Click to verify

HTTP Parser and message builder / objects in plain C Snapshot #include #include #include #include #include Go to the source code of this file. struct tagSTRINGPAIR struct tagHTTP_MESSAGE struct tagHTTP_REQUEST struct tagHTTP_RESPONSE struct tagHTTP_PARSER struct tagHTTP_REQUEST_PARSER struct tagHTTP_RESPONSE_PARSER struct tagHTTP_RESPONSE_WRITER #define HTTP_MESSAGE_FLAG_CONNECTION_CLOSE   1 #define HTTP_MESSAGE_FLAG_TRANSFER_CHUNKED   2 #define HTTP_MESSAGE_FLAG_HAS_CONTENT_LENGTH   4 #define HTTP_MESSAGE_FLAG_KEEPALIVE   8 typedef struct tagSTRINGPAIR STRINGPAIR typedef struct tagHTTP_MESSAGE HTTP_MESSAGE typedef int(* HEADER_ACTION)(struct tagHTTP_MESSAGE *request, struct tagHTTP_PARSER *parser) callback that is invoked in order to parse contents of a specific http header typedef enum tagHttp_method_type Http_method_type typedef enum tagHttp_version_type Http_version_type typedef struct tagHTTP_REQUEST HTTP_REQUEST typedef struct tagHTTP_RESPONSE HTTP_RESPONSE typedef int(* HTTP_RESP_HEADER_PARSED)(HTTP_RESPONSE *request, void *ctx) typedef int(* HTTP_RESP_MESSAGE_BODY_DATA)(HTTP_RESPONSE *request, void *data, size_t data_size, void *ctx) typedef int(* HTTP_RESP_FINISHED)(HTTP_RESPONSE *request, void *ctx) typedef enum tagHTTP_STATE_PARSING HTTP_STATE_PARSING typedef struct tagHTTP_PARSER HTTP_PARSER typedef int(* HTTP_PROCESS_MSG_DATA)(HTTP_MESSAGE *msg, void *data, size_t data_size, void *ctx) typedef int(* HTTP_REQ_HEADER_PARSED)(HTTP_REQUEST *request, void *ctx) typedef int(* HTTP_REQ_MESSAGE_BODY_DATA)(HTTP_REQUEST *request, void *data, size_t data_size, void *ctx) typedef int(* HTTP_REQ_FINISHED)(HTTP_REQUEST *request, void *ctx) typedef struct tagHTTP_REQUEST_PARSER HTTP_REQUEST_PARSER typedef struct tagHTTP_RESPONSE_PARSER HTTP_RESPONSE_PARSER typedef struct tagHTTP_RESPONSE_WRITER HTTP_RESPONSE_WRITER enum tagHttp_method_type {   HTTP_METHOD_GET, HTTP_METHOD_POST, HTTP_METHOD_PUT, HTTP_METHOD_HEAD,   HTTP_METHOD_OPTIONS, HTTP_METHOD_DELETE, HTTP_METHOD_TRACE, HTTP_METHOD_CONNECT } enum tagHttp_version_type {   HTTP_VERSION_1_0, HTTP_VERSION_1_1 } enum tagHTTP_STATE_PARSING {   HTTP_STATE_PARSING_REQUEST_LINE, HTTP_STATE_PARSING_HEADERS, HTTP_STATE_PARSING_BODY_CONTENT_LENGTH, HTTP_STATE_PARSING_BODY_CHUNK_HEADER,   HTTP_STATE_PARSING_BODY_CHUNK_DATA,   HTTP_STATE_PARSING_BODY_CHUNK_EOF_AFTER_DATA, HTTP_STATE_PARSING_BODY_CHUNK_TRAILER } enum PARSER_STATUS { PARSER_STATUS_ERROR = -1, PARSER_STATUS_COMPLETED = 0, PARSER_STATUS_NEED_MORE_DATA = 1 } enum HTTP_TK_TYPE { HTTP_TK_QUOTED_STRING, HTTP_TK_TEXT,   HTTP_TK_SEPARATOR, HTTP_TK_EOF } enum HTTP_RESPONSE_WR_STATE {   HTTP_RESPONSE_WR_STATUS_LINE, HTTP_RESPONSE_WR_CONNECTION_CLOSE, HTTP_RESPONSE_WR_CHUNKED, HTTP_RESPONSE_WR_CONTENT_LENGTH,   HTTP_RESPONSE_WR_HEADERS, HTTP_RESPONSE_WR_EOF } int HTTP_MESSAGE_init (HTTP_MESSAGE *message) void HTTP_MESSAGE_free (HTTP_MESSAGE *message) int HTTP_MESSAGE_add_header (HTTP_MESSAGE *message, const char *name, const char *value) M_INLINE void HTTP_MESSAGE_set_content_length (HTTP_MESSAGE *message, int content_length) const char * HTTP_MESSAGE_find_header (HTTP_MESSAGE *message, const char *name) STRINGPAIR * HTTP_MESSAGE_first_header (HTTP_MESSAGE *message, DLISTUNR_position *pos) STRINGPAIR * HTTP_MESSAGE_next_header (HTTP_MESSAGE *message, DLISTUNR_position *pos) int HTTP_REQUEST_is_persistent (HTTP_REQUEST *message) M_INLINE int HTTP_REQUEST_init (HTTP_REQUEST *message) M_INLINE void HTTP_REQUEST_free (HTTP_REQUEST *message) M_INLINE int HTTP_RESPONSE_init (HTTP_RESPONSE *message, Http_version_type version, int status_code) M_INLINE void HTTP_RESPONSE_free (HTTP_RESPONSE *message) int HTTP_PARSER_init (HTTP_PARSER *parser) int HTTP_PARSER_free (HTTP_PARSER *parser) int HTTP_add_header_parser (HTTP_PARSER *parser, const char *header_name, HEADER_ACTION action) add a parser for a specific http header type PARSER_STATUS HTTP_get_line (BF *bf, char **start_line) returns next raw line from http header, adjoins line continuations. int