```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include "media_transfer.h"
#include "parser.h"

/*
 * This functions checks if a request contains
 * "list" or "get" as the first few bytes. Function,
 * then returns a command type based on request.
 */
command_t get_command_from_request(const char *request) {
    if(request == NULL) {
        return INVALID;
    }
    else if(request[0] == '#') {
        return COMMENT;
    }
    else if(strncmp(request, "list", 4) == 0) {
        return LIST;
    }
    else if(strncmp(request, "get", 3) == 0) {
        int len = strlen(request);
        if(len <= 4) { // no file name specified
            printf("No file name specified for get command\n");
            return INVALID;
        }
        return GET;
    }
    else if(strncmp(request, "exit", 4) == 0) {
        return EXIT;
    }
    else {
        return INVALID;
    }
}

/*
 * A contructor function for header struct
 * @returns an empty header struct
 */
header create_header() {
    header h;
    h.status = 0;
    h.length = 0;
    h.type = 0;
    h.host = 0;

    return h;
}

/*
 * @param string - buffer to find occurence of chracter from
 * @param c      - value of char whose occurence to be found
 * @param n      - number of occurences to be found
 * @returns      - position index of the nth occurence.
 */
int get_occurrence_n(char * string, char c, int n) {
    if (string != NULL) {
        int occ = 0;
        int i;
        for (i = 0; i < strlen(string); i++) {
            if (string[i] == c) {
                if ((++occ) == n) return i;
            }
        }
    }

    return -1;
}

void get_time_spec_to_string(char *buf, size_t buflen) {
    struct timespec ts;
    timespec_get(&ts, 1); //TIME_UTC = 1
    char temp[buflen];
    strftime(temp, buflen, "%D %T", gmtime(&ts.tv_sec));
    sprintf(buf, "%s.%09ld UTC", temp, ts.tv_nsec);
}

/*
 * @param str - string to find the number of lines it contains
 * @returns   - number of lines in a string
 */
int count_lines(char const *str)
{
    char const *p = str;
    int count;
    for (count = 0; ; ++count) {
        p = strstr(p, "\r\n");
        if (!p)
            break;
        p = p + 2;
    }
    return count - 1;
}

/*
 * @param header - buffer containing header information
 * @param line_number - spefic line of header buffer to return
 * @returns
 *      a particular line from header buffer
 */
char * get_line(char * header_text, unsigned int line_number) {
    char * ret = 0;
    int line_count = 1;
    int start = -2;
    int cur = 0;
    int i;
    for (i = 0; i < line_number; ++i) {
        start = cur;
        cur = start + 2;
        while (header_text[cur] && header_text[cur] != '\r') {
            if (header_text[cur + 1] && header_text[cur + 1] == '\n') break;
            cur++;
        }

        if (header_text[cur + 2] && header_text[cur + 2] == '\r') {
            if (header_text[cur + 3] && header_text[cur + 3] == '\n') {
                break;
            }
        }

        line_count++;
    }
    if (line_number > line_count) return NULL;

    if (line_number == 1) {
        ret = calloc(cur + 1, sizeof(char));
        strncpy(ret, header_text, cur);
    }
    else {
        ret = calloc(cur - start - 1, sizeof(char));
        strncpy(ret, header_text + start + 2, cur - start - 2);
    }

    return ret;
```

```c
}

/*
 * @param socket - socket id to receive header text from
 * @returns      - prints and then returns a buffer containing header text
 */
char * read_header_text(int socket) {
    char buffer[BUFLEN] = {0};
    int buf_ind = 0;
    int ret_size = 0;
    int cont = 1;
    char *header_text = NULL;
    while (cont) {
        while (buf_ind < BUFLEN && 1 == read(socket, &buffer[buf_ind], 1)) {
            if (buf_ind > 2                    &&
                '\n' == buffer[buf_ind]      &&
                '\r' == buffer[buf_ind - 1] &&
                '\n' == buffer[buf_ind - 2] &&
                '\r' == buffer[buf_ind - 3])
            {
                cont = 0;
                break;
            }
            buf_ind++;
        }

        buf_ind++;


        if (header_text == NULL) {
            header_text = (char*)malloc(buf_ind * sizeof(char) + 1);
            memset(header_text, 0, buf_ind + 1);
            strncpy(header_text, buffer, buf_ind);

            ret_size = buf_ind + 1;
        } else {
            header_text = (char*) realloc(header_text, (ret_size += buf_ind));
            memset(header_text + ret_size - 1, 0, 1);
            strncat(header_text, buffer, buf_ind);
        }

        memset(buffer, 0, BUFLEN);
        buf_ind = 0;
    }

    //printf("%s\n", header_text);
    return header_text;
}

/*
 * @param header_text - buffer to read from
 * @param h           - storage location to store information
 * @returns           - success or failure
 */
int buffer_to_header(char * header_text, header *h) {

    if(!header_text) {
        return -1;
    }

    char * line = NULL;
    int current = 1;
    int additional_count = 0;
    while ((line = get_line(header_text, current)) != NULL) {
        int token_loc = get_occurrence_n(line, ':', 1);
        if (token_loc > 0) {
            char key[token_loc + 1];
            char value[strlen(line) - token_loc];

```

```c
                memset(key, 0, sizeof(key));
                memset(value, 0, sizeof(value));
