Πολυτεχνείο Κρήτης - Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Εργαλεία Ανάπτυξης Λογισμικού και Προγραμματισμός Συστημάτων

Χειμερινό Εξάμηνο 2019-2020

Αναφορά 3ης Εργασίας

Σπυριδάκης Χρήστος 2014030022
Ζαχαριουδάκης Νικόλας 2016030073

Παρασκευή 10 Ιανουρίου 2020

1 Εισαγωγή

Στο συγκεκριμένο εργαστήριο χρειάστηκε να υλοποιήσουμε ένα Client-Server σύστημα σε γλώσσα C. Σκοπός αυτού του συστήματος είναι να απαντάει ο Server σε ερωτήματα εκτέλεσης UNIX εντολών που του στέλνει ο Client. Συγκεκριμένα υποστηρίζονται μόνο οι εξής πέντε εντολές: ls, cat, cut, tr και grep με χρήση παραμέτρων σε αυτές, οι οποίες μπορούν να συνδυαστούν μεταξύ τους μέσω pipes. Ο Client διαβάζει αυτές τις εντολές από ένα αρχείο εισόδου και αν στο αρχείο αυτό υπάρχουν σε κάποια σειρά εντολές που διαχωρίζονται με semicolon, εκτελεί μόνο την πρώτη αν αυτή είναι αποδεκτή. Για μη υποστηριζόμενες εντολές, εκτελεί αν υπάρχει μέχρι και αυτήν (στην περίπτωση των pipes) το αποδεκτό κομμάτι της σειράς. Ενώ θα έπρεπε ως έξτρα να υπάρχει υποστήριξη των εντολών end και timeToStop, οι οποίες θα τερματίζουν το παιδί που την έλαβε ή όλα τα processes του Server αντίστοιχα. Η σύνδεση του Client με τον Server για την αποστολή των requests θα έπρεπε να υλοποιηθεί με χρήση TCP σύνδεσης σε socket ενώ για την απάντηση των responses αποστολή datagrams από τον Server στον Client.

Πρώτο βήμα πριν την υλοποίηση ήταν να γίνει μία σχεδίαση του τρόπου λειτουργίας του συστήματος από τα δύο μέλη της ομάδας. Αφού έγινε αυτό ξεκίνησε η υλοποίηση των υποσυστημάτων και τέλος έγινε ο συνδυασμός τους. Με δεδομένο ότι το μέγεθος του κώδικα είναι μαζί με το inline documentation πάνω από 900 γραμμές κώδικα, να σημειωθεί σε αυτό το σημείο ότι στην αναφορά θα υπάρχει μία όχι τόσο αυστηρή επεξήγηση (και πιθανόν κάποια ενδιάμεσα κομμάτια να μην παραθέτονται ή να είναι ελαφρός αλλαγμένα για να εμφανιστούν στην αναφορά), προκειμένου να αναφερθούν τα σημαντικα σημεία αλλά παρόλα αυτά να είναι και σύντομη.

Δεν μπορούν να δοθούν, συγκεκριμένες συναρτήσεις ή source lines of code με τα οποία ασχολήθηκε ο κάθε φοιτητής καθώς και οι δύο, λίγο πολύ έκαναν διορθώσεις ή έδωσαν συμβουλές για τις υλοποιήσεις του άλλου μέλους. Αλλά μπορούμε να πούμε ότι περισσότερη προσοχή έδωσε ο Σπυριδάκης στα κομμάτια σχετικά με το error handling, σύνδεση και αποστολή μέσω TCP, αποστολή datagrams και λήψη μέσω UDP, αποστολή signals στον πατέρα και τερματισμός σε περίπτωση εντολών end/timeToStop καθώς επίσης και το parsing του αποδεκτού μέρους κάθε εντολής. Ενώ ο Ζαχαριουδάκης στην δημιουργία και λειτουργία των πολλαπλών παιδιών στον Server, στο διάβασμα των πακέτων που έρχονται από το TCP connection πολλαπλών Clients, την αποθήκευση των λαμβανόμενων request στο pipe από το πατρικό process, την ανάγνωση πληροφοριών του pipe από τα θυγατρικά process, την εκτέλεση των εντολών από τα θυγατρικά process καθώς και το σπάσιμο των responses σε πακέτα προκειμένου να μπορούν να χωρέσουν στα datagrams.

Σημείωση: Να αναφερθεί ότι κατά την υλοποίηση της πρώτης εργασίας, τα μέλη της ομάδας ήταν δύο ανεξάρτητες ομάδες. ΄΄Υστερα από συνεννόηση με τον κύριο Δεληγιαννάκη, δημιουργήθηκε η συγκεκριμένη ομάδα δύο ατόμων για την παράδοση των εναπομεινάντων σετ ασκήσεων. Κατά την πρώτο σετ ο κωδικός της ομάδας του φοιτητή Σπυριδάκη ήταν LAB21142505 ενώ του φοιτητή Ζαχαριουδάκη LAB21143105, του οποίου ο κωδικός ομάδας κρατήθηκε.

