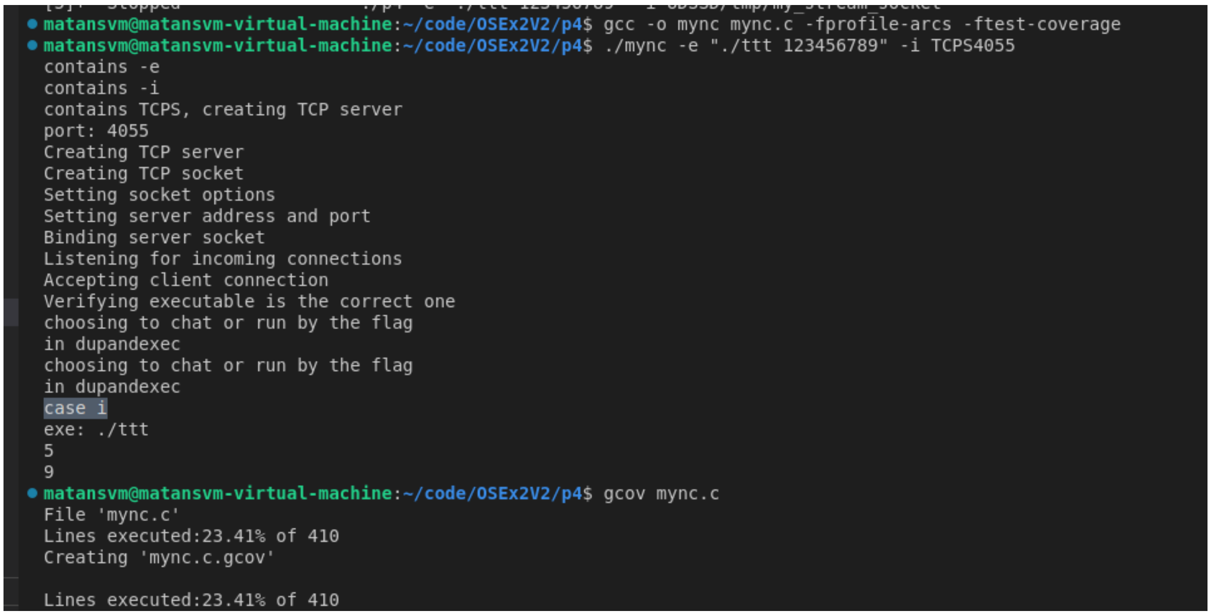
כיסוי קוד:

בוצע כיסוי קוד לנדרש אך משום שמדובר על קוד ענק בכל הרצה רק מספר קטן באופן יחסי הורץ, הסיבה לכיסוי הנמוח היא מורכבות הקוד ולא אי שימוש בשורות.

בצורפת ביקת כיסוי קוד למטה וצילם מסך של הטרמינל.



```
● matansvm@matansvm-virtual-machine:~/code/OSEx2V2/p4$ gcc -o mync mync.c -fprofile-arcs -ftest-coverage
● matansvm@matansvm-virtual-machine:~/code/OSEx2V2/p4$ ./mync -e "./ttt 123456789" -i TCPS4055
  contains -e
  contains -i
  contains TCPS, creating TCP server
  port: 4055
  Creating TCP server
  Creating TCP socket
  Setting socket options
  Setting server address and port
  Binding server socket
  Listening for incoming connections
  Accepting client connection
  Verifying executable is the correct one
  choosing to chat or run by the flag
  in dupandexec
  choosing to chat or run by the flag
  in dupandexec
  case i
  exe: ./ttt
  5
  9
● matansvm@matansvm-virtual-machine:~/code/OSEx2V2/p4$ gcov mync.c
  File 'mync.c'
  Lines executed:23.41% of 410
  Creating 'mync.c.gcov'

  Lines executed:23.41% of 410
```

```
-:    0:Source:mync.c
-:    0:Graph:mync.gcno
-:    0:Data:mync.gcda
-:    0:Runs:3
-:    1:#include <stdio.h>
-:    2:#include <stdlib.h>
-:    3:#include <string.h>
-:    4:#include <unistd.h>
-:    5:#include <arpa/inet.h>
-:    6:#include <sys/wait.h>
-:    7:#include <stdbool.h>
-:    8:#include <signal.h>
-:    9:#include <time.h>
-:   10:#include <stdio.h>
-:   11:#include <sys/wait.h>
```

```
        -:   12:#include <stdbool.h>
        -:   13:#include <getopt.h>
        -:   14:#include <errno.h>
        -:   15:#include <sys/un.h>
        -:   16:#include <sys/socket.h>
        -:   17:
        -:   18:
        -:   19:// Buffer size for communication
        -:   20:#define BUFFER_SIZE 1024
        -:   21:
        -:   22:
        -:   23:// Function to execute a program with specific I/O redirection based on mode
        2:   24:void dupandexec(pid_t pid, char mode, int client_sock, int server_sock, const char *exe, const char *arg){
        2:   25:    printf("in dupandexec\n");
        -:   26:    //explanation to this part:
        -:   27:    // we dont have to change the input cecause the input is already coming from the stdin (to the socket)
        -:   28:    // we have to change the output to the socket because we wnat to despaly the output to the client
        2:   29:    if (pid == 0) { // Child process
        1:   30:        switch (mode) {
        1:   31:            case 'i':
        1:   32:                printf("case i\n");
        -:   33:                // print the exe
        1:   34:                printf("exe: %s\n", exe);                                   //debug print delete later
        1:   35:                dup2(client_sock, 1);  // Redirect standard output to the client socket
        1:   36:                printf("dup2 done\n");                                      //debug print delete later
        1:   37:                break;
        -:   38:
    #####:   39:            case 'b':
```

```
#####:   40:         printf("case b\n");
#####:   41:            dup2(client_sock, 0);  // Redirect both input and output
#####:   42:            dup2(client_sock, 1);
#####:   43:            break;
    -:   44:      }
    -:   45:      // Execute the specified program
    1:   46:       printf("Executing command: %s\n", exe);
    -:   47:      // Prepare the argument vector
    1:   48:       char *args[] = { (char *)exe, (char *)arg, NULL };  // Argument vector
    -:   49:      // Execute the command using execvp
    1:   50:       execvp(exe, args);
    -:   51:      // If execvp returns, there was an error
    1:   52:       perror("Execution of exe failed");
#####:   53:       exit(EXIT_FAILURE);
    -:   54:   }
    -:   55:
    -:   56:   // Wait for the child process to finish
    -:   57:   int status;
    1:   58:    waitpid(pid, &status, 0);
    -:   59:
    -:   60:   // Close the sockets based on whether client_sock and server_sock are the same
    1:   61:    if (client_sock == server_sock) {
#####:   62:       close(client_sock);
    -:   63:   } else {
    1:   64:       close(server_sock);
    1:   65:       close(client_sock);
    -:   66:   }
    1:   67:}
    -:   68:
    -:   69:// Function to handle communication over a TCP connection
    1:   70:void TCPchat(int client_sock, pid_t pid, char *buffer){
        1:   71:    printf("Welcome to TCP chat\n");
        1:   72:    if (pid == 0) { // Child process: read from server
        7:   73:        while (1) {
        -:   74:            // Read data from the client
        8:   75:            ssize_t bytes_read = read(client_sock, buffer, BUFFER_SIZE - 1);
        8:   76:            if (bytes_read <= 0) {
        1:   77:                break; // Exit loop if no data or error occurs
