```
#include <stdio.h>
#include <strings.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include<netinet/in.h>
#define PORT 5000
#define MAXLINE 1000

// Driver code
int main()
{
      char buffer[100];
      char *message = "Hello Client";
      int listenfd, len;
      struct sockaddr_in servaddr, cliaddr;
      bzero(&servaddr, sizeof(servaddr));

      // Create a UDP Socket
      listenfd = socket(AF_INET, SOCK_DGRAM, 0);
      servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
      servaddr.sin_port = htons(PORT);
      servaddr.sin_family = AF_INET;

      // bind server address to socket descriptor
      bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr));

      //receive the datagram
      len = sizeof(cliaddr);
      int n = recvfrom(listenfd, buffer, sizeof(buffer),
                 0, (struct sockaddr*)&cliaddr,&len); //receive message
from server
      buffer[n] = '\0';
      puts(buffer);

      // send the response
      sendto(listenfd, message, MAXLINE, 0,
            (struct sockaddr*)&cliaddr, sizeof(cliaddr));
}
```

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

#define POLYNOMIAL 0xEDB88320 // CRC-32 polynomial
#define INITIAL_REMAINDER 0xFFFFFFFF
#define FINAL_XOR_VALUE 0xFFFFFFFF

uint32_t crc_table[256];

// Initialize the CRC table
void crc32_init() {
    uint32_t remainder;
    for (int i = 0; i < 256; i++) {
        remainder = i;
        for (int j = 8; j > 0; j--) {
            if (remainder & 1) {
                remainder = (remainder >> 1) ^ POLYNOMIAL;
            } else {
                remainder = (remainder >> 1);
            }
        }
        crc_table[i] = remainder;
    }
}

// Compute CRC for the given data
uint32_t crc32_compute(const uint8_t *data, size_t length) {
    uint32_t crc = INITIAL_REMAINDER;
    while (length--) {
        uint8_t table_index = (crc ^ *data++) & 0xFF;
        crc = (crc >> 8) ^ crc_table[table_index];
    }
    return crc ^ FINAL_XOR_VALUE;
}

// Example function to simulate data transmission
void simulate_data_transmission(const char *data) {
    uint32_t crc = crc32_compute((const uint8_t *)data, strlen(data));
    printf("Original data: %s\n", data);
    printf("Computed CRC-32: %08X\n", crc);

    // Simulate data reception and CRC check
    printf("Verifying data...\n");
    uint32_t received_crc = crc32_compute((const uint8_t *)data,
strlen(data));
    if (received_crc == crc) {
        printf("Data is correct. CRC-32 match.\n");
    } else {
        printf("Data is incorrect. CRC-32 mismatch.\n");
    }
}

int main() {
    crc32_init(); // Initialize CRC table

    const char *data = "Hello, CRC!";
    simulate_data_transmission(data);

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <fcntl.h>
#include <sys/stat.h>

#define PORT 12345
#define BUFFER_SIZE 1024

void handle_server() {
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    char buffer[BUFFER_SIZE];
    ssize_t bytes_received;
    int file_fd;

    // Create a TCP socket
    if ((server_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Set up server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind the socket
    if (bind(server_sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("bind");
        close(server_sock);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_sock, 1) < 0) {
        perror("listen");
        close(server_sock);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d\n", PORT);

    // Accept a client connection
    if ((client_sock = accept(server_sock, (struct sockaddr
*)&client_addr, &client_addr_len)) < 0) {
        perror("accept");
        close(server_sock);
        exit(EXIT_FAILURE);
    }

    // Open file for writing
    file_fd = open("received_file", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file_fd < 0) {
        perror("open");
        close(client_sock);
        close(server_sock);
        exit(EXIT_FAILURE);
    }

    // Receive file data from client
    while ((bytes_received = recv(client_sock, buffer, BUFFER_SIZE, 0)) >
0) {
        if (write(file_fd, buffer, bytes_received) < 0) {
            perror("write");
            close(file_fd);
            close(client_sock);
            close(server_sock);
            exit(EXIT_FAILURE);
        }
    }

    if (bytes_received < 0) {
        perror("recv");
    }

    printf("File received and saved as 'received_file'\n");

    close(file_fd);
    close(client_sock);
    close(server_sock);
}

void handle_client(const char *filename) {
    int sock;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];
    ssize_t bytes_sent, bytes_read;
    int file_fd;

    // Create a TCP socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Set up server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);

