

sep 24, 18 22:35

stack.h

Page 1/1

```

1  #ifndef __STACK_H__
2  #define __STACK_H__
3
4  /** Stack **/
5
6  #include "node.h"
7
8  typedef struct {
9      node_t* head;
10 } stack_t;
11
12 // Crea un stack.
13 // Pre: self apunta a un sector valido de memoria.
14 void stack_create(stack_t *self);
15 // Agrega el elemento pasado por parametro en el stack.
16 // Pre: self apunta un sector valido de memoria.
17 void stack_push(stack_t *self, int data);
18 // Devuelve el elemento que esta en le tope de la pila pero sin sacarlo.
19 // Pre: self apunta un sector valido de memoria.
20 int stack_top(stack_t *self);
21 // Devuelve el elemento que esta en le tope de la pila y lo saca.
22 // Pre: self apunta un sector valido de memoria.
23 int stack_pop(stack_t *self);
24 // Destruye la instancia self liberando sus recursos.
25 // Pre: self apunta un sector valido de memoria
26 // self fue inicializado mediante stack_create
27 void stack_destroy(stack_t *self);
28
29 #endif

```

sep 24, 18 22:35

stack.c

Page 1/1

```

1  #include "stack.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "node.h"
5
6  void stack_create(stack_t *self) {
7      self->head = NULL;
8  }
9
10 void stack_push(stack_t *self, int data) {
11     node_t *node = malloc(sizeof(node_t));
12     node_create(node);
13     node_set_data(node, data);
14
15     if(self->head != NULL) {
16         node_set_next(node, self->head);
17     }
18     self->head = node;
19 }
20
21 int stack_pop(stack_t *self) {
22     int data = node_get_data(self->head);
23     node_t* to_delete = self->head;
24     self->head = node_get_next(self->head);
25     free(to_delete);
26     return data;
27 }
28
29 int stack_top(stack_t *self) {
30     return node_get_data(self->head);
31 }
32
33 void stack_destroy(stack_t *self) {
34     if(self->head != NULL) {
35         stack_pop(self);
36         stack_destroy(self);
37     }
38 }

```

sep 24, 18 22:35

socket.h

Page 1/2

```

1  #ifndef __SOCKET_H__
2  #define __SOCKET_H__
3
4  /** Socket **/
5
6  #include <stddef.h>
7  #include <string.h>
8  #include <sys/socket.h>
9  #include <sys/types.h>
10 #include <netinet/in.h>
11
12 typedef struct {
13     int skt_fd; // socket file descriptor
14 } socket_t;
15
16 // Crea un socket.
17 // Pre: self apunta a un sector valido de memoria.
18 int socket_create(socket_t *self);
19 // Deshabilita la lectura de datos del socket.
20 // Pre: self apunta un sector valido de memoria.
21 int socket_shutdown_read(socket_t *self);
22 // Deshabilita la escritura de datos del socket.
23 // Pre: self apunta un sector valido de memoria.
24 int socket_shutdown_write(socket_t *self);
25 // Deshabilita la lectura y escritura de datos del socket.
26 // Pre: self apunta un sector valido de memoria.
27 int socket_shutdown(socket_t *self);
28 // Destruye la instancia self liberando sus recursos.
29 // Pre: self fue inicializado mediante socket_create o socket_accept.
30 int socket_destroy(socket_t *self);
31 // Deshabilita la lectura y escritura de datos del socket.
32 // Destruye la instancia self liberando sus recursos.
33 // Pre: self fue inicializado mediante socket_create o socket_accept.
34 int socket_shutdown_destroy(socket_t *self);
35 // Establece a que IP y puerto se quiere asociar el Socket.
36 // Pre: self apunta un sector valido de memoria.
37 int socket_bind(socket_t *self, unsigned short port);
38 /* Define cuantas conexiones en espera pueden esperar hasta ser
39  * aceptadas en el Socket.
40  */
41 // Pre: self apunta un sector valido de memoria
42 int socket_listen(socket_t *self, int max_standby_conn);
43 /* Establece una connexion a la maquina remota, indicada
44  * mediante host_name y port.
45  */
46 // Pre: self apunta un sector valido de memoria.
47 int socket_connect(socket_t *self, const char *host_name, unsigned short port);
48 // Crea un nuevo socket servidor para la comunicacion con el socket cliente.
49 // Pre: self apunta un sector valido de memoria.
50 // Post: Retorna el file descriptor del socket aceptado (socket servidor).
51 int socket_accept(socket_t *self, socket_t *socket_accepted);
52 // Envia el contenido de buffer en forma de bytes.
53 // Pre: self apunta un sector valido de memoria.
54 /* Post: Valores de retorno:
55  * s < 0 : Hubo un error inesperado.
56  * s == 0 : El socket fue cerrado.
57  * s > 0 : s bytes fueron enviados.
58  */
59 int socket_send(socket_t *self, const void *buffer, size_t length);
60 // Envia el int con signo "i" a traves del socket.
61 // Si detecta un error aplica un shutdown y destroy del socket.
62 // Post: Retorna 0 en caso de funcionar bien.
63 // Retorna -1 en caso de error.
64 int socket_send_int(socket_t *self, int i);
65 // Envia el char con signo "c" a traves del socket.
66 // Si detecta un error aplica un shutdown y destroy del socket.