        -:   78:            }
        7:   79:            buffer[bytes_read] = '\0'; // Null-terminate the string
        7:   80:            printf("Received from server: %s\n", buffer); // Print received data
        -:   81:        }
        1:   82:        close(client_sock); // Close the socket
        1:   83:        exit(0); // Exit child process
        -:   84:    } else { // Parent process: write to server
        -:   85:        while (1) {
    #####:   86:            printf("Enter message (-1 to quit): \n");
        -:   87:            // Get user input
    #####:   88:            fgets(buffer, sizeof(buffer), stdin);
        -:   89:            //check for fgets error
    #####:   90:            if(ferror(stdin)){
    #####:   91:                perror("fgets failed");
    #####:   92:                exit(EXIT_FAILURE);
        -:   93:            }
        -:   94:            // Remove newline character
    #####:   95:            buffer[strcspn(buffer, "\n")] = '\0'; // Remove newline character
        -:   96:
        -:   97:            // Exit loop if user enters "-1"
    #####:   98:            if (strcmp(buffer, "-1") == 0) {
    #####:   99:                break;
        -:  100:            }
        -:  101:
```

```
        -:  102:            // Send user input to the server (check if the buffer is not empty)
    #####:  103:            if (strlen(buffer) > 0){
    #####:  104:                if (write(client_sock, buffer, strlen(buffer)) <= 0) {
    #####:  105:                    break;
        -:  106:                }
        -:  107:            }
        -:  108:        }
        -:  109:        // Close the socket and kill the child process that is reading from the server
    #####:  110:        printf("Closing client socket and kiling the proces\n");
    #####:  111:        close(client_sock);
    #####:  112:        kill(pid, SIGKILL);
        -:  113:    }
    #####:  114:}
        -:  115:
        -:  116:
        -:  117:
        -:  118:// Function to create a TCP client and initiate communication
    #####:  119:int TCPclient(int port, char mode, const char *server_ip){
    #####:  120:    printf("Creating TCP client\n");
        -:  121:    int client_sock;
        -:  122:    struct sockaddr_in server_addr;
        -:  123:    char buffer[BUFFER_SIZE];
        -:  124:
        -:  125:    // Create a TCP socket
    #####:  126:    printf("Creating TCP socket\n");
    #####:  127:    client_sock = socket(AF_INET, SOCK_STREAM, 0);
    #####:  128:    if (client_sock < 0) {
    #####:  129:        perror("socket failed");
    #####:  130:        exit(EXIT_FAILURE);
        -:  131:    }
        -:  132:
```

```
        -:  133:    // Set server address and port
    #####:  134:    printf("Setting server address and port\n");
    #####:  135:    server_addr.sin_family = AF_INET;
    #####:  136:    server_addr.sin_port = htons(port);
    #####:  137:    server_addr.sin_addr.s_addr = inet_addr(server_ip);
        -:  138:
        -:  139:    // Connect to the server
    #####:  140:    printf("Connecting to server\n");
    #####:  141:    if (connect(client_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    #####:  142:        perror("connect failed");
    #####:  143:        exit(EXIT_FAILURE);
        -:  144:    }
        -:  145:
        -:  146:    // Create a child process to handle communication
    #####:  147:    printf("Forking process to handle communication\n");
    #####:  148:    pid_t pid = fork();
        1:  149:    if (pid < 0) {
    #####:  150:        perror("fork failed");
    #####:  151:        exit(EXIT_FAILURE);
        -:  152:    }
        -:  153:
        -:  154:    // Call the TCPchat function to handle TCP communication
        1:  155:    printf("Calling TCPchat function\n");
        1:  156:    TCPchat(client_sock, pid, buffer);
    #####:  157:    return 0;
        -:  158:}
        -:  159:
        -:  160:// Function to create a TCP server and handle client connections
        1:  161:int TCPserver(char mode, int port, const char *exe, const char *arg, int flag){
        1:  162:    printf("Creating TCP server\n");
    -:  163:    int server_sock, client_sock;
    -:  164:    struct sockaddr_in server_addr, client_addr;
    -:  165:    char buffer[BUFFER_SIZE];
    -:  166:
    -:  167:    // Create a TCP socket
    1:  168:    printf("Creating TCP socket\n");
    1:  169:    server_sock = socket(AF_INET, SOCK_STREAM, 0);
    1:  170:    if (server_sock < 0) {
#####:  171:        perror("socket failed");
#####:  172:        exit(EXIT_FAILURE);
    -:  173:    }
    -:  174:
    -:  175:    // Set socket options to allow reuse of the address
    1:  176:    printf("Setting socket options\n");
    1:  177:    int optval = 1;