2 Υλοποίηση

Η εξήγηση της υλοποίησης του συγκεκριμένου project θα είναι με την ίδια σειρά με την οποία γίνεται και το dataflow (αναφερόμενοι στον Server κυρίως). Ο λόγος είναι, το να υπάρχει συνοχή και να αναφέρονται όταν χρειάζεται ενδιαφέροντα σημεία. Επίσης να σημειωθεί ότι υπάρχουν 4 σημαντικά αρχεία μέσα στο project tree. Τα remoteClient.c και remoteServer.c τα οποία περιέχουν την λογική του Client και του Server αντίστοιχα, από τα οποία στην συνέχεια δημιουργούνται τα ζητούμενα εκτελέσιμα **remoteClient** και **remoteServer**. Το Makefile το οποίο κάνει compile τα sources. Ενώ από την στιγμή που δεν υπάρχει στην εκφώνηση κάποιο σημείο απαγόρευσης για επιπλέον αρχεία, έχουμε προσθέσει το handling.h το οποίο είναι το header file που κάνουν import τόσο ο server οσο και ο client και περιέχει macros και functions τα οποία χρησιμοποιούνται από τα εκτελέσιμα. Ένα τέτοιο παράδειγμα, που γίνεται κλήση κώδικα του header και από τα δύο εκτελέσιμα έχει να κάνει με το error handling και περιγράφεται παρακάτω. Συνεπώς για λόγους maintenance και consistency δεν μπήκαν ξεχωριστά στο κάθε source file.

2.1 Error handling

Πριν ξεκινήσουμε με την επεξήγηση αυτή καθεαυτή, να αναφερθεί ότι για την υλοποίηση του συγκεκριμένου project πρέπει να υπάρχει ένα εύχρηστο και αξιόπιστο σύστημα για error handling, διότι σε θέματα σχετικά με sockets, pipes, etc είναι πολύ πιθανό να υπάρξει κάποιο πρόβλημα. Πράγμα που μπορούμε να το δούμε ότι ισχύει εύκολα (και μας βοηθάει) στο ότι μοιάζουν λίγο πολύ το τι επιστρέφουν αυτές οι συναρτήσεις σε περίπτωση σφάλματος. Δημιουργήθηκαν λοιπόν τα παρακάτω Function-like Macros τα οποία χρησιμοποιούνται κατα μήκος της εκτέλεσης. Σκοπό έχουν σε περίπτωση σφάλματος να εμφανίζουν στο stderr σχετικό μήνυμα μαζί με έξτρα πληροφορίες. Ο λόγος που χρησιμοποιήθηκαν Function-like Macros είναι διότι ήταν πολύ εύκολο να πάρουμε έξτρα πληροφορίες σχετικά με το αρχείο ή την σειρά στην οποία προέκυψε το αντίστοιχο error. Με παρόμοια λογική υπάρχει και DEBUG messages τα οποία εμφανίζονται επίσης κατά μήκος της εκτέλεσης όταν το DEBUG mode είναι ενεργοποιημένο.

```
#define CHECKNO(X) {if ((X) == (-1)) FATALERR(errno);}
#define CHECKNU(X) {if ((X) == NULL) FATALERR(errno);}
#define CHECKNE(X) {if ((X) < 0) FATALERR(errno);}
```

2.2 Client

Το πρώτο ζητούμενο κατά την εκτέλεση του προγράμματος ήταν να διαβάσουμε και να σιγουρευτούμε ότι ο αριθμός των παραμέτρων εισόδου είναι αυτός που θα έπρεπε. Αφού το κάναμε αυτό δημιουργούμε ένα child process το οποίο θα είναι υπεύθυνο για την λήψη των datagrams και την εγγραφή των απαντήσεων σε αρχεία. Ενώ το parent process θα είναι αυτό που θα διαβάζει και θα στέλνει μέσω TCP τις εντολές προς εκτέλεση.

2.2.1 Client Father

Ο πατέρας ενός Client είναι υπεύθυνος για το διάβασμα του αρχείου, την σύνδεση μέσω TCP με τον Server καθώς και την αποστολή πληροφορίας ανά 5 δευτερόλεπτα. Σημειώνουμε ότι επειδή μας ενδιαφέρουν πολλαπλά πράγματα για τις απαντήσεις των εντολών, στον Server δεν στέλνουμε κάποιο string αλλά ολόκληρο struct στο οποίο περιλαμβάνονται οι αναγκαίες πληροφορίες, όπως π.χ. το line της εντολής.

```
typedef struct address {
  char command[1024];
  char address[1024];
  int port;
  int lineNumber;
} commandPackage;
```

Για την σύνδεση του Client με τον Server είναι σημαντική η μετατροπή του serverName σε πληροφορία που μπορεί να κατανοήσει το **struct sockaddr_in** πεδίο που χρησιμοποιούμε για να κάνουμε το connect του TCP socket, πράγμα που το καταφέρνουμε με την χρήση της **gethostbyname**. Ενώ επίσης δεν πρέπει να ξεχάσουμε να συνδεθούμε στην σωστή port.