```

sep 24, 18 22:35

socket.h

Page 2/2

```

67 // Post: Retorna 0 en caso de funcionar bien.
68 // Retorna -1 en caso de error.
69 int socket_send_char(socket_t *socket, char c);
70 // Recibe el contenido de buffer en forma de bytes.
71 // Pre: self apunta un sector valido de memoria.
72 /* Post: Valores de retorno:
73  * s < 0 : Hubo un error inesperado.
74  * s == 0 : El socket fue cerrado.
75  * s > 0 : s bytes fueron recibidos.
76  */
77 int socket_receive(socket_t *self, void *buffer, size_t length);
78 // Recibe el int con signo "i" a traves del socket.
79 // Si detecta un error aplica un shutdown y destroy del socket.
80 // Post: Retorna 0 en caso de funcionar bien.
81 // Retorna -1 en caso de error.
82 int socket_recv_int(socket_t *socket, int *i);
83 // Recibe el char con signo "c" a traves del socket.
84 // Si detecta un error aplica un shutdown y destroy del socket.
85 // Post: Retorna 0 en caso de funcionar bien.
86 // Retorna -1 en caso de error.
87 int socket_recv_char(socket_t *socket, char *c);
88 // Recibe el char sin signo "c" a traves del socket.
89 // Si detecta un error aplica un shutdown y destroy del socket.
90 // Post: Retorna 0 en caso de funcionar bien.
91 // Retorna -1 en caso de error.
92 int socket_recv_unsigned_char(socket_t *socket, unsigned char *c);
93
94 #endif