HTTP_get_header_token (HTTP_PARSER *parser) return next token while parsing http header value. PARSER_STATUS HTTP_parse_header_line (HTTP_PARSER *parser, HTTP_MESSAGE *request, BF *bf, int *eof_header) dispatch parsing of one http header int HTTP_PARSER_content_length_init (HTTP_PARSER *parser, HTTP_MESSAGE *msg) start parsing content length message body PARSER_STATUS HTTP_PARSER_content_length_process (HTTP_PARSER *parser, BF *bf, HTTP_PROCESS_MSG_DATA cb, HTTP_MESSAGE *msg, void *ctx) consume content length message body. int HTTP_PARSER_chunked_data_init (HTTP_PARSER *parser) start parsing chunked data PARSER_STATUS HTTP_PARSER_chunked_data_process (HTTP_PARSER *parser, BF *bf, HTTP_PROCESS_MSG_DATA cb, HTTP_MESSAGE *msg, void *ctx) finish parsind chunked content data. int HTTP_REQUEST_PARSER_init (HTTP_REQUEST_PARSER *parser, HTTP_REQ_HEADER_PARSED header_parsed, HTTP_REQ_MESSAGE_BODY_DATA on_message_body_data, HTTP_REQ_FINISHED on_request_finished, void *ctx) initialise request parser object PARSER_STATUS HTTP_REQUEST_PARSER_process (HTTP_REQUEST_PARSER *parser, HTTP_REQUEST *request, BF *data) parse a http request int HTTP_RESPONSE_PARSER_init (HTTP_RESPONSE_PARSER *parser, HTTP_RESP_HEADER_PARSED header_parsed, HTTP_RESP_MESSAGE_BODY_DATA on_message_body_data, HTTP_RESP_FINISHED on_request_finished, void *ctx) initialise request parser object PARSER_STATUS HTTP_RESPONSE_PARSER_process (HTTP_RESPONSE_PARSER *parser, HTTP_RESPONSE *response, BF *data) parse a http response M_INLINE void HTTP_RESPONSE_WRITER_init (HTTP_RESPONSE_WRITER *writer, HTTP_RESPONSE *response) PARSER_STATUS HTTP_RESPONSE_WRITER_write (HTTP_RESPONSE_WRITER *writer, BF *data) Define Documentation #define HTTP_MESSAGE_FLAG_CONNECTION_CLOSE   1 Definition at line 13 of file http.h. #define HTTP_MESSAGE_FLAG_HAS_CONTENT_LENGTH   4 Definition at line 15 of file http.h. #define HTTP_MESSAGE_FLAG_KEEPALIVE   8 Definition at line 16 of file http.h. #define HTTP_MESSAGE_FLAG_TRANSFER_CHUNKED   2 Definition at line 14 of file http.h. Here’s the translation of the HTTP server example from Go to C, formatted in Markdown suitable for Hugo:#include #include #include #include #include #define PORT 8090 #define BUFFER_SIZE 1024 void handle_hello(int client_socket) { char *response = "HTTP/1.1 200 OK\rContent-Type: text/plain\r\rhello"; send(client_socket, response, strlen(response), 0); } void handle_headers(int client_socket, char *request) { char response[BUFFER_SIZE]; char *header_start = strstr(request, "\r") + 2; char *header_end = strstr(header_start, "\r\r"); snprintf(response, sizeof(response), "HTTP/1.1 200 OK\rContent-Type: text/plain\r\r"); send(client_socket, response, strlen(response), 0); char *header = strtok(header_start, "\r"); while (header != NULL && header < header_end) { snprintf(response, sizeof(response), "%s", header); send(client_socket, response, strlen(response), 0); header = strtok(NULL, "\r"); } } void handle_request(int client_socket) { char buffer[BUFFER_SIZE]; recv(client_socket, buffer, sizeof(buffer), 0); if (strncmp(buffer, "GET /hello", 10) == 0) { handle_hello(client_socket); } else if (strncmp(buffer, "GET /headers", 12) == 0) { handle_headers(client_socket, buffer); } else { char *response = "HTTP/1.1 404 Not Found\rContent-Type: text/plain\r\r404 Not Found"; send(client_socket, response, strlen(response), 0); } close(client_socket); } int main() { int server_socket, client_socket; struct sockaddr_in server_addr, client_addr; socklen_t addr_len = sizeof(client_addr); server_socket = socket(AF_INET, SOCK_STREAM, 0); if (server_socket == -1) { perror("Socket creation failed"); exit(EXIT_FAILURE); } server_addr.sin_family = AF_INET; server_addr.sin_addr.s_addr = INADDR_ANY; server_addr.sin_port = htons(PORT); if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) { perror("Bind failed"); exit(EXIT_FAILURE); } if (listen(server_socket, 10) < 0) { perror("Listen failed"); exit(EXIT_FAILURE); } printf("Server listening on port %d", PORT); while (1) { client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &addr_len); if (client_socket < 0) { perror("Accept failed"); continue; } handle_request(client_socket); } close(server_socket); return 0; }This C program implements a basic HTTP server similar to the original example. Let’s break down the key components:We include necessary headers for socket programming and standard I/O operations.The handle_hello function sends a simple “hello” response to the client.The handle_headers function reads the HTTP request headers and echoes them back to the client.The handle_request function processes incoming requests, routing them to the appropriate handler based on the URL path.In the main function, we set up a socket, bind it to a port, and enter a loop to accept and handle incoming connections.To compile and run the server:$ gcc http_server.c -o http_server $ ./http_server Server listening on port 8090You can then access the server using curl or a web browser:$ curl localhost:8090/hello hello $ curl localhost:8090/headers User-Agent: curl/7.68.0 Accept: */*This C implementation provides similar functionality to the original example, demonstrating how to create a basic HTTP server using socket programming in C. You can’t perform that action at this time. To send an HTTP request in C, you’ll generally lean on libraries like libcurl, as C does not have built-in support for web protocols. Here’s a simple example using libcurl to perform a GET request:First, ensure you have libcurl installed on your system. Then, include the necessary headers and link against the libcurl library in your source file:#include #include int main(void) { CURL *curl; CURLcode res; curl = curl_easy_init(); // Initialize a libcurl handle if(curl) { // Set the URL that receives the libcurl handle curl_easy_setopt(curl, CURLOPT_URL, " "); // Define a callback to get the data curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, NULL); // Perform the request, res will get the return code res = curl_easy_perform(curl); // Check for errors if(res != CURLE_OK) fprintf(stderr, "curl_easy_perform() failed: %s", curl_easy_strerror(res)); // Always cleanup curl_easy_cleanup(curl); } return 0; }Compile this with something akin to gcc -o http_request http_request.c -lcurl, running it should perform a simple GET request to “ ”.Sample Output Since the example doesn’t process the server’s response, running it won’t produce a visible output beyond potential error messages. Integrating the callback function for processing received data is essential for meaningful interaction.Deep Dive The concept of sending HTTP requests from a C program hinges on the language’s powerful networking capabilities, coupled with external libraries since C itself is a low-level language without built-in high-level internet protocol support. Historically, programmers would manually use socket programming in C, a complex and tedious process, to interact with web servers before the advent of dedicated libraries like libcurl.Libcurl, built on top of C, streamlines the process, abstracting away the gritty details of socket programming and HTTP protocol specifics. It supports a multitude of protocols beyond HTTP/HTTPS, including FTP, SMTP, and more, making it a versatile tool for network programming in C.While using libcurl for HTTP requests in C is practical, modern programming often gravitates towards languages with built-in support for such tasks, like Python (requests library) or JavaScript (Fetch API). These alternatives offer simpler, more readable syntax at the expense of the granular control and performance optimizations possible in C through direct socket manipulation and finely-tuned library use.For critical performance applications or where direct system-level interaction is necessary, C remains a viable option, particularly with libcurl smoothing over the complexities of web communication. However, for most high-level web interactions, exploring more dedicated web programming languages might prove more efficient.Last updated on March 13, 2024