    1:  178:    setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
    -:  179:
    -:  180:    // Set server address and port
    1:  181:    printf("Setting server address and port\n");
    1:  182:    server_addr.sin_family = AF_INET;
    1:  183:    server_addr.sin_port = htons(port);
    1:  184:    server_addr.sin_addr.s_addr = INADDR_ANY;
    -:  185:
    -:  186:    // Bind the server socket
    1:  187:    printf("Binding server socket\n");
    1:  188:    if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
#####:  189:        perror("bind failed");
#####:  190:        exit(EXIT_FAILURE);
    -:  191:    }
    -:  192:
    -:  193:    // Listen for incoming connections
```

```
        1:  194:    printf("Listening for incoming connections\n");
        1:  195:    if (listen(server_sock, 5) < 0) {
    #####:  196:        perror("listen failed");
    #####:  197:        exit(EXIT_FAILURE);
        -:  198:    }
        -:  199:
        -:  200:    // Accept a client connection
        1:  201:    printf("Accepting client connection\n");
        1:  202:    socklen_t client_addr_size = sizeof(client_addr);
        1:  203:    client_sock = accept(server_sock, (struct sockaddr *)&client_addr,
&client_addr_size);
        1:  204:    if (client_sock < 0) {
    #####:  205:        perror("accept failed");
    #####:  206:        exit(EXIT_FAILURE);
        -:  207:    }
        -:  208:
        -:  209:    // Ensure the correct executable is specified
        1:  210:    printf("Verifying executable is the correct one\n");
        1:  211:    if(!flag){
        1:  212:        if (strcmp(exe, "./ttt") != 0) {
    #####:  213:            puts(exe);
    #####:  214:            perror("exe is not ./ttt");
    #####:  215:            exit(EXIT_FAILURE);
        -:  216:        }
        -:  217:    }
        -:  218:
        -:  219:    // Fork a child process to handle the client
        1:  220:    pid_t pid = fork();
        2:  221:    if (pid < 0) {
    #####:  222:        perror("fork failed");
    #####:  223:        exit(EXIT_FAILURE);
```

```
        -:  224:    }
        -:  225:
        -:  226:    // Handle communication or run a program based on the flag
        2:  227:    printf("choosing to chat or run by the flag\n");
        2:  228:    if (flag) {
    #####:  229:        TCPchat(client_sock, pid, buffer);
        -:  230:    } else {
        2:  231:        dupandexec(pid, mode, client_sock, server_sock, exe, arg);
        -:  232:    }
        1:  233:    return 0;
        -:  234:}
        -:  235:
        -:  236:// Function to handle communication over a UDP connection
    #####:  237:void UDPchat(int client_sock, pid_t pid, char *buffer, struct sockaddr_in
server_addr){
    #####:  238:    printf("Welcome to UDP chat\n");
    #####:  239:    if (pid == 0) { // Child process: read from server
    #####:  240:        while (1) {
        -:  241:            // Read data from the client using recvfrom (UDP)
    #####:  242:            ssize_t bytes_read = recvfrom(client_sock, buffer, BUFFER_SIZE, 0, NULL,
0);
    #####:  243:            if (bytes_read <= 0) {
    #####:  244:                break; // Exit loop if no data or error occurs
        -:  245:            }
    #####:  246:            buffer[bytes_read] = '\0'; // Null-terminate the string
    #####:  247:            printf("Received from server: %s\n", buffer); // Print received data
        -:  248:        }
    #####:  249:        close(client_sock); // Close the socket
    #####:  250:        exit(0); // Exit child process
        -:  251:    } else { // Parent process: write to server
        -:  252:        while (1) {
    #####:  253:            printf("Enter message (-1 to quit): \n");
```

```
        -:  254:        // Get user input
    #####:  255:        fgets(buffer, sizeof(buffer), stdin);
        -:  256:        // Remove newline character
    #####:  257:        buffer[strcspn(buffer, "\n")] = '\0';
        -:  258:
        -:  259:        // Exit loop if user enters "-1"
    #####:  260:        if (strcmp(buffer, "-1") == 0) {
    #####:  261:            break;
        -:  262:        }
        -:  263:        // Send user input to the server using sendto (UDP)
    #####:  264:        if (sendto(client_sock, buffer, strlen(buffer), 0, (struct sockaddr *)&server_addr, sizeof(server_addr)) <= 0) {
        -:  265:            // break;
        -:  266:        }
        -:  267:    }
        -:  268:    // Close the socket and kill the child process that is reading from the server
    #####:  269:    printf("Closing client socket and kiling the proces\n");
    #####:  270:    close(client_sock); // Close the socket
    #####:  271:    kill(pid, SIGKILL);
        -:  272:  }
    #####:  273:}
        -:  274:
        -:  275:// Function to create a UDP server and handle communication
    #####:  276:int UDPserver(char mode, int port, const char *exe, const char *arg, int flag, int timeout){
    #####:  277:    printf("Creating UDP server\n");
        -:  278:  int server_sock;
        -:  279:  struct sockaddr_in server_addr, client_addr;
        -:  280:  char buffer[BUFFER_SIZE];
        -:  281:
        -:  282:  // Create a UDP socket
    #####:  283:    printf("Creating UDP socket\n");
    #####:  284:    server_sock = socket(AF_INET, SOCK_DGRAM, 0);
    #####:  285:    if (server_sock < 0) {
    #####:  286:        perror("socket failed");
    #####:  287:        exit(EXIT_FAILURE);
        -:  288:    }
        -:  289:
        -:  290:    // Set socket options to allow reuse of the address
    #####:  291:    printf("Setting socket options\n");
    #####:  292:    int optval = 1;
    #####:  293:    setsockopt(server_sock, SOL_SOCKET, SO_REUSEADDR, &optval,
sizeof(optval));
        -:  294:
        -:  295:    // Set server address and port
    #####:  296:    printf("Setting server address and port\n");
    #####:  297:    server_addr.sin_family = AF_INET;
    #####:  298:    server_addr.sin_port = htons(port);
    #####:  299:    server_addr.sin_addr.s_addr = INADDR_ANY;
        -:  300:
        -:  301:    // Bind the server socket
    #####:  302:    printf("Binding server socket\n");
    #####:  303:    if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    #####:  304:        perror("bind failed");
    #####:  305:        exit(EXIT_FAILURE);
        -:  306:    }
        -:  307:
        -:  308:    // Ensure the correct executable is specified
    #####:  309:    printf("Verifying executable is the correct one\n");
    #####:  310:    if(!flag){
    #####:  311:        if (strcmp(exe, "./ttt") != 0) {
    #####:  312:            puts(exe);
    #####:  313:            perror("exe is not ./ttt");
    #####:  314:          exit(EXIT_FAILURE);
        -:  315:      }
        -:  316:   }
        -:  317:
        -:  318:   // // Receive data from the client
    #####:  319:   printf("Receiving data from the client\n");
    #####:  320:   socklen_t client_addr_size = sizeof(client_addr);
    #####:  321:   memset(&client_addr, 0, client_addr_size);
    #####:  322:   size_t bytes = recvfrom(server_sock, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&client_addr, &client_addr_size);
    #####:  323:   printf("Received from client: %s\n", buffer);
        -:  324:
        -:  325:   // Fork a child process to handle the client
    #####:  326:   pid_t pid = fork();
    #####:  327:   if (pid < 0) {
    #####:  328:      perror("fork failed");
    #####:  329:      exit(EXIT_FAILURE);
        -:  330:   }
        -:  331:
        -:  332:   // Set a timeout if specified
    #####:  333:   if (timeout > 0) {
    #####:  334:      printf("Setting timeout for %d seconds\n", timeout);
    #####:  335:      alarm(timeout);
        -:  336:   }
        -:  337:
        -:  338:   // Handle communication or run a program based on the run flag
    #####:  339:   printf("choosing to chat or run by the flag\n");
    #####:  340:   if (!flag) {
    #####:  341:      dupandexec(pid, mode, server_sock, server_sock, exe, arg);
    #####:  342:   }else if(flag && pid == 0){
    #####:  343:      UDPchat(server_sock, pid, buffer, client_addr);
```

```
    -:  344:    }
    -:  345:     else {
#####:  346:        UDPchat(server_sock, pid, buffer, client_addr);
    -:  347:      }
#####:  348:    return 0;
    -:  349:}
    -:  350:
    -:  351:// Function to create a UDP client and initiate communication
#####:  352:int UDPclient(int port, char mode, const char *server_ip, int timeout){
#####:  353:    printf("Creating UDP client\n");
    -:  354:    int client_sock;
    -:  355:    struct sockaddr_in server_addr;
    -:  356:    char buffer[BUFFER_SIZE];
    -:  357:
    -:  358:    // Create a UDP socket
#####:  359:    printf("Creating UDP socket\n");
#####:  360:    client_sock = socket(AF_INET, SOCK_DGRAM, 0);
#####:  361:    if (client_sock < 0) {
#####:  362:        perror("socket failed");
#####:  363:        exit(EXIT_FAILURE);
    -:  364:    }
    -:  365:
    -:  366:    // Set server address and port
#####:  367:    printf("Setting server address and port\n");
#####:  368:    server_addr.sin_family = AF_INET;
#####:  369:    server_addr.sin_port = htons(port);
#####:  370:    server_addr.sin_addr.s_addr = inet_addr(server_ip);
    -:  371:
    -:  372:    // Send data to the server
#####:  373:    printf("Sending data to the server\n");
```

```
    #####:  374:    sendto(client_sock, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&server_addr, sizeof(server_addr));
        -:  375:
        -:  376:    // Fork a child process to handle communication