```

sep 24, 18 22:35

socket.c

Page 1/4

```

1  #define _POSIX_C_SOURCE 200112L
2
3  #include <string.h>
4  #include "socket.h"
5  #include <stddef.h>
6  #include <sys/socket.h>
7  #include <sys/types.h>
8  #include <unistd.h>
9  #include <netinet/in.h>
10 #include <arpa/inet.h>
11 #include <netdb.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <errno.h>
15 #include <stdbool.h>
16
17 #define SUCCESS 0
18 #define ERROR -1
19
20 #define SIZE_OF_INT 4 // bytes
21 #define SIZE_OF_CHAR 1 // bytes
22
23 int socket_create(socket_t *self) {
24     self->skt_fd = socket(AF_INET, SOCK_STREAM, 0);
25     return self->skt_fd;
26 }
27
28 int socket_bind(socket_t *self, unsigned short port) {
29     struct sockaddr_in skt_addr;
30     skt_addr.sin_family = AF_INET;
31     skt_addr.sin_port = htons(port);
32     skt_addr.sin_addr.s_addr = INADDR_ANY;
33
34     return bind(self->skt_fd, (struct sockaddr *) &skt_addr, sizeof(skt_addr));
35 }
36
37 int socket_connect(socket_t *self, const char* hostname, unsigned short port) {
38     bool are_we_connected = false;
39     int s = 0;
40     struct addrinfo hints;
41     struct addrinfo *result, *ptr;
42
43     char portString[6];
44     snprintf(portString, sizeof(portString), "%d", port);
45
46     memset(&hints, 0, sizeof(struct addrinfo));
47     hints.ai_family = AF_INET; /* IPv4 (or AF_INET6 for IPv6) */
48     hints.ai_socktype = SOCK_STREAM; /* TCP (or SOCK_DGRAM for UDP) */
49     hints.ai_flags = 0; /* None (or AI_PASSIVE for server) */
50
51     s = getaddrinfo(hostname, portString, &hints, &result);
52
53     for (
54         ptr = result;
55         ptr != NULL ^ are_we_connected == false;
56         ptr = ptr->ai_next
57     ) {
58         if (self->skt_fd != ERROR) {
59             s = connect(self->skt_fd, ptr->ai_addr, ptr->ai_addrlen);
60             if (s == ERROR) close(self->skt_fd);
61             are_we_connected = (s != -1);
62         }
63     }
64
65     freeaddrinfo(result);
66

```

sep 24, 18 22:35

socket.c

Page 2/4

```

67     if (are_we_connected == false) return ERROR;
68
69     return SUCCESS;
70 }
71
72 int socket_listen(socket_t *self, int max_standby_conn) {
73     return listen(self->skt_fd, max_standby_conn);
74 }
75
76 int socket_accept(socket_t *self, socket_t *socket_accepted) {
77     socket_accepted->skt_fd = accept(self->skt_fd, (struct sockaddr*) NULL, NULL);
78     return socket_accepted->skt_fd;
79 }
80
81 int socket_send(socket_t *self, const void *buffer, size_t lenght) {
82     return send(self->skt_fd, buffer, lenght, MSG_NOSIGNAL);
83 }
84
85 int socket_send_int(socket_t *socket, int i) {
86     int i_big_endian = htonl(i);
87     int bytes_sent = 0;
88     int skt_still_open = 1;
89     while (SIZE_OF_INT > bytes_sent ^ skt_still_open) {
90         int s = socket_send(socket, &i_big_endian, SIZE_OF_INT - bytes_sent);
91         if (s == -1) {
92             socket_shutdown(socket);
93             socket_destroy(socket);
94             return ERROR;
95         } else if (s == 0) {
96             skt_still_open = 0;
97             return ERROR;
98         } else {
99             bytes_sent += s;
100         }
101     }
102
103     return SUCCESS;
104 }
105
106 int socket_send_char(socket_t *socket, char i) {
107     int bytes_sent = 0;
108     int skt_still_open = 1;
109     while (SIZE_OF_CHAR > bytes_sent ^ skt_still_open) {
110         int s = socket_send(socket, &i, SIZE_OF_CHAR - bytes_sent);
111         if (s == -1) {
112             socket_shutdown(socket);
113             socket_destroy(socket);
114             return ERROR;
115         } else if (s == 0) {
116             skt_still_open = 0;
117             return ERROR;
118         } else {
119             bytes_sent += s;
120         }
121     }
122
123     return SUCCESS;
124 }
125
126 int socket_receive(socket_t *self, void *buffer, size_t lenght) {
127     return read(self->skt_fd, buffer, lenght);
128 }
129
130 int socket_rcv_int(socket_t *socket, int *i) {
131     int bytes_rcv = 0;
132     int skt_still_open = 1;
133     while (SIZE_OF_INT > bytes_rcv ^ skt_still_open) {