Επίσης για το διάβασμα του αρχείου χρησιμοποιούμε την fopen για να ανοίξουμε το input αρχείο και με την βοήθεια της getline() διαβάζουμε κάθε γραμμή μέχρι το τέλος του αρχείου.

Αφού διαβάσουμε μία σειρά δημιουργούμε ένα commandPackage instance που περιέχει ότι χρειαζόμαστε μέσα και το στέλνουμε στον Server. Κάθε 10 πακέτα περιμένουμε για 5 δευτερόλεπτα ενώ αφού έχουμε στείλει όλα τα πακέτα στέλνουμε ένα τελευταίο το οποίο ενημερώνει ότι τελειώσαμε να στέλνουμε.

```
...
if(!(remote_addr=gethostbyname(serverName))){FATAL("ERROR with hostname!")}
CHECKNU(fp = fopen(inputFileWithCommands, "r"));
CHECKNO(sockfd=socket(AF_INET, SOCK_STREAM, 0));
CHECKNO(connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr)));
...
while ((read = getline(&line, &len, fp)) != -1) {
        ...
        CHECKNE(send(sockfd, &cp, sizeof(cp), 0));
        cntMess++;
        if(cntMess%10==0) {sleep(SLEEP);};
}
 ...
CHECKNE(send(sockfd, &cp, sizeof(cp), 0));
close(sockfd); fclose(fp); free(line);
```

2.2.2 Client Child

Το θυγατρικό process του client είναι αυτό το οποίο θα δημιουργήσει ένα UDP listener και θα περιμένει να του έρθουν απαντήσεις τις οποίες τις γράφει στα κατάλληλα αρχεία. Όπως αναφέραμε κατά την αποστολή requests προς τον server στέλνουμε πακέτα τα οποία περιέχουν commandPackage πληροφορία, με παρόμοια λογική κατά το response τα datagram περιέχουν επίσης struct (διαφορετικό όμως) στο οποίο υπάρχει το line of code για το οποίο είναι το response καθώς επίσης και το response που λάβαμε. Με αυτόν τον τρόπο, και αφού ξέρουμε ο συγκεκριμένος listener σε ποια port ακούει, ξέρουμε το όνομα του αρχείου στο οποίο θα κάνουμε append το datagram που λάβαμε. Κάποια ενδιαφέροντα σημεία είναι τα εξής: Έπρεπε να προσέξουμε ότι το struct που στέλνει ο Server έχει το κατάλληλο μέγεθος για να χωρέσει σε datagram. Επίσης έπρεπε να βρούμε με ποιον τρόπο θα τερματίζει την λειτουργία του ο Client. Υλοποιήσαμε το receiver του Client να μπορεί να τερματίσει με δύο τρόπους. Αρχικά δημιουργήθηκε ένα alarm signal για timeout το οποίο ουσιαστικά μπορεί να παρομοιαστεί με την έννοια του watchdog, αυτό που δηλαδή κάνει είναι να βλέπει αν η στιγμή που πήρε το τελευταίο response είναι πολύ μεγάλη, συνεπώς δεν έχει επικοινωνία πλέον με τον server και σε αυτήν την περίπτωση τερματίζει (χρήσιμο π.χ. αν τρέξει από άλλον client το timeToStop). Άλλος ένας τρόπος είναι όταν λάβει σαν response το EOC, το οποίο ο Server το στέλνει μετά από ένα χρονικό διάστημα αναμονής αφού έχει λάβει και την τελευταία εντολή κάποιο παιδί (το οποίο στέλνει σε πολλαπλά αντίγραφα για την περίπτωση που χαθεί ένα ή περισσότερα από τα datagrams).

```
signal(SIGALRM, alarm_handler);
...
CHECKNE(sockfd=socket(AF_INET, SOCK_DGRAM, 0));
CHECKNE(setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const char*) &reuse, sizeof(reuse)));
CHECKNE(ret = bind(sockfd, (struct sockaddr *)&receive_addr, sizeof(struct sockaddr)));
LAST_RECEIVED=gettime();
alarm(ALARM_TIME);
while(TRUE){
        ...
        ret = recvfrom(sockfd, &rp, sizeof(rp), 0, (struct sockaddr *)&server, &serv_len);
        ...
        LAST_RECEIVED=gettime();
        ...
        if(strcmp(res, "EOC") == 0){break;}
        writeToFile(process, receivePort, rp.lineNumber, res);
}
close(sockfd);
```

Μία τελευταία σημείωση είναι η εξής: Όπως θα παρουσιαστεί και παρακάτω ο Client κάνει append στο κάθε αρχείο το response μίας εντολής, με την σειρά που έρχονται για αυτην την εντολή τα datagrams. Με την λογική ότι ένα child **και μόνο** στέλνει το response ενός command, θα πρέπει και τα datagrams για το αρχείο αναφοράς να έρθουν με την σωστή σειρά (**δεν μας ενδιαφέρει αν για διαφορετικά αρχεία τα datagrams είναι με άλλη σειρά, καθώς με τον τρόπο υλοποίησης δεν υπάρχει πρόβλημα σε αυτό**) ο λόγος που αναφέρεται είναι διότι αν θέλαμε για N εντολές εκτέλεσης με M πακέτα η κάθε μία να ελέγχουμε αν το πακέτο που λάβαμε πρέπει να γραφτεί τώρα ή όχι η πολυπλοκότητα αυξάνεται σημαντικά. Το UDP δεν προσφέρει αξιοπιστία και αφού χρειάζεται να χρησιμοποιήσουμε αυτό το πρωτόκολλο απλά βεβαιωνόμαστε ότι γράφεται στο κατάλληλο αρχείο το κάθε response.