    #####:  377:    pid_t pid = fork();
    #####:  378:    if (pid < 0) {
    #####:  379:        perror("fork failed");
    #####:  380:        exit(EXIT_FAILURE);
        -:  381:    }
        -:  382:
        -:  383:    // Set a timeout if specified
    #####:  384:    if (timeout > 0) {
    #####:  385:        printf("Setting timeout for %d seconds\n", timeout);
    #####:  386:        alarm(timeout);
        -:  387:    }
        -:  388:
        -:  389:    // Handle UDP communication
    #####:  390:    memset(buffer, 0, BUFFER_SIZE);
    #####:  391:    UDPchat(client_sock, pid, buffer, server_addr);
    #####:  392:    return 0;
        -:  393:}
        -:  394:
        -:  395:// Function to handle communication over a UNIX Datagram socket
    #####:  396:void UDSchat(int client_sock, pid_t pid, char* buffer, struct sockaddr_un server_addr) {
    #####:  397:    if (pid == 0) { // Child process: read from server
    #####:  398:        while (1) {
        -:  399:            // Read data from the client using recvfrom (UDP-style communication)
    #####:  400:            ssize_t bytes_read = recvfrom(client_sock, buffer, BUFFER_SIZE, 0, NULL, 0);
    #####:  401:            if (bytes_read <= 0) {
```

```
    #####:  402:            perror("read failed");  // If reading fails or connection is closed, break the loop
    #####:  403:            break;
        -:  404:        }
    #####:  405:        buffer[bytes_read] = '\0';  // Null-terminate the string
    #####:  406:        printf("Received from server: %s\n", buffer);  // Print the data received from server
        -:  407:    }
    #####:  408:    close(client_sock);  // Close the client socket after done reading
    #####:  409:    exit(0);  // Exit child process
        -:  410:  } else {  // Parent process: write to server
        -:  411:    while (1) {
    #####:  412:        printf("Enter message (-1 to quit): \n");
    #####:  413:        fgets(buffer, sizeof(buffer), stdin);  // Get user input
    #####:  414:        buffer[strcspn(buffer, "\n")] = '\0';  // Remove newline character
    #####:  415:        if (strcmp(buffer, "-1") == 0) {
    #####:  416:            break;  // Exit loop if user enters "-1"
        -:  417:        }
    #####:  418:        printf("%s\n", server_addr.sun_path);  // Print server path (for debug purposes)
        -:  419:
        -:  420:        // Send user input to the server using sendto (Datagram communication)
    #####:  421:        if (sendto(client_sock, buffer, strlen(buffer), 0, (struct sockaddr *)&server_addr, sizeof(server_addr)) <= 0) {
    #####:  422:            perror("sendto failed");  // If sendto fails, print an error and exit
    #####:  423:            break;
        -:  424:        }
        -:  425:    }
    #####:  426:    close(client_sock);  // Close client socket when done writing
    #####:  427:    kill(pid, SIGKILL);  // Kill the child process that was reading from the server
        -:  428:  }
    #####:  429:}
```

```
        -:  430:
        -:  431:// Function to create a UNIX Domain Socket Datagram (SOCK_DGRAM) server
    #####:  432:int UDSDatagramServer(char mode, char *exe, const char* path, const char *arg, int run, int timeout) {
        -:  433:    int server_sock;  // Server socket file descriptor
        -:  434:    struct sockaddr_un server_addr, client_addr;  // Structures for server and client addresses
        -:  435:    char buffer[BUFFER_SIZE];  // Buffer for communication
        -:  436:
        -:  437:    // Check if the executable is "ttt", if not, print error and exit
    #####:  438:    if (strcmp(exe, "./ttt") != 0) {
    #####:  439:        puts(exe);
    #####:  440:        perror("exe is not ttt sadge");
    #####:  441:        exit(EXIT_FAILURE);
        -:  442:    }
        -:  443:
        -:  444:    // Create a UNIX Datagram socket (SOCK_DGRAM)
    #####:  445:    server_sock = socket(AF_UNIX, SOCK_DGRAM, 0);
    #####:  446:    if (server_sock < 0) {
    #####:  447:        perror("socket failed");  // Print error if socket creation fails
    #####:  448:        exit(EXIT_FAILURE);
        -:  449:    }
        -:  450:
    #####:  451:    int optval = 1;  // This can be used if you want to set socket options
        -:  452:
        -:  453:    // Set the server address family to UNIX domain and assign the file path
    #####:  454:    server_addr.sun_family = AF_UNIX;
    #####:  455:    strcpy(server_addr.sun_path, path);  // Copy the file path to the address structure
        -:  456:
        -:  457:    // Unlink (remove) any previous socket file with the same name to avoid conflicts
    #####:  458:    unlink(server_addr.sun_path);
```

```
        -:  459:
        -:  460:    // Bind the server socket to the specified address and path