```

sep 24, 18 22:35

socket.c

Page 3/4

```

133     int s = socket_receive(socket, i, SIZE_OF_INT - bytes_recv);
134     if (s == -1) {
135         socket_shutdown(socket);
136         socket_destroy(socket);
137         return ERROR;
138     } else if (s == 0) {
139         skt_still_open = 0;
140         return ERROR;
141     } else {
142         bytes_recv += s;
143     }
144 }
145
146 *i = ntohl(*i);
147
148 return SUCCESS;
149 }
150
151 int socket_recv_char(socket_t *socket, char *c) {
152     int bytes_recv = 0;
153     int skt_still_open = 1;
154     while (SIZE_OF_CHAR > bytes_recv ^ skt_still_open) {
155         int s = socket_receive(socket, c, SIZE_OF_CHAR - bytes_recv);
156         if (s == -1) {
157             socket_shutdown(socket);
158             socket_destroy(socket);
159             return ERROR;
160         } else if (s == 0) {
161             skt_still_open = 0;
162             return ERROR;
163         } else {
164             bytes_recv += s;
165         }
166     }
167     return SUCCESS;
168 }
169
170 int socket_recv_unsigned_char(
171     socket_t *socket,
172     unsigned char *c
173 ) {
174     int bytes_recv = 0;
175     int skt_still_open = 1;
176     while (SIZE_OF_CHAR > bytes_recv ^ skt_still_open) {
177         int s = socket_receive(socket, c, SIZE_OF_CHAR - bytes_recv);
178         if (s == -1) {
179             socket_shutdown(socket);
180             socket_destroy(socket);
181             return ERROR;
182         } else if (s == 0) {
183             skt_still_open = 0;
184             return ERROR;
185         } else {
186             bytes_recv += s;
187         }
188     }
189     return SUCCESS;
190 }
191
192 int socket_shutdown_read(socket_t *self) {
193     return shutdown(self->skt_fd, SHUT_RD);
194 }
195
196 int socket_shutdown_write(socket_t *self) {
197     return shutdown(self->skt_fd, SHUT_WR);
198 }

```

sep 24, 18 22:35

socket.c

Page 4/4

```

199
200 int socket_shutdown(socket_t *self) {
201     return shutdown(self->skt_fd, SHUT_RDWR);
202 }
203
204 int socket_destroy(socket_t *self) {
205     return close(self->skt_fd);
206 }
207
208 int socket_shutdown_destroy(socket_t *self) {
209     if (
210         (shutdown(self->skt_fd, SHUT_RDWR) == ERROR) ^
211         (close(self->skt_fd) == ERROR)
212     ) {
213         return ERROR;
214     } else {
215         return SUCCESS;
216     }
217 }

```

sep 24, 18 22:35

server.h

Page 1/1

```

1 #ifndef __SERVER_H__
2 #define __SERVER_H__
3
4 // Funcion encargada de realizar las tareas del servidor.
5 int server(char *argv[]);
6
7 #endif

```

sep 24, 18 22:35

server.c

Page 1/2

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "socket.h"
4 #include "java_virtual_machine.h"
5 #include "bytecodes.h"
6
7 #define MAX_STANDBY_CONNS 10
8
9 #define ERROR -1
10 #define SUCCESS 0
11
12 int server(char *argv[]) {
13     // argv[2] = Port
14
15     if (argv[2] == NULL) return ERROR;
16
17     short unsigned int port = (short unsigned int) strtoul(argv[2], NULL, 10);
18
19     socket_t skt_acceptor;
20     socket_t skt_server;
21
22     if (socket_create(&skt_acceptor) == ERROR) return ERROR;
23     if (socket_bind(&skt_acceptor, port) == ERROR) return ERROR;
24     if (socket_listen(&skt_acceptor, MAX_STANDBY_CONNS) == ERROR) return ERROR;
25     if (socket_accept(&skt_acceptor, &skt_server) == ERROR) return ERROR;
26
27     int N;
28
29     if (socket_recv_int(&skt_server, &N) == ERROR) {
30         socket_shutdown_destroy(&skt_acceptor);
31         return ERROR;
32     }
33
34     unsigned char bytecode;
35     char c;
36
37     jvm_t java_virtual_machine;
38     jvm_create(&java_virtual_machine, N);
39
40     printf("%s\n", "Bytecode trace");
41
42     while(socket_recv_unsigned_char(&skt_server, &bytecode) == SUCCESS) {
43         switch(bytecode) {
44             case BIPUSH: printf("bipush\n");
45                 if (socket_recv_char(&skt_server, &c) == ERROR) {
46                     socket_shutdown_destroy(&skt_acceptor);
47                     jvm_destroy(&java_virtual_machine);
48                     return ERROR;
49                 }
50                 jvm_execute_instruction_bipush(&java_virtual_machine, c); break;
51             case DUP: printf("dup\n");
52                 jvm_execute_instruction_dup(&java_virtual_machine); break;
53             case ISTORE: printf("istore\n");
54                 if (socket_recv_char(&skt_server, &c) == ERROR) {
55                     socket_shutdown_destroy(&skt_acceptor);
56                     jvm_destroy(&java_virtual_machine);
57                     return ERROR;
58                 }
59                 jvm_execute_instruction_istore(&java_virtual_machine, c); break;
60             case ILOAD: printf("iload\n");
61                 if (socket_recv_char(&skt_server, &c) == ERROR) {
62                     socket_shutdown_destroy(&skt_acceptor);
63                     jvm_destroy(&java_virtual_machine);
64                     return ERROR;
65                 }
66                 jvm_execute_instruction_ildload(&java_virtual_machine, c); break;

```

sep 24, 18 22:35

server.c

Page 2/2