2.3 Server

Τα πρώτα βήματα μας, για τον server, ήταν να δημιουργία του process του πατέρα και των παιδίων που μας ζητείται απο εκείνον μέσω της fork. Επειδή με την fork γίνεται ένα είδος κλονοποίησης του κώδικα που ακολουθεί πριν το fork έχουμε την δημιουργία του pipe επικοινωνίας και μετά απο αυτό έχουμε την διαδικασία που θα ακολουθήσει κάθε process αναλόγως της κατάστασής του (child-parent)

```
void runServer(int numChildren, int portNumber){
    int mstsockfd, pip[2], i, reuse = 1;
    int FATHER = getpid();
    ...
    // PIPE creation
    CHECKNO(pipe(pip));
     // PRE-FORKING THE SERVER TO GIVEN NUYMBER OF CHILDREN
    for (i = 0; i < numChildren; i++){
        CHECKNE(getpid());
        if (getpid() == FATHER){
            CHECKNE(childpid = fork());
            if (childpid > 0){
                CPID[i] = childpid;
            }

        }
    }
    if (getpid() == FATHER){
        // CREATING AND INITIALIZING THE SERVER (CREATE SOCKET, BIND, LISTEN)
        CHECKNE(mstsockfd = socket(AF_INET, SOCK_STREAM, 0));
        CHECKNE(setsockopt(mstsockfd, SOL_SOCKET,
                SO_REUSEADDR, (const char *)&reuse, sizeof(reuse)));
        CHECKNE(bind(mstsockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)));
        CHECKNE(listen(mstsockfd, SERVER_BACKLOG));
        while (TRUE){
            runParent(activefds, readfds, pip, mstsockfd, client_addr);
        }
    }
    else if (childpid == 0){
        runChild(pip);
    }
```

2.3.1 Server Father

Στην περίπτωση του πατέρα έχουμε δύο συναρτήσεις, τις **runParent() readFromConnection()** με τις εξείς λειτουργίες. Η **runParent()** είναι υπεύθυνη να ελέγχει για καινούργια δραστηριότητα μέσω της select. Ακολούθως μόλις λάβει σήμα, το οποίο είναι θετικό, σημαίνει ότι έχουμε καινούργιο αίτημα σύνδεσης και πρέπει να ελέγξουμε αν είναι ένα απο τα υπάρχοντα που εξυπηρετούνται ή αν είναι νέα σύνδεση. Στην πρώτη περίπτωση γίνεται κλήση της **readFromConnection()** ενώ για νέες συνδέσεις καλούμε την **accept()**. Μετά την κλήση της **readFromConnection()**, αν επιστραφεί -1, τότε τελείωσε η λήψη και πρέπει να κλείσουμε τον client.

```
void runParent( fd_set activefds,fd_set readfds , int pip[2], int mstsockfd, struct
    sockaddr_in client_addr){

    //wait for an activity on one of the sockets , timeout is NULL
    activity = select(FD_SETSIZE, &readfds, NULL, NULL, NULL);
    CHECKNE(activity);
    for (int i = 0; i < FD_SETSIZE; i++){
        if (FD_ISSET(i, &readfds)){
            if (i==mstsockfd){
                addr_size = sizeof(struct sockaddr_in);
                CHECKNO(client_socket = accept(mstsockfd,
                        (struct sockaddr *)&client_addr, (socklen_t *)&addr_size));
                FD_SET(client_socket, &activefds); //put client in active client list
            }
        }
        else if (FD_ISSET(i, &activefds))
        {
            if (readFromConnection(i, pip, client_addr) <0) {
                close(i);
                FD_CLR(i, &activefds);
            }
        }
    }
}
```

Η **readFromConnection()** είναι υπεύθυνη για την λήψη απο τον client. Αυτό γίνεται μέσα απο την συνάρτηση **recv()** το αποτέλεσμα της οποίας είναι ένα commandPackage (έχει γίνει αναφορά στην περιγραφή του client). Στην συνέχεια χρησιμοποιούμε όμως το handlePackage hp, διότι χρειαζόμαστε και την διεύθυνση του client που θα στείλουμε πίσω την απάντησημ μας. Τέλος εκείνη είναι αυτή που γράφει στο pipe που θα βλέπουν τα παιδιά για να εξυπηρετήσουν και να εκτελέσουν εντολές.-

```
typedef struct handler {
   commandPackage cp;
   struct sockaddr_in client_addr;
} handlePackage;
```

```

int readFromConnection(int client_socket,int pip[2], struct sockaddr_in client_addr){
    while (TRUE){
        size_t bytes_read;
        commandPackage cp;
        handlePackage hp;
        CHECKNO(bytes_read = recv(client_socket, &cp, sizeof(commandPackage), 0));
        if (strcmp(cp.command, "EOF") == 0){
            ...
            // Write the handlepackage
            ret=write(pip[WRITE], &hp, sizeof(handlePackage));
            return -1;
        }
        else if (memcmp (&cp, &lastPackage, sizeof(commandPackage)) == 0) {
            return -1;
        }
        ...
        // Write the handlepackage
        ret=write(pip[WRITE], &hp, sizeof(handlePackage));
    }
}
```

