ΕΡΓΑΛΕΙΑ ΑΝΑΠΤΥΞΗΣ ΛΟΓΙΣΜΙΚΟΥ

Αναφορά 3ης άσκησης

*Η υλοποίηση ενός μοντέλου **server** – **client** με χρήση pipes & sockets*

Βιριράκης Γεώργιος (2016030035)
Γαλάνης Μιχάλης (2016030036)

ΕΙΣΑΓΩΓΗ

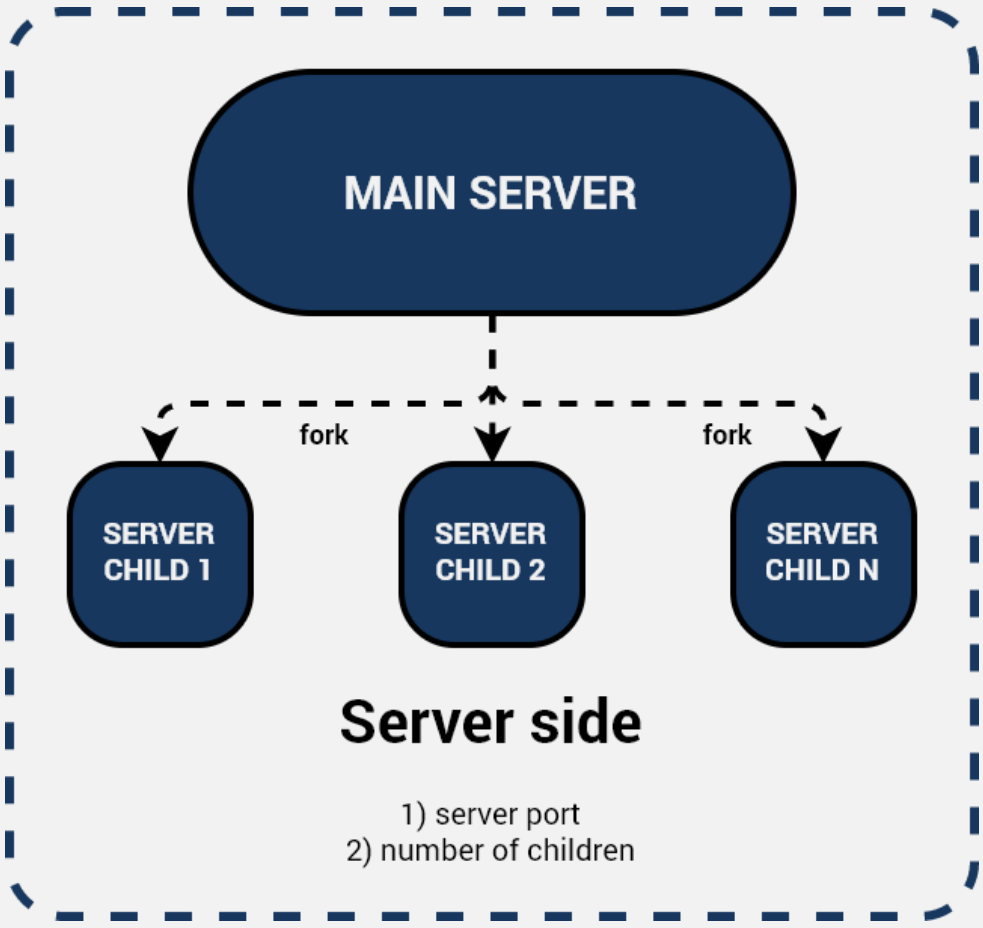
Στη τρίτη και τελευταία εργαστηριακή άσκηση, κληθήκαμε να κατασκευάσουμε ένα μοντέλο πελάτη-εξυπηρετητή σε γλώσσα προγραμματισμού C. Όλες οι λειτουργίες βασίζονται στη χρήση σωλήνων (pipes) και υποδοχών (sockets). Έπρεπε επίσης να λάβουμε υπόψιν μας θέματα όπως πρωτόκολλα επικοινωνίας, εκτέλεση εντολών πελάτη, συγχρονισμός, signals, διαχείριση αρχείων και δημιουργία makefile.