```

67     case IADD: printf("iadd\n");
68         jvm_execute_instruction_iadd(&java_virtual_machine);      break;
69     case ISUB: printf("isub\n");
70         jvm_execute_instruction_isub(&java_virtual_machine);      break;
71     case IMUL: printf("imul\n");
72         jvm_execute_instruction_imul(&java_virtual_machine);      break;
73     case IDIV: printf("idiv\n");
74         jvm_execute_instruction_idiv(&java_virtual_machine);      break;
75     case IREM: printf("irem\n");
76         jvm_execute_instruction_irem(&java_virtual_machine);      break;
77     case INEG: printf("ineg\n");
78         jvm_execute_instruction_ineg(&java_virtual_machine);      break;
79     case IAND: printf("iand\n");
80         jvm_execute_instruction_iand(&java_virtual_machine);      break;
81     case IOR: printf("ior\n");
82         jvm_execute_instruction_ior(&java_virtual_machine);       break;
83     case IXOR: printf("ixor\n");
84         jvm_execute_instruction_ixor(&java_virtual_machine);      break;
85     }
86 }
87
88 printf("\nVariables dump\n");
89
90 int *variables_array = jvm_get_variables_array(&java_virtual_machine);
91 for(int i=0; i<N; ++i) {
92     printf("%08x\n", variables_array[i]);
93     int n = variables_array[i];
94     if(socket_send_int(&skt_server, n) == ERROR) {
95         socket_shutdown_destroy(&skt_acceptor);
96         jvm_destroy(&java_virtual_machine);
97         return ERROR;
98     }
99 }
100
101 socket_shutdown_destroy(&skt_acceptor);
102 socket_shutdown_destroy(&skt_server);
103 jvm_destroy(&java_virtual_machine);
104
105 return SUCCESS;
106 }

```

sep 24, 18 22:35

node.h

Page 1/1

```

1  #ifndef __NODE_H__
2  #define __NODE_H__
3
4  /**/
5
6  typedef struct node_t{
7      int data;
8      struct node_t* next;
9  } node_t;
10
11 // Crea el nodo.
12 // Pre: self apunta un sector valido de memoria
13 void node_create(node_t* self);
14 // Setea el dato "data" en el nodo.
15 // Pre: self apunta un sector valido de memoria
16 void node_set_data(node_t* self, int data);
17 // Devuelve el dato que guarda el nodo.
18 // Pre: self apunta un sector valido de memoria
19 int node_get_data(node_t* self);
20 // Setea la direccion del nodo al que apunta la instancia de nodo self.
21 // Pre: self apunta un sector valido de memoria
22 void node_set_next(node_t* self, node_t* next);
23 // Devuelve el nodo apuntado por la instancia "self".
24 // Pre: self apunta un sector valido de memoria
25 node_t* node_get_next(node_t* self);
26 // Destruye los recursos asociados a la instancia de nodo self.
27 // Pre: self apunta un sector valido de memoria
28 void node_destroy(node_t* self);
29
30 #endif

```

sep 24, 18 22:35

node.c

Page 1/1

