3.2 NETWORK FLOW

martedì 5 marzo 2024 08:33

web client

```c
#include <unistd.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
int main() {
  struct sockaddr_in addr; // Structure for addressing internet service.
  int s; // Socket descriptor.
  int t; // Temporary variable for bytes count.
  int total; // Total bytes read from the server.
  int i; // Loop counter.
  char request[5000], response[1000000]; // Request and response buffers.
  unsigned char targetip[4] = { 142, 250, 178, 4 }; // Server IP address.
  // Create a socket for the client.
  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s == -1) {
    perror("Socket failed"); // Print error if socket creation fails.
    printf("errno=%d\n", errno);
    return 1;
  }
  // Set the address structure.
  addr.sin_family = AF_INET; // Internet address family.
  addr.sin_port = htons(80); // Server port, HTTP default.
  addr.sin_addr.s_addr = *(unsigned int*)targetip; // Server IP address.
  // Connect to the server.
  if (-1 == connect(s, (struct sockaddr *)&addr, sizeof(struct
sockaddr_in))) {
    perror("Connect failed"); // Print error if connection fails.
    return 1;
  }
  printf("Socket descriptor: %d\n", s);
  // Prepare the HTTP GET request message.
  sprintf(request, "GET / \r\n");
  // Send the GET request to the server.
  if (-1 == write(s, request, strlen(request))
    perror("Write failed"); // Print error if write operation fails.
    return 1;
  }
  // Read the server's response.
  for (total = 0; (t = read(s, response + total, sizeof(response) -
total)) > 0; total += t) {
  // Check for read error.
  if (t == -1) {
    perror("Read failed"); // Print error if read operation fails.
    return 1;
  }
  // Print the server's response.
  for (i = 0; i < total; i++) {
    printf("%c", response[i]);
  }
  return 0; // End of program.
```

web server

```c
#include <unistd.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
int main() {
  struct sockaddr_in addr, remote_addr; // Server and client address structures.
  int s, s2; // Socket file descriptors for the server and client.
  int len; // Length of the client address.
  int t; // Temporary variable for return values.
  int yes = 1; // Flag for setsockopt.
  char *method, *path, *ver; // Pointers to the method, path, and version in the
request.
  char request[5000], response[10000]; // Buffers for storing the request and
response.
  FILE *fin; // File pointer for the requested file.

  // Create a TCP socket.
  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s == -1) {
    perror("Socket Failed");
    return 1;
  }

  // Initialize the server address structure.
  addr.sin_family = AF_INET; // Address family.
  addr.sin_port = htons(8033); // Port number.
  addr.sin_addr.s_addr = INADDR_ANY; // Listen on all interfaces.
  // Allow the server to reuse the address (helpful for restarting the server
quickly).
  t = setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
  if (t == -1) {
    perror("setsockopt Failed");
    return 1;
  }

  // Bind the socket to the address and port number.
  if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) == -1) {
    perror("bind Failed");
    return 1;
  }

  // Listen for incoming connections.
  if (listen(s, 5) == -1) {
    perror("Listen Failed");
    return 1;
  }

  // Server loop.
  while(1) {
    // Accept a connection.
    len = sizeof(remote_addr);
    s2 = accept(s, (struct sockaddr *)&remote_addr, &len);
    if (s2 == -1) {
      perror("Accept Failed");
      continue; // Continue to the next iteration.
    }

    // Read the request from the client.
    t = read(s2, request, sizeof(request) - 1);
    if (t == -1) {
      perror("Read Failed");
      close(s2);
      continue; // Continue to the next iteration.
    }

    // Properly terminate the request string.
    request[t] = '\0';
    printf("%s", request);

    // Parse the request method, path, and version.
    method = request;
    for(i = 0; request[i] != ' '; i++) {}
    request[i] = '\0';
    path = request + i + 1;
    for(i++; request[i] != ' '; i++) {}
    request[i] = '\0';
    ver = request + i + 1;
    for(i++; request[i] != '\r'; i++) {}
    request[i] = '\0';
    printf("method=%s path=%s ver=%s\n", method, path, ver);

    // Try to open the requested file.
    fin = fopen(path + 1, "rt"); // Skip the leading '/'
    if (fin == NULL) {
      // If the file is not found, send a 404 response.
      sprintf(response, "HTTP/1.1 404 Not Found\r\n\r\n");
      write(s2, response, strlen(response));
    } else {
      // If the file is found, send a 200 response and then the file content.
      sprintf(response, "HTTP/1.1 200 OK\r\n\r\n");
      write(s2, response, strlen(response));
      while ((t = fgetc(fin)) != EOF) {
        write(s2, &t, 1);
      }
      fclose(fin);
    }

    // Close the client connection.
    close(s2);
  }
  return 0; // End of the program (never reached in this case).
```

To delve deeper into how the four service primitives (Request, Indication, Response, Confirm) map onto read and write operations on sockets, especially in the context of an HTTP transaction, it's essential to understand that sockets provide the communication endpoint for sending or receiving data through the network. Using the HTTP request example, let's explore how these operations translate into function calls and blocking function returns in both the client and server contexts.

Client-Side Operations (Browser making an HTTP GET Request)

1. Request (Write operation)
   - Function Call: The client initiates the communication by performing a write operation, typically using a `write()` function in socket programming, to send the HTTP GET request to the server.
   - Mapping to Service Primitive: This write operation maps to the "Request" primitive, where the client application requests a service (in this case, asking for a webpage).

2. Confirm (Read operation):
   - Blocking Function Return: After sending the request, the client will perform a read operation, usually using a `read()` function, which is blocking. The client waits (blocks) until the server sends the HTTP Response back.
   - Mapping to Service Primitive: This read operation maps to the "Confirm" primitive, implicitly confirming that the server received the request and processed it by sending a response.

Server-Side Operations (Web Server processing the HTTP GET Request)

1. Indication (Read operation)
   - Blocking Function Return: The server, upon accepting a connection, waits for incoming requests using a blocking `read()` function. When the HTTP GET request arrives, this function returns, providing the server with the incoming request data.
   - Mapping to Service Primitive: This read operation corresponds to the "Indication" primitive, indicating to the server that there is an incoming service request (the HTTP GET request from the client).

2. Response (Write operation):
   - Function Call After processing the request, the server responds by calling a write operation, such as `send()`, to transmit the HTTP Response (containing the requested webpage) back to the client.
   - Mapping to Service Primitive: This write operation maps to the "Response" primitive, where the server responds to the client's request with the appropriate service outcome (the webpage content).