Η εργασία έχει πραγματοποιηθεί και πάλι από κοινού.

ΕΠΙΣΚΟΠΗΣΗ

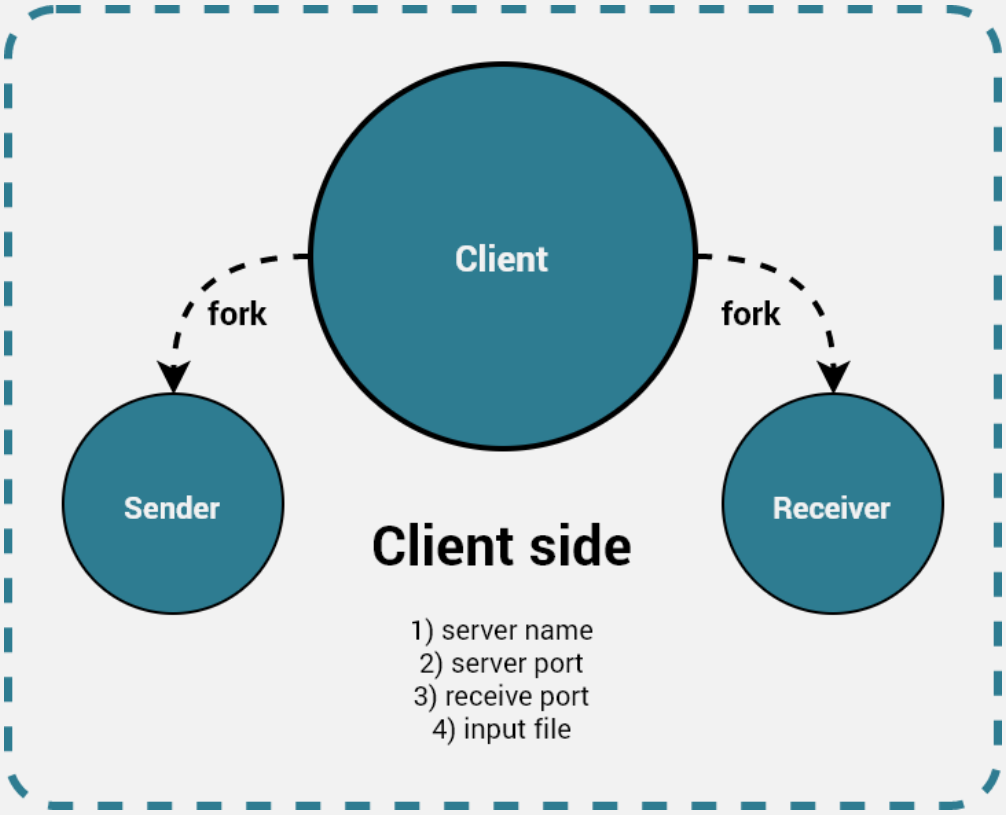
Server

Ο pre-fork εξυπηρετητής περιγράφεται από ένα port number και τον αριθμό των παιδιών – διεργασιών που θα δημιουργήσει. Κάθε παιδί λαμβάνει εξ ολοκλήρου μια εντολή ενός πελάτη που θα εκτελέσει. Η διαδικασία θα εξηγηθεί αναλυτικά αργότερα. Στο ακόλουθο σχήμα, φαίνεται η δομή του εξυπηρετητή.