2.3.2 Server Child

Συνεχίζοντας στην περίπτωση που είμαστε σε ένα child process τότε καλούμε, όπως φαίνεται και παραπάνω, την **runChild(int pip[2])** η οποία διαβάζει την πρώτη διαθέσιμη εντολή απο το pipe και καλεί την **commandExecu-**

tion(rechp) και εν συνεχεία ελέγχει την τιμή επιστροφής της έτσι ώστε αν έχουμε end ή timeToStop να στείλει τα κατάλληλα σήματα στον πατέρα.

```
void runChild(int pip[2]){
    commandPackage reccp;
    handlePackage rechp;
    int term, ret;
    while (TRUE){
        ret=read(pip[READ], &rechp, sizeof(handlePackage));
        reccp = rechp.cp;
        term = commandExecution(rechp);
        if (term == 0){
            kill(getppid(),SIGEND);
            kill(getpid(),SIGINT);
            exit(EXIT_SUCCESS);
        }
        else if (term == -1){
            kill(getppid(), SIGTTS);
        }
    }
}
```

Όπως φαίνεται στον παρακάτω κώδικα η **commandExecution(handlePackage hp)** εκτελεί την δεδομένη εντολή με την συνάρτηση **popen()**. Αρχικά στην ***command*** παίρνουμε την εντολή μας απο τον parser που έχουμε υλοποιήσει και αν αυτή είναι end ή timeToStop τότε στέλνουμε τιμή επιστροφής 0 και -1 αντίστοιχα. Σε διαφορετική περίπτωση εκτελούμε την εντολή μας και παίρνουμε την απάντησή μας σε μικρά πακετάκια των 512 και καλούμε για κάθε πακέτο την response() που στέλνει με udp την απάντηση πίσω στον client.

```
int commandExecution(handlePackage hp)
{
    //GIve the line to parser and take the command via "command"
    int comRetVal = commToExecute(hp.cp.command, command);
    if (strcmp(command, "end") == 0){
        return 0;
    }
    else if (strcmp(command, "timeToStop") == 0){
        return -1;
    }
    else{
        char execCommand[BUFSIZE] = "";
        CHECKNU(fp = popen(execCommand, "r"));

        int buffSize = TEXT_RESPONSE_LEN;
        char *newBuffer = (char *)malloc(buffSize);
        /* Read the output a line at a time - output it. */
        while (fgets(path, sizeof(path), fp) != NULL){
            if (strlen(newBuffer) + strlen(path) < buffSize){
                strcat(newBuffer, path);
            }
            else{
                response(hp,newBuffer);
                newBuffer = (char *)malloc(buffSize);
                strcpy(newBuffer, path);
            }
        }
        response(hp,newBuffer);
        sleep(1);
        pclose(fp);
        return 1;
    }
    // return 1;
}
```

2.4 Parser

Προκειμένου να μπορούμε να απομονώσουμε από ένα line μόνο το κομμάτι της εντολής το οποίο μπορεί να εκτελεστεί (αν υπάρχει μέσα σε αυτό) δημιουργήσαμε τον δικό μας parser ο οποίος σκοπό έχει να βλέπει χαρακτήρα χαρακτήρα ένα line προκειμένου να καταλάβει τι ακριβώς διαβάζει και να επιστρέψει το κατάλληλο command εκτέλεσης. Μία γρήγορη τακτική για την απομόνωση της αποδεκτής εντολής θα ήταν να παίρνουμε απλά με βάση κάποιον delimiter π.χ. το ; το κομμάτι του line που μας ενδιαφέρει και χρήση της strok(). Κάτι που δεν είναι όμως σωστό καθώς το semicolon μπορεί να εμφανίζεται μέσα σε κάποιο string και να μην υποδηλώνει την εναλλαγή των εντολών (π.χ. ls /tmp/ | tr ";" '|'). Για την επίτευξη αυτού του parser λοιπόν δημιουργήσαμε επιπλέον τις εξής συναρτήσεις.

```
bool acceptableCommand(const char* command); // #define bool int
int findNextCommand(int *start, const char* str);
int parseStr(int *start, const char* str);
int parseLine(const char* str);
int commToExecute(const char * line, char *command);
```