    // Convert server IP address
    if (inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr) <= 0) {
        perror("inet_pton");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("connect");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Open the file to be sent
    file_fd = open(filename, O_RDONLY);
    if (file_fd < 0) {
        perror("open");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Send file data to the server
    while ((bytes_read = read(file_fd, buffer, BUFFER_SIZE)) > 0) {
        bytes_sent = send(sock, buffer, bytes_read, 0);
        if (bytes_sent < 0) {
            perror("send");
            close(file_fd);
            close(sock);
            exit(EXIT_FAILURE);
        }
    }

    if (bytes_read < 0) {
        perror("read");
    }

    printf("File '%s' sent successfully\n", filename);

    close(file_fd);
    close(sock);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <server/client> [filename]\n",
argv[0]);
        exit(EXIT_FAILURE);
    }

    if (strcmp(argv[1], "server") == 0) {
        handle_server();
    } else if (strcmp(argv[1], "client") == 0) {
        if (argc != 3) {
            fprintf(stderr, "Usage: %s client <filename>\n", argv[0]);
            exit(EXIT_FAILURE);
        }
        handle_client(argv[2]);
    } else {
        fprintf(stderr, "Invalid argument. Use 'server' or 'client'.\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h> // read(), write(), close()
#define MAX 80
#define PORT 8080
#define SA struct sockaddr

// Function designed for chat between client and server.
void func(int connfd)
{
      char buff[MAX];
      int n;
      // infinite loop for chat
      for (;;) {
            bzero(buff, MAX);

            // read the message from client and copy it in buffer
            read(connfd, buff, sizeof(buff));
            // print buffer which contains the client contents
            printf("From client: %s\t To client : ", buff);
            bzero(buff, MAX);
            n = 0;
            // copy server message in the buffer
            while ((buff[n++] = getchar()) != '\n')
                  ;

            // and send that buffer to client
            write(connfd, buff, sizeof(buff));

            // if msg contains "Exit" then server exit and chat ended.
            if (strncmp("exit", buff, 4) == 0) {
                  printf("Server Exit...\n");
                  break;
            }
      }
}

// Driver function
int main()
{
      int sockfd, connfd, len;
      struct sockaddr_in servaddr, cli;

      // socket create and verification
      sockfd = socket(AF_INET, SOCK_STREAM, 0);
      if (sockfd == -1) {
            printf("socket creation failed...\n");
            exit(0);
      }
      else
            printf("Socket successfully created..\n");
      bzero(&servaddr, sizeof(servaddr));

      // assign IP, PORT
      servaddr.sin_family = AF_INET;
```

```
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(PORT);

    // Binding newly created socket to given IP and verification
    if ((bind(sockfd, (SA*)&servaddr, sizeof(servaddr))) != 0) {
        printf("socket bind failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully binded..\n");

    // Now server is ready to listen and verification
    if ((listen(sockfd, 5)) != 0) {
        printf("Listen failed...\n");
        exit(0);
    }
    else
        printf("Server listening..\n");
    len = sizeof(cli);

    // Accept the data packet from client and verification
    connfd = accept(sockfd, (SA*)&cli, &len);
    if (connfd < 0) {
        printf("server accept failed...\n");
        exit(0);
    }
    else
        printf("server accept the client...\n");

    // Function for chatting between client and server
    func(connfd);

    // After chatting close the socket
    close(sockfd);
}
```

```c
#include<stdio.h>
int main()
{
  int w,i,f,frames[50];
  printf("Enter window size: ");
  scanf("%d",&w);
  printf("\nEnter number of frames to transmit: ");
  scanf("%d",&f);
  printf("\nEnter %d frames: ",f);
  for(i=1;i<=f;i++)
    scanf("%d",&frames[i]);
  printf("\nWith sliding window protocol the frames will be sent in the
following manner (assuming no corruption of frames)\n\n");
  printf("After sending %d frames at each stage sender waits for
acknowledgement sent by the receiver\n\n",w);
  for(i=1;i<=f;i++)
  {
    if(i%w==0)
    {
      printf("%d\n",frames[i]);
      printf("Acknowledgement of above frames sent is received by
sender\n\n");
    }
    else
      printf("%d ",frames[i]);
  }
  if(f%w!=0)
    printf("\nAcknowledgement of above frames sent is received by
sender\n");
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/ip.h>
#include <netinet/if_ether.h>
#include <netinet/if_arp.h>
#include <sys/ioctl.h>

#define ARP_REQUEST 1
#define ARP_REPLY 2

void send_arp_request(const char *interface, const char *target_ip) {
    int sockfd;
    struct ifreq ifr;
    struct ether_arp arp_req;
    struct sockaddr_ll sa;
    struct ethhdr eth_hdr;
    unsigned char src_mac[6];
    unsigned char dst_mac[6] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff}; //
Broadcast MAC
    struct sockaddr_ll sll;
    char packet[sizeof(struct ethhdr) + sizeof(struct ether_arp)];