Client

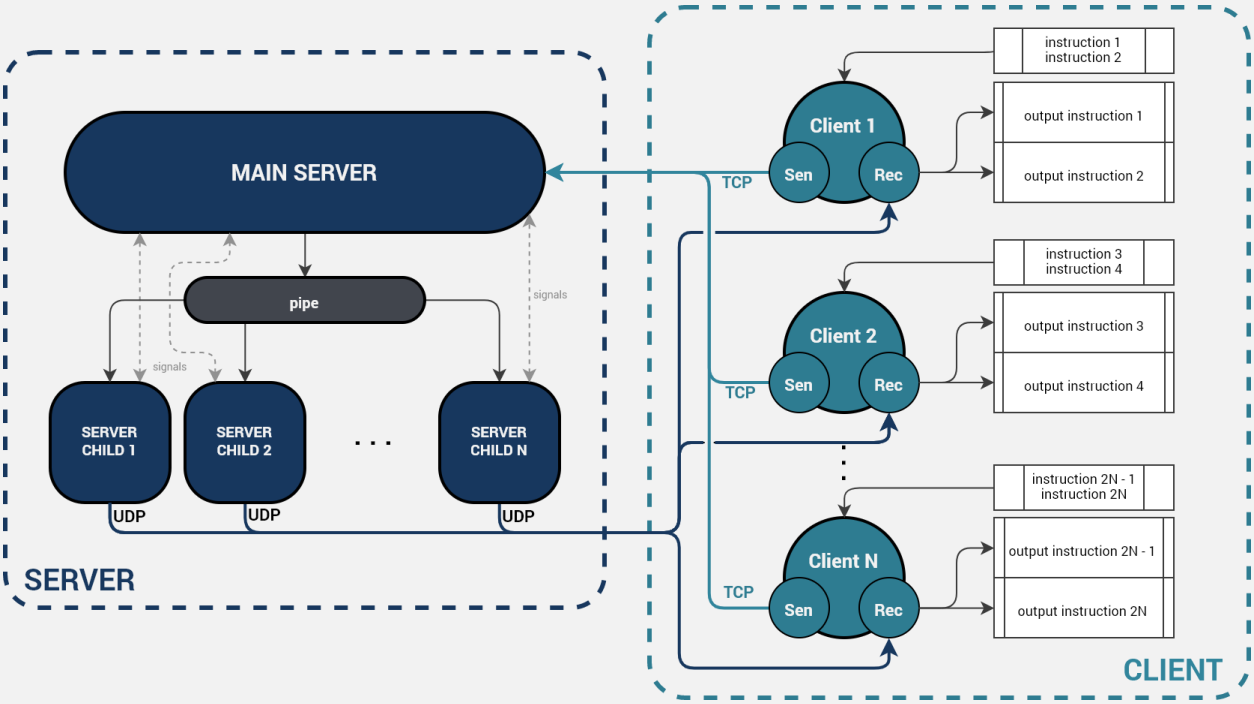
Ο πελάτης εκτελείται με όρισμα το όνομα και τη θύρα του εξυπηρετητή, τη θύρα επιστροφής καθώς και το αρχείο εισόδου εντολών. Ο client επίσης έχει 2 διεργασίες – παιδιά για την επίτευξη παραλληλισμού. Η sender διεργασία είναι υπεύθυνη για την ανάγνωση του αρχείου εισόδου και την αποστολή του TCP πακέτου στον εξυπηρετητή, ενώ η receiver διεργασία είναι υπεύθυνη για την παραλαβή UDP πακέτου από τον εξυπηρετητή και την εγγραφή της απάντησης σε αρχεία εξόδου. Η δομή του πελάτη εμφανίζεται στο επόμενο σχήμα.