Ουσιαστικά για κάθε γραμμή βρίσκουμε το δείκτη του χαρακτήρα μέχρι την οποία είναι αποδεκτή μία εντολή, με την χρήση της parseLine() και με βάση αυτόν τον δείκτη απομονώνουμε από το line σε ένα string μόνο το acceptable μέρος με την χρήση της commToExecute(). Συγκεκριμένα για κάθε σειρά, μέχρι το τέλος του line όταν έχουμε καινούργια εντολή (στην αρχή της σειράς ή μετά από pipe) βρίσκουμε την πρώτη λέξη με την χρήση της findNextCommand() και ελέγχουμε αν αυτή η λέξη είναι αποδεκτή εντολή με την χρήση της acceptableCommand(), που δεν κάνει τίποτα άλλο από το να βλέπει αν είναι ίδια με ένα από τα πεδία του predefined array με τα acceptable commands. Σε περίπτωση που δεν είναι επιστρέφουμε ένα δείκτη που δείχνει ακριβώς πριν το pipe ή το 0 αν η πρώτη εντολή μιας σειράς δεν είναι δεκτή. Αλλιώς για όλο το μήκος της συγκεκριμένης υπο-εντολής μετακινούμε τον δείκτη μέχρι να εμφανιστεί pipe ή semicolon (το οποίο δεν είναι σε string). Κάθε φορά που εμφανίζονται quotes βρίσκουμε που κλείνουν (αν κλείνουν αλλιώς παίρνουμε όλο το line) και συνεχίζουμε από το σημείο που έκλεισαν. Αν εμφανιστεί semicolon επιστρέφουμε ως δείκτη το σημείο ακριβώς πριν από αυτό. Δουλειά του συγκεκριμένου parser είναι να απομονώνει μόνο μέχρι το σημείο που μία εντολή πληροί τα requirements δεν μπορεί να ελέγξει αν είναι συντακτικά σωστή (π.χ. ls /dev/urandom/, δεν ειναι dir αλλά αρχείο). Σε αυτήν την περίπτωση όπως και σε εντολές που δεν είναι αποδεκτές έχουμε κενά αρχεία ως response στον Client. Σε περίπτωση που ο parser δει ότι σε μία σειρά υπάρχουν οι εντολές end ή timeToStop επιστρέφει συγκεκριμένες τιμές εξόδου προκειμένου το παιδί να το γνωρίζει και σε αντίθεση με την εκτέλεση της εντολής του parser να ακολουθεί ανάλογη διαδικασία τερματισμού.

2.5 Signals

Αρχικά δημιουργήσαμε handler ο οποιος θα διαχειρίζεται το SIGINT signal και να αναφερθεί ότι κάθε remoteServer process το χρησιμοποιεί. Σκοπός αυτού του handler είναι να κάνει display ότι πρόκειται να πεθάνει μαζί με το PID του και μετά να πεθαίνει. Για την υλοποίηση της λειτουργίας που ζητείται στα signal χρησιμοποιούμε ουσιαστικά τα διαθέσιμα USER signals.

```
#define SIGEND SIGUSR1
#define SIGTTS SIGUSR2
```

Ενώ έχουμε στον parent δύο handlers για το καθένα από αυτά.

```
void runParent( fd_set activefds,fd_set readfds , int pip[2], int mstsockfd, struct
    sockaddr_in client_addr){
   signal(SIGEND, endChild);
   signal(SIGTTS, timeToStop);
...
```

΄Οταν έρθει end, το process το οποίο το έλαβε στέλνει στον πατέρα το SIGEND signal όπου ο πατέρας ενημερώνεται ότι ένα από τα παιδιά του πέθανε και ανανεώνει το πλήθος των ζωντανών παιδιών του. Στην συνέχεια το συγκεκριμένο παιδί στέλνει SIGINT στον εαυτό του προκειμένου να πεθάνει.

Παρόμοια διαδικασία γίνεται και στο timeToStop. Σε αυτήν την περίπτωση μόλις λάβει ο πατέρας το συγκεκριμένο signal στέλνει SIGINT σήμα στα παιδιά του προκειμένου να τα σκοτώσει και στην συνέχεια στέλνει SIGINT στον εαυτό του. Παρακάτω παρατίθεται ο κώδικας που μόλις αναφερθήκαμε.

```
void endChild(){
    signal(SIGEND, endChild);
    ALIVE_KIDS--;
    if (!alive())
        kill(getpid(), SIGINT);
}

void timeToStop(){
    int i;
    char mess[200];
    for(i=0;i<KIDS_NUM; i++)
        kill(CPID[i], SIGINT);

    kill(getpid(), SIGINT);
    ALIVE_KIDS=0;
}
```

3 Testing

Για το testing θα δοθούν δύο σενάρια εκτέλεσεις στο οποίο παρουσιάζονται η βασική λειτουργικότητα.