                int i;
                for (i = 0; i < sizeof(key) - 1; i++) key[i] = line[i];
                for (i = 0; i < sizeof(value) - 1; i++) value[i] = line[token_loc + i + 2];

                if (strcmp(key, "Status") == 0) h->status = atoi(value);
                else if (strcmp(key, "Host") == 0) {
                    h->host = malloc(sizeof(value));
                    strcpy(h->host, value);
                } else if (strcmp(key, "Type") == 0) {
                    h->type = malloc(sizeof(value));
                    strcpy(h->type, value);
                } else if (strcmp(key, "Length") == 0) h->length = atoi(value);
            }

            free(line);
            line = NULL;
            if (++current > count_lines(header_text)) break;
        }
        return 1;
    }

    /* Handle command from a string value
     * @param socker        - socket to use for server communication
     * @param command       - command string read from usr or file
     * @param len           - len of incoming command
     * @returns             - success or failure
     */
    int handle_command(int socket, char *command, int len) {
        switch (get_command_from_request(command)) {
            case GET:
                process_command(socket, command, BUFLEN);
                break;
            case LIST:
                process_command(socket, command, BUFLEN);
                break;
            case EXIT:
                printf("Good bye\n");
                return 1;
                break;
            case INVALID:
                printf("Invalid Command: %s\n", command);
            default:
                break;
        }
        return 1;
    }

    /*
     * Handle get request from client
     * @param server_socket - socket to communicate to server
     * @returns - success or failure
     */
    int process_command(int server_socket, char *command, int len) {

        /* send out user command */
        write(server_socket, command, len);

        // read header response
        char *header_text = read_header_text(server_socket);
        char time_stamp[TIME_BUFFER_LEN];
        get_time_spec_to_string(time_stamp, BUFLEN);
        printf("%s: Header Response Received\n", time_stamp);
        if(!header_text) {
            perror("fatal error\n");
        }

        // store buffer information to header struc
        header h = create_header();
        buffer_to_header(header_text, &h);

        free(header_text);
        header_text = NULL;

        get_time_spec_to_string(time_stamp, TIME_BUFFER_LEN);
        printf("%s: Status:%d  Host:%s  Length:%ld  Type:%s  \n", time_stamp,h.status, h.host, h.length, h.type);

        switch (h.status) {
            case 100:
                if (strcmp(h.type, "Text") == 0) {
                    char list[h.length + 1];
                    list[h.length];
                    memset(list, 0, h.length + 1);

                    size_t received = 0;

                    while (received < h.length) {
                        if (read(server_socket, list + received, 1)) ++received;
                    }

                    printf("%s\n", list);
                    get_time_spec_to_string(time_stamp, TIME_BUFFER_LEN);
                    printf("%s: File Listing Received\n", time_stamp);
                }
                else {
                    command[strcspn(command, "\n")] = 0;
                    // get output name of the file from user
                    char output_name[BUFLEN];
                    printf("%s: Name of the file to put data received from server to: ", time_stamp);
                    fgets(output_name, BUFLEN, stdin);
                    output_name[strcspn(output_name, "\n")] = 0;

                    // store to the output file
                    receive_media(server_socket, output_name, h.length);
                    get_time_spec_to_string(time_stamp, TIME_BUFFER_LEN);
                    printf("%s: Media Received and Downloaded\n", time_stamp);
                }
                break;
            case 301:
                fprintf(stderr, "Unknown command!\n");
                break;
            case 404:
                fprintf(stderr, "File not found!\n");
                break;
            default:
                fprintf(stderr, "Undefined error!\n");
                break;
        }
    }


    /*
     * Runs commands from batch script
     * @param clientrc_path - path to read client commands from
     */
    int process_batch(int socket, char * clienrc_path) {
        if(!clienrc_path) {
            perror("Could not find script path\n");
            return - 1;
        }

        FILE* fp = fopen(clienrc_path, "r");
        if(!fp) {
            perror("Could not find script path\n");
            return -1;
        }
        char buffer[BUFLEN];
        while(fgets(buffer, BUFLEN, fp)){
            switch(get_command_from_request(buffer)) {
                case GET:
                    handle_command(socket, buffer, BUFLEN);         /* send it out */
                    break;
                case LIST:
                    handle_command(socket, buffer, BUFLEN);         /* send it out */
                    break;
                case EXIT:
                    return 1;
                default:
                    break;
            }
        }
    }
```