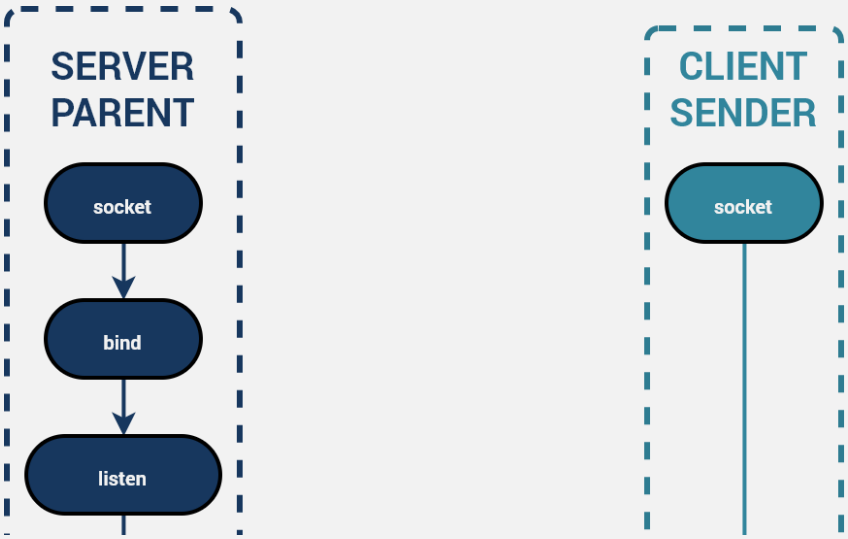
Γενικά

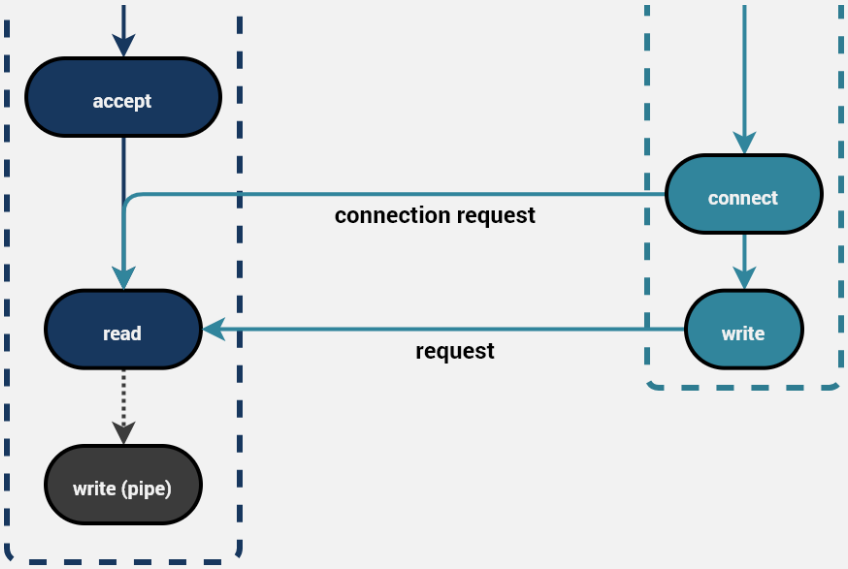
Όπως θα δούμε στη συνέχεια, κάθε γραμμή του αρχείου περιέχει τουλάχιστον μία εντολή. Ο client sender διαβάζει το αρχείο αυτό και δημιουργεί για κάθε γραμμή ένα TCP πακέτο το οποίο θα στείλει στον εξυπηρετητή. Η διεργασία πατέρα του server αφού λάβει το πακέτο αυτό και ελέγξει για το αν η εντολή ξεπερνάει τους 100 χαρακτήρες, την περνάει στο pipe. Το πρώτο διαθέσιμο παιδί θα αναλάβει την εντολή αυτή (θα την «φιλτράρει» σύμφωνα με τους δοθέντες κανόνες και θα την εκτελέσει), και θα αποθηκεύσει την απάντησή της σε πακέτα UDP τα οποία και στέλνει πίσω στον συγκεκριμένο πελάτη.

Ο πατέρας και τα παιδιά διεργασίες του server μπορούν να επικοινωνήσουν μεταξύ τους με signals για πληροφορίες όπως τερματισμό διεργασίας παιδιού και ολικό τερματισμό του εξυπηρετητή. Να σημειωθεί ότι ο εξυπηρετητής – πατέρας επιλέγει κατάλληλα κάθε φορά από ποιον πελάτη θα λάβει το TCP πακέτο. Υπάρχει επίσης αναμονή 5 δευτερολέπτων ανά 10 εντολές που στέλνει ο πελάτης. Η συνολική δομή του προγράμματος παρουσιάζεται στο επόμενο διάγραμμα.