3.1 Simple 1

Στο πρώτο σενάριο αυτά που θέλουμε να δούμε ότι δουλεύουν σωστά είναι τα εξής. Αρχικά ότι δημιουργούνται επιτυχώς και τρέχουν τα εκτελέσιμα, επίσης ότι μπορεί να συνδεθεί ο Client στον Server μέσω TCP και ότι ο Server μπορεί να δημιουργήσει παιδιά, να κατανείμει το φόρτο, να εκτελέσει τις αποδεκτές εντολές να απαντήσει μέσω datagrams και ο Client να τα γράψει σε αρχεία. Ουσιαστικά όλη η βασική λειτουργικότητα. Το αρχείο εισόδου έχει δημιουργηθεί έτσι ώστε να δούμε ότι χρησιμοποιεί μόνο τις 5 acceptable εντολές. Οι άρτιες σειρές είναι από εντολές που δεν είναι αποδεκτές ενώ στις περιττές προσπαθούμε να δημιουργήσουμε βήμα βήμα την παρακάτω εντολή **ls -al /tmp/ | grep "-" | tr -s " " | cut -d" " -f1 | cat** έχοντας με κάποιες από αυτές περιττά pipes ή επιπρόσθετες εντολές για να επαληθεύσουμε ότι θα εκτελεστεί μέχρι εκεί που πρέπει το αποτέλεσμα.

```
ls -al /tmp/
echo "Hello World"
ls -al /tmp/ | grep "-"
figlet remoteServer
ls -al /tmp/ | grep "-" | tr -s " " ; ls /
seq 1 1 10 | cat
ls -al /tmp/ | grep "-" | tr -s " " | cut -d" " -f1  | echo "One more wrong command"
nano /dev/urandom
ls -al /tmp/ | grep "-" | tr -s " " | cut -d" " -f1 | cat
```

Όπως αναφέρθηκε και παραπάνω έχουμε δημιουργήσει DEBUG messages θα δούμε λοιπόν με την χρήση αυτών την ομαλή λειτουργία του Server και του Client. Ξεκινώντας στο παρακάτω figure μπορούμε να δούμε την δημιουργείτων παιδιών στον Server καθώς και του TCP socket, ότι στην συνέχεια συνδέεται κάποιος Client όπως επίσης και ότι λαμβάνει μηνύματα.

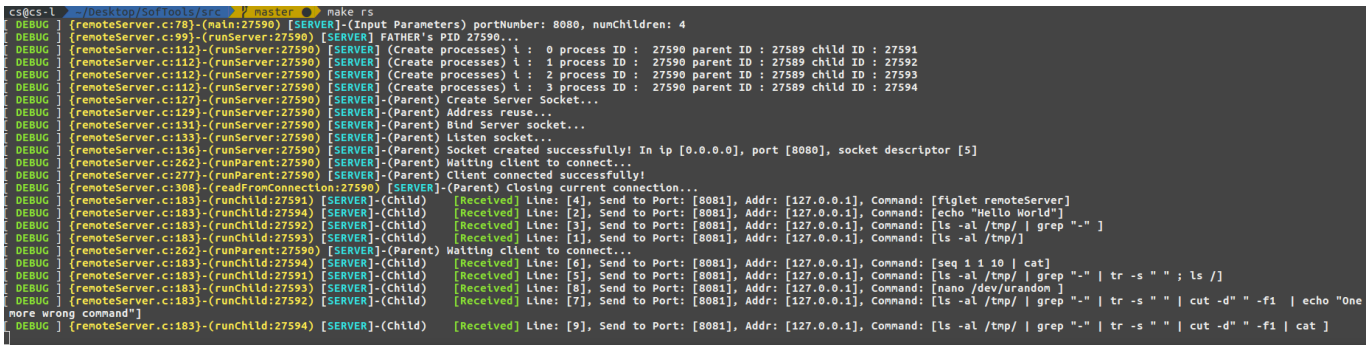

Figure 1: `Server's debug messages`

Στο παρακάτω figure μπορούμε να δούμε ότι ο Client συνδέεται στον Server, ότι στέλνει τα πακέτα όπως επίσης παίρνει και ότι λαμβάνει τα πρώτα responses.