```

1  #include "node.h"
2  #include <stdio.h>
3
4  void node_create(node_t* self) {
5      self->data = 0;
6      self->next = NULL;
7  }
8
9  void node_set_data(node_t* self, int data) {
10     self->data = data;
11 }
12
13 int node_get_data(node_t* self) {
14     return self->data;
15 }
16
17 void node_set_next(node_t* self, node_t* next) {
18     self->next = next;
19 }
20
21 node_t* node_get_next(node_t* self) {
22     return self->next;
23 }
24
25 void node_destroy(node_t* self) {
26 }

```

sep 24, 18 22:35

main.c

Page 1/1

```

1  #include <string.h>
2
3  #include "client.h"
4  #include "server.h"
5
6  #include "stack.h"
7  #include <stdio.h>
8
9  #define SUCCESS 0
10 #define ERROR -1
11
12 int main(int argc, char* argv[]) {
13     if (argv[1] == NULL) return ERROR;
14
15     if (strcmp(argv[1], "client") == SUCCESS) return client(argv);
16     else if (strcmp(argv[1], "server") == SUCCESS) return server(argv);
17     else return ERROR;
18 }

```

sep 24, 18 22:35

java_virtual_machine.h

Page 1/1

```

1  #ifndef __JAVA_VIRTUAL_MACHINE_H__
2  #define __JAVA_VIRTUAL_MACHINE_H__
3
4  /** Java Virtual Machine */
5
6  #include "stack.h"
7
8  typedef struct {
9      stack_t* stack;
10     int *numbers;
11 } jvm_t;
12
13 // Crea la virtual machine.
14 void jvm_create(jvm_t *self, int N);
15 // Ejecuta la instruccion bipush.
16 // Pre: self apunta un sector valido de memoria.
17 void jvm_execute_instruction_bipush(jvm_t *self, int data);
18 // Ejecuta la instruccion dup.
19 // Pre: self apunta un sector valido de memoria.
20 void jvm_execute_instruction_dup(jvm_t *self);
21 // Ejecuta la instruccion store.
22 // Pre: self apunta un sector valido de memoria.
23 void jvm_execute_instruction_istore(jvm_t *self, char c);
24 // Ejecuta la instruccion load.
25 // Pre: self apunta un sector valido de memoria.
26 void jvm_execute_instruction_ildload(jvm_t *self, char c);
27 // Ejecuta la instruccion and.
28 // Pre: self apunta un sector valido de memoria.
29 void jvm_execute_instruction_iand(jvm_t *self);
30 // Ejecuta la instruccion or.
31 // Pre: self apunta un sector valido de memoria.
32 void jvm_execute_instruction_ior(jvm_t *self);
33 // Ejecuta la instruccion xor.
34 // Pre: self apunta un sector valido de memoria.
35 void jvm_execute_instruction_ixor(jvm_t *self);
36 // Ejecuta la instruccion rem.
37 // Pre: self apunta un sector valido de memoria.
38 void jvm_execute_instruction_irem(jvm_t *self);
39 // Ejecuta la instruccion neg.
40 // Pre: self apunta un sector valido de memoria.
41 void jvm_execute_instruction_ineg(jvm_t *self);
42 // Ejecuta la instruccion div.
43 // Pre: self apunta un sector valido de memoria.
44 void jvm_execute_instruction_idiv(jvm_t *self);
45 // Ejecuta la instruccion add.
46 // Pre: self apunta un sector valido de memoria.
47 void jvm_execute_instruction_iadd(jvm_t *self);
48 // Ejecuta la instruccion mul.
49 // Pre: self apunta un sector valido de memoria.
50 void jvm_execute_instruction_imul(jvm_t *self);
51 // Ejecuta la instruccion sub.
52 // Pre: self apunta un sector valido de memoria.
53 void jvm_execute_instruction_isub(jvm_t *self);
54 // Devuelve las variables.
55 // Pre: self apunta un sector valido de memoria.
56 int* jvm_get_variables_array(jvm_t *self);
57 // Libera los recursos asociados a la instancia self.
58 // Pre: self apunta un sector valido de memoria.
59 void jvm_destroy(jvm_t *self);
60
61 #endif

```

sep 24, 18 22:35

java_virtual_machine.c

Page 1/2