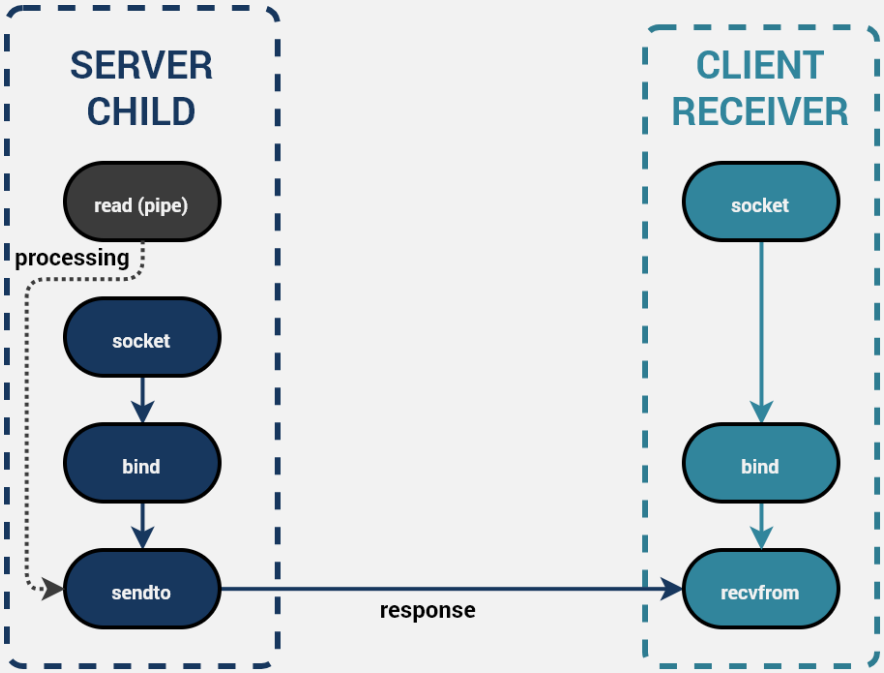


Στο παρακάτω διάγραμμα εμφανίζονται τα βήματα που πρέπει να κάνει η κάθε πλευρά για να επιτευχθεί TCP σύνδεση (πελάτης -> εξυπηρετητής):





Στο παρακάτω διάγραμμα εμφανίζονται τα βήματα που πρέπει να κάνει η κάθε πλευρά για να επιτευχθεί UDP σύνδεση (εξυπηρετητής -> πελάτης):



ΥΛΟΠΟΙΗΣΗ

Να σημειωθεί ότι θα αναλύσουμε μόνο τα σημαντικά σημεία του κώδικα για λόγους απλότητας.

Για να μπορέσει να εκτελεστεί ο πελάτης και ο εξυπηρετητής χρειάστηκε να κατασκευάσουμε ένα **makefile** (επειδή έχουμε πολλά source και header αρχεία). Αφού επιλέξαμε τον **gcc** compiler και σημειώσαμε τα **objects** και τα **dependencies**, προχωρήσαμε και στην δημιουργία μερικών run configurations όπως **server**, **client**, **fewclients** (3 clients), **manyclients** (20 clients).

Γενικά

Πακέτα Μεταφοράς: Τα πακέτα που στέλνουμε είναι structs. Έχουμε τρία ξεχωριστά structs για TCP πακέτο, UDP πακέτο και PIPE πακέτο.

