Receiving Logic ---
    //
========================================================================
==
    else if (receive_mode) {
        char recv_buffer[BUF_MAX];
        struct sockaddr_in sender_addr; // To store who sent the
packet
        socklen_t sender_addr_len;

        printf("\nWaiting to receive SPP packets (proto %d)...\n",
SPP_PROTOCOL);

        while (1) { // Loop forever to receive packets
            memset(recv_buffer, 0, BUF_MAX);
            sender_addr_len = sizeof(sender_addr); // Reset length
each time

            // Block until a packet for our protocol arrives
            ssize_t bytes_received = recvfrom(sock_fd,
                                              recv_buffer,        //
Buffer for incoming data
                                              BUF_MAX,            //
Max size of buffer
                                              0,                  //
Flags
                                              (struct sockaddr
*)&sender_addr, // Get sender's info
                                              &sender_addr_len);  //
Get sender addr struct size

            if (bytes_received < 0) {
                // If interrupted by signal (e.g., Ctrl+C), continue
or break? Let's break.
                if (errno == EINTR) {
                    printf("Interrupted. Exiting.\n");
                    break;
                }
                perror("recvfrom failed");
                // Consider whether to break or continue on other
errors
                continue; // Example: Continue listening despite error
on one packet
            }

            // --- Packet Received - Parse It ---
            // Pass socket fd and local IP so parser can send PONG if
needed
            parse_spp_packet(sock_fd, local_ip_str, recv_buffer,
bytes_received, &sender_addr);

        } // End while loop
    }

    // --- Cleanup ---
    printf("Closing socket.\n");
    close(sock_fd);
    return 0;
}
```