```

1  #include "java_virtual_machine.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  void jvm_create(jvm_t *self, int N) {
6      self->stack = malloc(sizeof(stack_t));
7      stack_create(self->stack);
8
9      self->numbers = malloc(sizeof(int)*N);
10     for (int i=0; i<N; ++i) self->numbers[i] = 0;
11 }
12
13 void jvm_execute_instruction_bipush(jvm_t *self, int number) {
14     stack_push(self->stack, number);
15 }
16
17 void jvm_execute_instruction_dup(jvm_t *self) {
18     stack_push(self->stack, stack_top(self->stack));
19 }
20
21 void jvm_execute_instruction_istore(jvm_t *self, char c) {
22     char hex[3];
23     hex[0] = '\0';
24     snprintf(hex, sizeof(hex), "%02X", c);
25     hex[2] = '\0';
26     int pos = (int) strtol(hex, (char **)NULL, 10);
27     self->numbers[pos] = stack_pop(self->stack);
28 }
29
30 void jvm_execute_instruction_ildload(jvm_t *self, char c) {
31     char hex[3];
32     hex[0] = '\0';
33     snprintf(hex, sizeof(hex), "%02X", c);
34     hex[2] = '\0';
35     int pos = (int) strtol(hex, (char **)NULL, 10);
36     stack_push(self->stack, self->numbers[pos]);
37 }
38
39 void jvm_execute_instruction_iadd(jvm_t *self) {
40     stack_push(self->stack, stack_pop(self->stack) + stack_pop(self->stack));
41 }
42
43 void jvm_execute_instruction_isub(jvm_t *self) {
44     int a = stack_pop(self->stack);
45     int b = stack_pop(self->stack);
46     stack_push(self->stack, b - a);
47 }
48
49 void jvm_execute_instruction_imul(jvm_t *self) {
50     stack_push(self->stack, stack_pop(self->stack) * stack_pop(self->stack));
51 }
52
53 void jvm_execute_instruction_idiv(jvm_t *self) {
54     int a = stack_pop(self->stack);
55     int b = stack_pop(self->stack);
56     stack_push(self->stack, b / a);
57 }
58
59 void jvm_execute_instruction_irem(jvm_t *self) {
60     int a = stack_pop(self->stack);
61     int b = stack_pop(self->stack);
62     stack_push(self->stack, b % a);
63 }
64
65 void jvm_execute_instruction_ineg(jvm_t *self) {
66     stack_push(self->stack, -stack_pop(self->stack));

```


sep 24, 18 22:35

java_virtual_machine.c

Page 2/2

```

67 }
68
69 void jvm_execute_instruction_iand(jvm_t *self) {
70     stack_push(self->stack, stack_pop(self->stack) & stack_pop(self->stack));
71 }
72
73 void jvm_execute_instruction_ior(jvm_t *self) {
74     stack_push(self->stack, stack_pop(self->stack) | stack_pop(self->stack));
75 }
76
77 void jvm_execute_instruction_ixor(jvm_t *self) {
78     stack_push(self->stack, stack_pop(self->stack) ^ stack_pop(self->stack));
79 }
80
81 int* jvm_get_variables_array(jvm_t *self) {
82     return self->numbers;
83 }
84
85 void jvm_destroy(jvm_t *self) {
86     stack_destroy(self->stack);
87     free(self->stack);
88     free(self->numbers);
89 }

```

sep 24, 18 22:35

client.h

Page 1/1