TCP packet

Χρησιμοποιούμε μια μεταβλητή **order** για να μεταδώσουμε ως πληροφορία τη σειρά με την οποία στέλνουμε τα πακέτα. Το **receive_port** χρειάζεται έτσι ώστε να ξέρει το παιδί – εξυπηρετητής που να στείλει την απάντηση της εντολής. Η συμβολοσειρά **data** περιέχει την πληροφορία (εντολή) που έστειλε ο πελάτης.

```
42  //Structure of TCP packets
43  struct send_info {
44      int order;
45      int receive_port;
46      char data[TRANSFERRED_CMD_LEN];
47  };
```

UDP packet

Το order είναι το ίδιο με αυτό του TCP packet. Το **sub_order** χρησιμοποιείται για να γνωρίζουμε τη σειρά με την οποία επιστρέφουν τα πακέτα της εκτελεσμένης εντολής στον πελάτη. Το sub_last δηλώνει εάν αυτό το πακέτο είναι το τελευταίο της εκτελεσμένης εντολής. Το data εξηγήθηκε στο προηγούμενο πακέτο.

```
46  //Structure of UDP packets
47  struct receive_info {
48      int order;
49      int sub_order;
50      char data[MAX_UDP_LEN - 2 * sizeof(int) - sizeof(bool)];
51      bool sub_last;
52  };
```

Pipe packet

Το pipe packet περιέχει ένα **tcp_packet** καθώς και μια μεταβλητή για τη διεύθυνση του πελάτη **client_address**.

```
54  //Structure of Pipe packets
55  struct pipe_info {
56      struct in_addr client_address;
57      struct send_info si;
58  };
```

Server

Δημιουργία διεργασιών: Δημιουργούμε διεργασίες παιδιά με την εντολή **fork()** στη συνάρτηση main.

```c
int main(int argc, char *argv[]){ //server_port,  num_of_children
    //Checking parameters
    if (argc != 3) perror_exit("usage: server_port num_of_children!", ERR_S_PARAM);

    //Creating Pipe for children communication
    int pipe_fd[2];
    if (pipe(pipe_fd) == -1) perror_exit("Could not create pipe!", ERR_S_PIPE_CREATE);

    //Create num_of_children children processes
    int children_number = atoi(argv[2]);
    for (int i = 0; i < children_number; i++){
        pid_t pid;
        switch (pid = fork()) {
        case 0:
            close(pipe_fd[1]);
            serverChild(pipe_fd[0]); //pipe_fd[0] -> pipe_read
            close(pipe_fd[0]);
            exit(0);
        case -1:
            fprintf(stderr, "Could not initiate child server!\n");
            continue;
        }
    }

    //Begin the server
    close(pipe_fd[0]);
    serverParent(pipe_fd[1], atoi(argv[1])); //pipe_fd[1] -> pipe_write
    close(pipe_fd[1]);

    //Waiting for children to terminate
    while(wait(NULL) >= 0);
    fprintf(stderr, "[Parent %d] : Children Exited. Killing Parent\n", getpid());
    return 0;
}
```

Σήματα: Όσον αφορά τα σήματα, χρησιμοποιούμε 3 σήματα για την επικοινωνία πατέρα - παιδιών (**SIGUSR1**, **SIGTERM**, **SIGCHLD**). SIGUSR1 στέλνει το παιδί στον πατέρα όταν συναντά εντολή **timeToStop**. SIGCHLD στέλνει αυτόματα το παιδί στον πατέρα όταν αυτό πεθαίνει, ενώ SIGTERM στέλνει ο πατέρας στο παιδί ώστε αυτό να τερματιστεί. Επίσης αγνοούμε όλα τα σήματα **SIGPIPE** που τυχόν προκύπτουν. Αναθέτουμε handlers για τα σήματα αυτά με την συνάρτηση **sigaction.**

remoteServer.c:

```
7   //Stop handler (when "timeToStop")
8   bool stop_requested = false;
9   void stop_handler(int sig){
10      stop_requested = true;
11  }
12
13  //End handler (when "end")
14  void end_handler(int sig){
15      pid_t id;
16      while ((id = waitpid(-1, NULL, WNOHANG)) > 0);
17      if (id < 0) stop_requested = true;
18  }
19
20  //Term handler (when parent kills children)
21  bool child_terminate = false;
22  void term_handler(){
23      child_terminate = true;
24  }
```