    // Create raw socket
    if ((sockfd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ARP))) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Get source MAC address
    memset(&ifr, 0, sizeof(ifr));
    strncpy(ifr.ifr_name, interface, IFNAMSIZ-1);
    if (ioctl(sockfd, SIOCGIFHWADDR, &ifr) < 0) {
        perror("ioctl");
        close(sockfd);
        exit(EXIT_FAILURE);
    }
    memcpy(src_mac, ifr.ifr_hwaddr.sa_data, 6);

    // Prepare Ethernet header
    memcpy(eth_hdr.h_dest, dst_mac, 6);
    memcpy(eth_hdr.h_source, src_mac, 6);
    eth_hdr.h_proto = htons(ETH_P_ARP);

    // Prepare ARP request
    memset(&arp_req, 0, sizeof(arp_req));
    arp_req.arp_hrd = htons(ARPHRD_ETHER);
    arp_req.arp_pro = htons(ETH_P_IP);
    arp_req.arp_hln = 6;
    arp_req.arp_pln = 4;
    arp_req.arp_op = htons(ARP_REQUEST);
    memcpy(arp_req.arp_sha, src_mac, 6);
    memset(arp_req.arp_tha, 0, 6);
    inet_pton(AF_INET, "0.0.0.0", arp_req.arp_spa);
    inet_pton(AF_INET, target_ip, arp_req.arp_tpa);
    // Construct packet
    memcpy(packet, &eth_hdr, sizeof(struct ethhdr));
    memcpy(packet + sizeof(struct ethhdr), &arp_req, sizeof(struct
ether_arp));

    // Get the index of the network interface
    memset(&sll, 0, sizeof(sll));
    sll.sll_family = AF_PACKET;
    sll.sll_ifindex = if_nametoindex(interface);

    // Send ARP request
    if (sendto(sockfd, packet, sizeof(packet), 0, (struct sockaddr
*)&sll, sizeof(sll)) < 0) {
        perror("sendto");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    printf("ARP request sent for IP: %s\n", target_ip);
    close(sockfd);
}

int main() {
    const char *interface = "eth0"; // Change this to your network
interface
    const char *target_ip = "192.168.1.1"; // Change this to the IP you
want to resolve

    send_arp_request(interface, target_ip);

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 12345
#define BUFFER_SIZE 1024

void handle_client(int client_sock) {
    char buffer[BUFFER_SIZE];
    ssize_t bytes_received;

    // Loop to handle client communication
    while ((bytes_received = recv(client_sock, buffer, BUFFER_SIZE, 0)) >
0) {
        // Echo back the received data
        send(client_sock, buffer, bytes_received, 0);
    }

    if (bytes_received < 0) {
        perror("recv");
    }

    close(client_sock);
}

void run_server() {
    int server_sock, client_sock;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);

    // Create a TCP socket
    if ((server_sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Set up server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(PORT);

    // Bind the socket
    if (bind(server_sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("bind");
        close(server_sock);
        exit(EXIT_FAILURE);
    }

    // Listen for incoming connections
    if (listen(server_sock, 5) < 0) {
        perror("listen");
        close(server_sock);
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d\n", PORT);
    // Accept and handle incoming connections
    while ((client_sock = accept(server_sock, (struct sockaddr
*)&client_addr, &client_addr_len)) >= 0) {
        printf("Client connected\n");
        handle_client(client_sock);
    }

    if (client_sock < 0) {
        perror("accept");
    }

    close(server_sock);
}

void run_client(const char *server_ip) {
    int sock;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];
    ssize_t bytes_sent, bytes_received;

    // Create a TCP socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Set up server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);

    // Convert server IP address
    if (inet_pton(AF_INET, server_ip, &server_addr.sin_addr) <= 0) {
        perror("inet_pton");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Connect to the server
    if (connect(sock, (struct sockaddr *)&server_addr,
sizeof(server_addr)) < 0) {
        perror("connect");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Read input from the user
    printf("Enter message to send: ");
    fgets(buffer, BUFFER_SIZE, stdin);
    buffer[strcspn(buffer, "\n")] = '\0'; // Remove newline character

    // Send data to the server
    if ((bytes_sent = send(sock, buffer, strlen(buffer), 0)) < 0) {
        perror("send");
        close(sock);
        exit(EXIT_FAILURE);
    }