    #####:  461:    if (bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    #####:  462:        perror("bind failed");  // Print error if bind fails
    #####:  463:        exit(EXIT_FAILURE);
        -:  464:    }
        -:  465:
        -:  466:    // Set the length of the client address structure for receiving messages
    #####:  467:    socklen_t client_addr_len = sizeof(client_addr);
        -:  468:
        -:  469:    // Wait for a message from the client (blocking call)
    #####:  470:    size_t bytes = recvfrom(server_sock, buffer, sizeof(buffer), 0, (struct sockaddr *)&client_addr, &client_addr_len);
    #####:  471:    printf("Received from client: %s\n", buffer);  // Print the message received from the client
        -:  472:
        -:  473:    // Print the client's socket path for debugging
    #####:  474:    printf("lo nahon %s\n", client_addr.sun_path);
        -:  475:
        -:  476:    // Fork a new process to handle communication with the client
    #####:  477:    pid_t pid = fork();
    #####:  478:    if (pid < 0) {
    #####:  479:        perror("fork failed");  // Print error if fork fails
    #####:  480:        exit(EXIT_FAILURE);
        -:  481:    }
        -:  482:
        -:  483:    // If a timeout is specified, set an alarm to handle the timeout
    #####:  484:    if (timeout > 0) {
    #####:  485:        alarm(timeout);  // Set the timeout
        -:  486:    }
        -:  487:
        -:  488:    // Based on the 'run' flag, either handle the communication or execute the program
```

```
#####:  489:    if (run) {
    -:  490:        // Run communication using UNIX datagram sockets (Datagram chat function)
#####:  491:        UDSchat(server_sock, pid, buffer, client_addr);
    -:  492:    } else {
    -:  493:        // Run the specified program using the child process
#####:  494:        dupandexec(pid, mode, server_sock, server_sock, exe, arg);
    -:  495:    }
#####:  496:    return 0;  // Return success
    -:  497:}
    -:  498:
    -:  499:
    -:  500:// Function to create a UNIX datagram socket client
#####:  501:int UDSDatagramClient(char mode, const char *path, int timeout) {
    -:  502:    int client_sock;
    -:  503:    struct sockaddr_un server_addr;
    -:  504:    char buffer[BUFFER_SIZE];
#####:  505:    strcpy(buffer, "barasi");  // Initial message to send to the server
    -:  506:
    -:  507:    // Create UNIX datagram socket
#####:  508:    client_sock = socket(AF_UNIX, SOCK_DGRAM, 0);
#####:  509:    if (client_sock < 0) {
#####:  510:        perror("socket failed");
#####:  511:        exit(EXIT_FAILURE);
    -:  512:    }
    -:  513:
    -:  514:    // Set up the server address structure
#####:  515:    server_addr.sun_family = AF_UNIX;
#####:  516:    strcpy(server_addr.sun_path, path);  // Copy the server path to the address structure
    -:  517:
    -:  518:    // Send initial message to the server
#####:  519:    sendto(client_sock, buffer, BUFFER_SIZE, 0, (struct sockaddr *)&server_addr, sizeof(server_addr));
    -:  520:
    -:  521:    // Fork a process to handle communication
#####:  522:    pid_t pid = fork();
#####:  523:    if (pid < 0) {
#####:  524:        perror("fork failed");
#####:  525:        exit(EXIT_FAILURE);
    -:  526:    }
    -:  527:
    -:  528:    // Set a timeout for the client (if specified)
#####:  529:    if (timeout > 0) {
#####:  530:        alarm(timeout);
    -:  531:    }
    -:  532:
    -:  533:    // Clear the buffer and start the communication process
#####:  534:    memset(buffer, 0, BUFFER_SIZE);
#####:  535:    UDSchat(client_sock, pid, buffer, server_addr);  // Handle client-server communication
#####:  536:    return 0;
    -:  537:}
    -:  538:
    -:  539:// Function to create a UNIX stream socket server
#####:  540:int UDSStreamServer(char mode, char *exe, const char *arg, int flag, const char *path) {
    -:  541:    struct sockaddr_un server, client;
#####:  542:    memset(&server, 0, sizeof(server));  // Clear the server address structure
#####:  543:    memset(&client, 0, sizeof(client));  // Clear the client address structure
#####:  544:    server.sun_family = AF_UNIX;
#####:  545:    strcpy(server.sun_path, path);  // Copy the path to the server address structure
    -:  546:
    -:  547:    int server_sock, client_sock;
    -:  548:    char buffer[BUFFER_SIZE];
    -:  549:
    -:  550:    // Create UNIX stream socket (for connection-oriented communication)
#####:  551:    server_sock = socket(AF_UNIX, SOCK_STREAM, 0);
#####:  552:    if (server_sock < 0) {
#####:  553:        perror("socket failed");
#####:  554:        exit(EXIT_FAILURE);
    -:  555:    }
    -:  556:
    -:  557:    // Unlink any previous socket with the same path
#####:  558:    unlink(server.sun_path);