Server Child:

```
68      //Assign handler for SIGTERM, SIGPIPE signal
69      struct sigaction term = {.sa_handler = term_handler, .sa_flags = 0};
70      sigaction(SIGTERM, &term, NULL);
71      signal(SIGPIPE, SIG_IGN);
72      printf("[Child %d] : Assigned signal handlers\n", getpid());
```

Server Parent:

```
188     //Assign handlers for SIGEND, SIGSTOP, SIG_CHLD, SIGPIPE signals
189     struct sigaction end = {.sa_handler = end_handler};
190     struct sigaction stop = {.sa_handler = stop_handler};
191     sigaction(SIGUSR1, &stop, NULL);
192     sigaction(SIGCHLD, &end, NULL);
193     signal(SIGPIPE, SIG_IGN);
```

Parsing: διαδικασία «φιλτραρίσματος» εισερχόμενης εντολής από πελάτη. Έχουμε μια αρχική συμβολοσειρά **original_data** που περιέχει την αρχική εντολή. Με ένα βρόγχο επανάληψης διατρέχουμε και ελέγχουμε αν κάθε **temp_instruction** εντολή είναι υποστηριζόμενη (ως εντολή εννοούμε μέχρι να συναντήσει ";", "|" ή "\0") έτσι ώστε να την περάσουμε στο τελικό **processed_data.** Ελέγχουμε επίσης για τυχόν τερματικές εντολές.

```
100     //Parsing instructions
101     while (1) { //Loops and examine each instruction seperately (ls /etc/ | rm -i) -> (ls /etc/) and (rm -i)
102         while(isspace(original_data[start_index])) start_index++; //Trimming leading space
103         end_index = start_index;
104         while(original_data[end_index] != ';' && original_data[end_index] != '|' && original_data[end_index] != '\0') end_index++;
105         strncpy(temp_instruction, original_data + start_index, end_index - start_index + 1); //copy to temp with offset and length
106 
107         //Examining if temp instruction is terminating instruction
108         if (strcmp(temp_instruction, END_INSTR) == 0){ // (end)
109             child_terminate = true;
110             break;
111         } else if (strcmp(temp_instruction, STOP_INSTR) == 0) // (timeToStop)
112             kill(getppid(), SIGUSR1); //send signal to parent
113 
114         //Examining if temp instruction is supported instruction
115         if (instr_valid(temp_instruction))
116             strncat(processed_data,temp_instruction, end_index - start_index + 1); //copies temp_instruction to processed_data
117         else {
118             processed_data[last_pipe_index] = '\0'; //removes pipe from string if it's been added by the previous instruction
119             break;
120         }
121 
122         //Examining the cause of the end index
123         if (original_data[end_index] == INSTR_SEMICOLON || original_data[end_index] == INSTR_END)
124             break;
125         else if (original_data[end_index] == INSTR_PIPE){   //space needed after pipe
126             last_pipe_index = end_index; //in case the next instruction is invalid
127             start_index = end_index + 1;
128             continue;
129         }
130 
131     }
```

Select: τη χρησιμοποιούμε για να δούμε ποιοι περιγραφητές έχουν διαθέσιμα δεδομένα χωρίς να μπλοκάρει αν κάπου δεν υπάρχουν δεδομένα.

```
221     //Accept connections and transfer data
222     struct in_addr client_addresses[FD_SETSIZE]; //Keep track of client addresses
223     fd_set active_set, read_set;
224     FD_ZERO(&active_set); //initializing set of active sockets
225     FD_SET(s, &active_set); //initializing set of active sockets
```

```
227     while(1) {
228         //Checks if (timeToStop)
229         if (stop_requested) break;
230 
231         read_set = active_set;
232         int sel = select(FD_SETSIZE, &read_set, NULL, NULL, NULL);
233         if (sel <= 0) continue;
234 
235         printf("[Parent %d] : %d file descriptors are ready.\n" , getpid(), sel);
236         for (int i = 0; i < FD_SETSIZE; i++){
237             if (FD_ISSET(i, &read_set)){
238                 if (i == s){ //Branch when new connection shows up
246                     assert(a <= FD_SETSIZE);
247                     FD_SET(a, &active_set);
```

```
249                 } else{ //Branch for existing connection
269                 }
270             }
271         }
272     }

274     //Closing active sets
275     for (int i = 0; i < FD_SETSIZE; i++){
276         if (FD_ISSET(i, &active_set)) close(i);
277     }
```