    // Receive and print the echoed data
    if ((bytes_received = recv(sock, buffer, BUFFER_SIZE - 1, 0)) < 0) {
        perror("recv");
        close(sock);
        exit(EXIT_FAILURE);
    }

    buffer[bytes_received] = '\0'; // Null-terminate the received data
    printf("Received echo: %s\n", buffer);

    close(sock);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <server/client> [server_ip]\n", 
argv[0]);
        exit(EXIT_FAILURE);
    }

    if (strcmp(argv[1], "server") == 0) {
        run_server();
    } else if (strcmp(argv[1], "client") == 0) {
        if (argc != 3) {
            fprintf(stderr, "Usage: %s client <server_ip>\n", argv[0]);
            exit(EXIT_FAILURE);
        }
        run_client(argv[2]);
    } else {
        fprintf(stderr, "Invalid argument. Use 'server' or 'client'.\n");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdint.h>

#define DNS_PORT 53
#define BUFFER_SIZE 512

// DNS Header Structure
typedef struct {
    uint16_t id; // Identification
    uint16_t flags; // Flags
    uint16_t qdcount; // Number of questions
    uint16_t ancount; // Number of answers
    uint16_t nscount; // Number of authority records
    uint16_t arcount; // Number of additional records
} DNSHeader;

// DNS Question Structure
typedef struct {
    uint16_t qtype; // Query type
    uint16_t qclass; // Query class
} DNSQuestion;

// DNS Response Parser
void parse_dns_response(const char *response, size_t length) {
    // Extract the DNS Header
    const DNSHeader *header = (const DNSHeader *)response;

    // Start parsing from the end of the questions section
    const char *ptr = response + sizeof(DNSHeader);

    // Skip over the questions section
    for (int i = 0; i < ntohs(header->qdcount); ++i) {
        while (*ptr != 0) ++ptr; // Skip domain name
        ptr += 5; // Skip NULL byte and QTYPE/QCLASS
    }

    // Parse Answers
    for (int i = 0; i < ntohs(header->ancount); ++i) {
        while (*ptr != 0) ++ptr; // Skip domain name
        ptr += 10; // Skip NULL byte and QTYPE/QCLASS

        // Extract the Type and Class
        uint16_t type = ntohs(*(uint16_t *)ptr);
        uint16_t class = ntohs(*(uint16_t *)(ptr + 2));
        uint16_t data_len = ntohs(*(uint16_t *)(ptr + 8));

        // Check if this is an A record
        if (type == 1 && class == 1) {
            ptr += 10; // Skip TYPE, CLASS, TTL, and length
            if (data_len == 4) {
                // Print IP Address
                printf("IP Address: %u.%u.%u.%u\n",
                    (unsigned char)ptr[0],
                    (unsigned char)ptr[1],
                    (unsigned char)ptr[2],
                    (unsigned char)ptr[3]);
            }
        }

        ptr += data_len;
    }
}

// Construct a DNS Query
void construct_dns_query(char *buffer, size_t *length, const char
*hostname) {
    DNSHeader *header = (DNSHeader *)buffer;
    memset(header, 0, sizeof(DNSHeader));
    header->id = htons(0x1234); // Transaction ID
    header->flags = htons(0x0100); // Standard query
    header->qdcount = htons(1); // Number of questions

    char *qname = buffer + sizeof(DNSHeader);
    char *p = qname;

    // Encode the hostname in DNS format
    const char *label = hostname;
    while (*label) {
        const char *start = label;
        while (*label && *label != '.') ++label;
        *p++ = (uint8_t)(label - start);
        memcpy(p, start, label - start);
        p += (label - start);
        if (*label) ++label; // Skip the dot
    }
    *p++ = 0; // Null byte for end of domain name

    DNSQuestion *question = (DNSQuestion *)p;
    question->qtype = htons(1); // A record
    question->qclass = htons(1); // IN (Internet)

    *length = p + sizeof(DNSQuestion) - buffer;
}

// Send DNS Query and Print Response
void dns_query(const char *dns_server_ip, const char *hostname) {
    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];
    size_t length;

    // Create UDP socket
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    // Setup server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(DNS_PORT);

    // Convert server IP address
    if (inet_pton(AF_INET, dns_server_ip, &server_addr.sin_addr) <= 0) {
        perror("inet_pton");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Construct DNS query
    construct_dns_query(buffer, &length, hostname);

    // Send DNS query
    if (sendto(sockfd, buffer, length, 0, (struct sockaddr
*)&server_addr, sizeof(server_addr)) < 0) {
        perror("sendto");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Receive DNS response
    struct sockaddr_in from_addr;
    socklen_t from_len = sizeof(from_addr);
    ssize_t recv_len = recvfrom(sockfd, buffer, sizeof(buffer), 0,
(struct sockaddr *)&from_addr, &from_len);

    if (recv_len < 0) {
        perror("recvfrom");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Parse and print the response
    parse_dns_response(buffer, recv_len);

    close(sockfd);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <DNS server IP> <hostname>\n",
argv[0]);
        exit(EXIT_FAILURE);
    }

    dns_query(argv[1], argv[2]);
    return 0;
}
```