    -:  559:
    -:  560:    // Bind the server socket to the specified path
#####:  561:    int len = strlen(server.sun_path) + sizeof(server.sun_family);
#####:  562:    printf("%s\n", server.sun_path);
#####:  563:    if (bind(server_sock, (struct sockaddr *)&server, len) < 0) {
#####:  564:        perror("bind failed");
#####:  565:        exit(EXIT_FAILURE);
    -:  566:    }
    -:  567:
    -:  568:    // Listen for incoming connections
#####:  569:    if (listen(server_sock, 1) < 0) {
#####:  570:        perror("listen failed");
#####:  571:        exit(EXIT_FAILURE);
    -:  572:    }
    -:  573:
    -:  574:    // Accept an incoming connection
#####:  575:    socklen_t client_addr_size = sizeof(client);
#####:  576:    client_sock = accept(server_sock, (struct sockaddr *)&client,
&client_addr_size);
#####:  577:    if (client_sock < 0) {
#####:  578:        perror("accept failed");
#####:  579:        exit(EXIT_FAILURE);
    -:  580:    }
    -:  581:
    -:  582:    // Fork a process to handle the client
#####:  583:    pid_t pid = fork();
#####:  584:    if (pid < 0) {
#####:  585:        perror("fork failed");
#####:  586:        exit(EXIT_FAILURE);
    -:  587:    }
    -:  588:
    -:  589:    // Handle communication or execute a program depending on the flag
#####:  590:    if (flag) {
#####:  591:        TCPchat(client_sock, pid, buffer);  // Handle stream communication
    -:  592:    } else {
#####:  593:        dupandexec(pid, mode, client_sock, server_sock, exe, arg);  // Run the specified program
    -:  594:    }
#####:  595:    return 0;
    -:  596:}
    -:  597:
    -:  598:// Function to create a UNIX stream socket client
#####:  599:int UDSStreamClient(char mode, const char *path) {
    -:  600:
    -:  601:    int client_sock;
#####:  602:    struct sockaddr_un server_addr = {.sun_family = AF_UNIX};
    -:  603:    char buffer[BUFFER_SIZE];
    -:  604:
    -:  605:    // Create UNIX stream socket
#####:  606:    printf("Creating UNIX stream socket\n");
#####:  607:    client_sock = socket(AF_UNIX, SOCK_STREAM, 0);
#####:  608:    if (client_sock < 0) {
#####:  609:        perror("socket failed");
#####:  610:        exit(EXIT_FAILURE);
    -:  611:    }
    -:  612:
    -:  613:    // Set the server address
#####:  614:    printf("Setting server address\n");
#####:  615:    strcpy(server_addr.sun_path, path);
    -:  616:
    -:  617:    // Connect to the server
#####:  618:    printf("Connecting to server\n");
#####:  619:    if (connect(client_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
#####:  620:        perror("connect failed");
#####:  621:        exit(EXIT_FAILURE);
    -:  622:    }
    -:  623:
    -:  624:    // Fork a process to handle communication
#####:  625:    pid_t pid = fork();
#####:  626:    if (pid < 0) {
#####:  627:        perror("fork failed");
#####:  628:        exit(EXIT_FAILURE);
    -:  629:    }
    -:  630:
    -:  631:    // Handle communication with the server
#####:  632:    TCPchat(client_sock, pid, buffer);
#####:  633:    return 0;
    -:  634:}
    -:  635:
    -:  636:
    -:  637:// Main function to handle command-line arguments and initiate server/client creation
```

```
1:  638:int main(int argc, char *argv[])
-:  639:{
-:  640:    //Exemple comends to run the program for every part of the program
-:  641:    // the formats of the arguments are as follows:
-:  642:    // PART 3
-:  643:    // ./mync -e "./ttt 123456789" -i TCPS4055
-:  644:    // ./mync -e "./ttt 123456789" -o TCPClocalhost,4055
-:  645:    // ./mync -e "./ttt 123456789" -b TCPS4055
-:  646:    // PART 3.5
-:  647:    // ./mync -e -i TCPS4060
-:  648:    // ./mync -e -o TCPClocalhost,4060
-:  649:    // PART 4
-:  650:    // ./mync -e "./ttt 123456789" -i UDPS4050 -t 10                    //with timeout
-:  651:    // ./mync -e "./ttt 123456789" -o UDPClocalhost,4050 -t 10           //with timeout
-:  652:    // ./mync -e "./ttt 123456789" -b UDPS4050 -t 10                    //with timeout
-:  653:    // PART 6
-:  654:    // ./mync -e "./ttt 123456789" -i UDSSD/tmp/sock -t 10              //with timeout
-:  655:    // ./mync -e "./ttt 123456789" -o UDSCD/tmp/sock -t 10              //with timeout
-:  656:
-:  657:    // ./mync -e "./ttt 123456789" -i UDSSS/tmp/sock -t 10
-:  658:    // ./mync -e "./ttt 123456789" -o UDSCS/tmp/sock -t 10
-:  659:
-:  660:    // over all it can be with or without timeout and it can be udp or tcp
1:  661:    int flag = 1;
1:  662:    char mode = 0;
1:  663:    int port = 0;
1:  664:    char *exe = NULL;
1:  665:    char *arg = NULL;
1:  666:    int timeout = -1;
-:  667:
    -:  668:    int opt;
    -:  669:    // Parse command-line options
    3:  670:    while ((opt = getopt(argc, argv, "e:t:iob")) != -1) {
    2:  671:        switch (opt) {