Client

Δημιουργία διεργασιών: Δημιουργούμε sender και receiver διεργασίες με την εντολή **fork()** στη συνάρτηση main.

```
7   int main(int argc, char *argv[]){ //server_name, server_port, receive_port, input_file
8       //Checking Parameters
9       if (argc != 5) perror_exit("usage: server_name server_port receive_port input_file!", ERR_C_PARAM);
10
11      //Initiating receiver-side of client
12      switch (fork()){
13          case 0: clientReceiver(atoi(argv[3]), argv[4]);
14          case -1: fprintf(stderr, "Could not initiate receiver-side of client!\n");
15      }
16      //Initiating sender-side of client
17      switch (fork()){
18          case 0: clientSender(argv[1], atoi(argv[2]), atoi(argv[3]), argv[4]);
19          case -1: fprintf(stderr, "Could not initiate sender-side of client!\n");
20      }
21      printf("[Parent %d] : Client Sender and Receiver created successfully!\n", getpid());
22
23      //Waiting for children to terminate
24      wait(NULL);
25      wait(NULL);
26      printf("[Parent %d] : Exiting Client %d!\n", getpid(), atoi(argv[3]));
27      return 0;
28  }
```

Input & Output: ο sender διαβάζει εντολές από το αρχείο εισόδου και ο receiver εξάγει απαντήσεις στο αρχείο εξόδου.

Sender (input):

```
58      //Reading instructions from input file and sending them to server parent
59      char line_buffer[MAX_LINE_LEN]; //each line is stored in this buffer
60      FILE *file = fopen(fileName, "r");
61      if (file == NULL) perror_exit("Could not open file!", ERR_CS_FILE_OPEN);
62      int line_counter = 0;
63      //Reading input file line by line and sending data
64      while(fgets(line_buffer, MAX_LINE_LEN, file)){ //for each line
85          line_counter++;
86      }
87      //Terminating connection
88      printf("[Sender %d] : Terminated connection with server %s:%d \n", getpi
89      close(file);
```

Receiver (output):

```
116     //Estimating number of instructions in input file.
117     //We need this so that we know how many "answers" we are waiting for.
118     FILE *input = fopen(fileName, "r");
119     char ch; int files = 1;
120     while ((ch = fgetc(input)) != EOF) if (ch == '\n') files++;
121     printf("[Receiver %d] : %d instructions found from %s\n", getpid(), files, fileName);
122
123     //Creating output files (keeps file fds open until writing is finished)
124     FILE *fds[files];
125     for (int i = 0; i < files; i++) {
126         char fileName[30];
127         sprintf(fileName, "output.%d.%d", receivePort, i);
128         fds[i] = fopen(fileName, "w");
129     }
130
131     //Receiving UDP Packets
132     while (1){
137         //Receiving data
138         int rcv = recvfrom(s, (char*)&ri, sizeof(struct receive_info), MSG_WAITALL, (stru
145         //Writing received data to output files
146         FILE *f = fds[ri.order];
147         fseek(f, sizeof(ri.data) * ri.sub_order, SEEK_SET);
148         fwrite(ri.data, strnlen(ri.data, sizeof(ri.data)), 1, f);
149
150         //This receiver is done when it receives the final part of the last executed inst
151         if (ri.order == files - 1 && ri.sub_last) break;
152     }
153     //Closing file fds and exiting
154     printf("[Receiver %d] : Shutting down!\n", getpid());
155     for (size_t i = 0; i < files; i++) fclose(fds[i]);
156     exit(0);
```