```

1  #ifndef __CLIENT_H__
2  #define __CLIENT_H__
3
4  // Funcion encargada de realizar las tareas del cliente.
5  int client(char *argv[]);
6
7  #endif

```

sep 24, 18 22:35

client.c

Page 1/1

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include "socket.h"
4
5  #define ERROR -1
6  #define SUCCESS 0
7
8  int client(char *argv[]) {
9      // argv[2] = Hostname
10     // argv[3] = Port
11     // argv[4] = N
12     // argv[5] = Namefile
13
14     if(argv[2] == NULL ∨ argv[3] == NULL ∨ argv[4] == NULL) return ERROR;
15
16     FILE* input;
17
18     if (argv[5] == NULL) {
19         input = stdin;
20     } else {
21         input = fopen(argv[5], "rb");
22     }
23
24     if (input == NULL) return ERROR;
25
26     socket_t skt_client;
27
28     short unsigned int port = (short unsigned int) strtol(argv[3], NULL, 10);
29     int N = (int) strtol(argv[4], NULL, 10);
30
31     if (socket_create(&skt_client) == ERROR) return ERROR;
32     if (socket_connect(&skt_client, argv[2], port) == ERROR) return ERROR;
33
34     if (socket_send_int(&skt_client, N) == ERROR) return ERROR;
35
36     do {
37         char c = fgetc(input);
38         if(socket_send_char(&skt_client, c) == ERROR) {
39             if(input != stdin) fclose(input);
40             return ERROR;
41         }
42     } while (!feof(input));
43
44     if(input != stdin) fclose(input);
45
46     socket_shutdown_write(&skt_client);
47
48     printf("%s\n", "Variables dump");
49
50     for(int i = 0; i<N; ++i) {
51         int number;
52         if (socket_recv_int(&skt_client, &number) == ERROR) return ERROR;
53         printf("%08x\n", number);
54     }
55
56     socket_shutdown_read(&skt_client);
57
58     socket_destroy(&skt_client);
59
60     return SUCCESS;
61 }

```

sep 24, 18 22:35

bytecodes.h

Page 1/1

```

1  #define ISTORE 0x36
2  #define ILOAD 0x15
3  #define BIPUSH 0x10
4  #define DUP 0x59
5  #define IAND 0x7E
6  #define IXOR 0x82
7  #define IOR 0x80
8  #define IREM 0x70
9  #define INEG 0x74
10 #define IDIV 0x6C
11 #define IADD 0x60
12 #define IMUL 0x68
13 #define ISUB 0x64

```

sep 24, 18 22:35

Table of Content

Page 1/1

1	Table of Contents					
2	1	<i>stack.h</i> sheets	1 to	1 (1) pages	1- 1 30 lines
3	2	<i>stack.c</i> sheets	1 to	1 (1) pages	2- 2 39 lines
4	3	<i>socket.h</i> sheets	2 to	2 (1) pages	3- 4 95 lines
5	4	<i>socket.c</i> sheets	3 to	4 (2) pages	5- 8 218 lines
6	5	<i>server.h</i> sheets	5 to	5 (1) pages	9- 9 8 lines
7	6	<i>server.c</i> sheets	5 to	6 (2) pages	10- 11 107 lines
8	7	<i>node.h</i> sheets	6 to	6 (1) pages	12- 12 31 lines
9	8	<i>node.c</i> sheets	7 to	7 (1) pages	13- 13 27 lines
10	9	<i>main.c</i> sheets	7 to	7 (1) pages	14- 14 19 lines
11	10	<i>java_virtual_machine.h</i>	sheets	8 to	8 (1) pages	15- 15 62 lines
12	11	<i>java_virtual_machine.c</i>	sheets	8 to	9 (2) pages	16- 17 90 lines
13	12	<i>client.h</i> sheets	9 to	9 (1) pages	18- 18 8 lines
14	13	<i>client.c</i> sheets	10 to	10 (1) pages	19- 19 62 lines
15	14	<i>bytecodes.h</i> sheets	10 to	10 (1) pages	20- 20 14 lines