    1:  672:case 'e': // Executable argument
    1:  673:            printf("contains -e\n");
    1:  674:            exe = strtok(optarg, " ");  // First part of optarg is the executable
    1:  675:            arg = strtok(NULL, " ");    // The rest is the argument
    1:  676:            flag = 0;
    1:  677:            break;
    1:  678:            case 'i': // Input mode
    1:  679:                printf("contains -i\n");
    1:  680:                mode = opt;
    1:  681:                break;
#####:  682:            case 'o': // Output mode
#####:  683:                printf("contains -o\n");
#####:  684:                mode = opt;
#####:  685:                break;
#####:  686:            case 'b': // Both input and output mode
#####:  687:                printf("contains -b\n");
#####:  688:                mode = opt;
#####:  689:                break;
#####:  690:            case 't': // Timeout argument
#####:  691:                printf("contains -t\n");
#####:  692:                timeout = atoi(optarg);
#####:  693:                printf("timeout: %d\n", timeout);
#####:  694:                if (timeout <= 0) {
#####:  695:                    fprintf(stderr, "Timeout must be a positive integer\n");
#####:  696:                    exit(EXIT_FAILURE);
    -:  697:                }
#####:  698:                break;
```

```
        -:  699:        }
        -:  700:    }
        -:  701:
        -:  702:    // Process remaining command-line arguments
        -:  703:    char *junk;    // junk variable to get into the right part of the string
        6:  704:    for (int i = 0; i < argc; i++) {
        5:  705:        if (strstr(argv[i], "TCPS")) { // TCP Server
        1:  706:            printf("contains TCPS, creating TCP server\n");
        1:  707:            junk = strtok(argv[i], "S");
        1:  708:            port = atoi(strtok(NULL, "S"));
        1:  709:            printf("port: %d\n", port);
        1:  710:            TCPserver(mode, port, exe, arg, flag);
        -:  711:        }
        5:  712:        if (strstr(argv[i], "TCPC")) { // TCP Client
    #####:  713:        printf("contains TCPC, creating TCP client\n");
    #####:  714:            junk = strtok(argv[i], ",");
    #####:  715:            port = atoi(strtok(NULL, ","));
    #####:  716:            printf("port: %d\n", port);
    #####:  717:            TCPclient(port, mode, "127.0.0.1");
        -:  718:        }
        5:  719:        if (strstr(argv[i], "UDPS")) { // UDP Server
    #####:  720:            printf("contains UDPS, creating UDP server\n");
    #####:  721:            junk = strtok(argv[i], "S");
    #####:  722:            port = atoi(strtok(NULL, "S"));
    #####:  723:            printf("port: %d\n", port);
    #####:  724:            UDPserver(mode, port, exe, arg, flag, timeout);
        -:  725:        }
        5:  726:        if (strstr(argv[i], "UDPC")) { // UDP Client
    #####:  727:            printf("contains UDPC, creating UDP client\n");
    #####:  728:            junk = strtok(argv[i], ",");
    #####:  729:            port = atoi(strtok(NULL, ","));
```

```
#####:  730:          printf("port: %d\n", port);
#####:  731:          UDPclient(port, mode, "127.0.0.1", timeout);
    -:  732:      }
    5:  733:      if(strstr(argv[i],"UDSSD")){   //UDS Datagram Server
#####:  734:          printf("contains UDSSD, creating UDS datagram server\n");
#####:  735:          char* path = strtok(argv[i],"UDSSD");
#####:  736:          path += 5;
#####:  737:          UDSDatagramServer(mode, exe, path, arg, flag, timeout);
    -:  738:      }
    5:  739:      if(strstr(argv[i],"UDSCD")){  //UDS Datagram Client
#####:  740:          printf("contains UDSCD, creating UDS datagram client\n");
#####:  741:          char* path = strtok(argv[i],"UDSCD");
#####:  742:          path += 5;
#####:  743:          UDSDatagramClient(mode, path, timeout);
    -:  744:      }
    5:  745:      if(strstr(argv[i],"UDSSS")){ //UDS Stream Server
#####:  746:          printf("contains UDSSS, creating UDS stream server\n");
#####:  747:          char *path = strtok(argv[i],"UDSSS");
#####:  748:          path += 5;
#####:  749:          printf("exe:   %s\n",exe);
#####:  750:          printf("path:   %s\n",path);
#####:  751:          UDSStreamServer(mode, exe, arg, flag, path);
    -:  752:      }
    5:  753:      if(strstr(argv[i],"UDSCS")){ //UDS Stream Client
#####:  754:          printf("contains UDSCS, creating UDS stream client\n");
#####:  755:          char *path = strtok(argv[i],"UDSCS");
#####:  756:          path += 5;
#####:  757:          printf("%s\n",path);
#####:  758:          UDSStreamClient(mode, path);
    -:  759:      }
    -:  760:   }
    -:  761:
    -:  762:
    1:  763:    return 0;
    -:  764:}
```