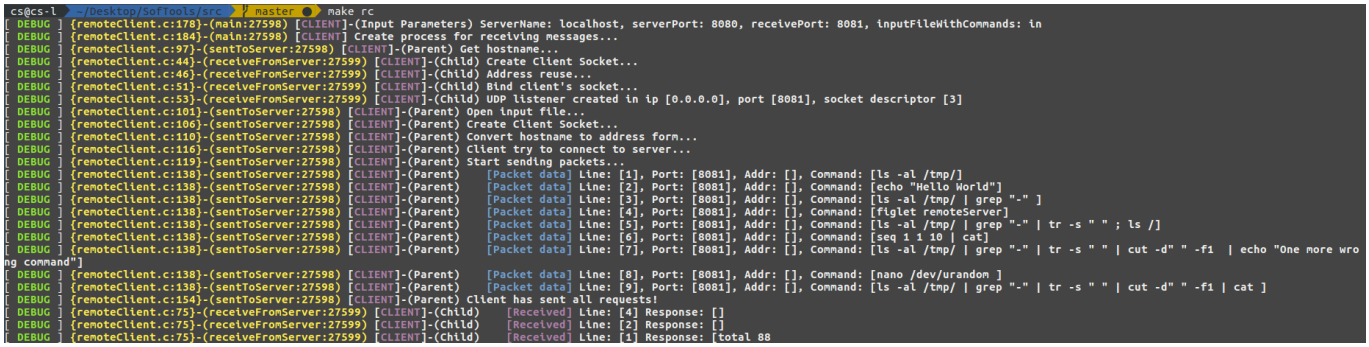

Figure 2: `Client's debug messages`

Σε αυτό το figure μπορούμε να δούμε τα αρχεία τα οποία έχει λάβει ο client, να επαληθεύσουμε ότι όλα τα ζυγά αρχεία είναι άδεια όπως και θα έπρεπε, ότι το αποτέλεσμα της πιο σύνθετης εντολής είναι αυτό που περιμένουμε όπως επίσης (επειδή τρέχουν στον ίδιο μηχάνημα) ότι στον server τρέχουν κανονικά τα παιδιά καθώς και ο πατέρας.

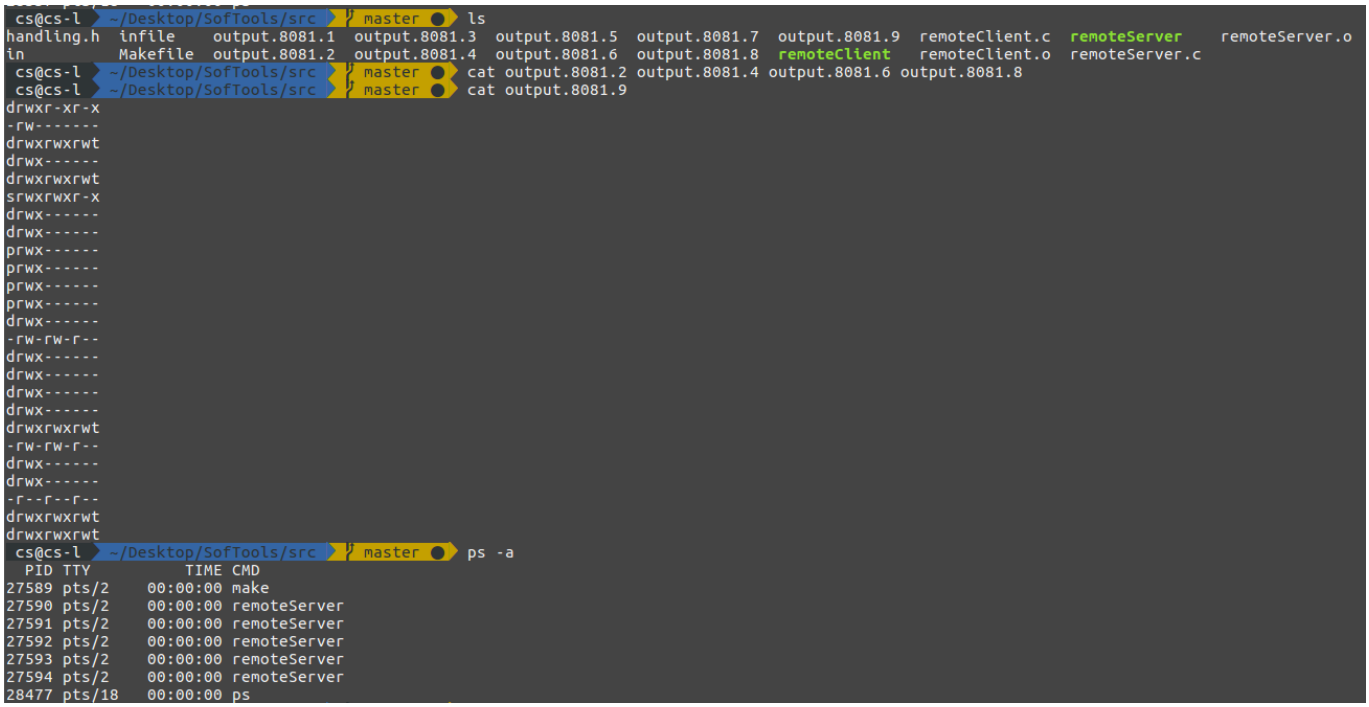

Figure 3: `Some usefull info`

3.2 Simple 2

Στο δεύτερο παράδειγμα θέλαμε να δούμε ότι λειτουργούν όπως θα έπρεπε το end, το timeToStop καθώς και κάποια θέματα σχετικά με την ύπαρξη semicolon και pipe μέσα σε strings (ουσιαστικά εκτελούμε την παραπάνω εντολή έχοντας κάνει στο δεύτερο στάδιο αντικατάστατη των ; με |).

```
end
ls -al /tmp/ | tr ";" "|" | grep "-" | tr -s " " | cut -d" " -f1 | cat
ls -al /tmp/ | tr ";" "|" | grep "-" | tr -s " " | cut -d" " -f1 | cat
ls -al /tmp/ | tr ";" "|" | grep "-" | tr -s " " | cut -d" " -f1 | cat
ls -al /tmp/ | tr ";" "|" | grep "-" | tr -s " " | cut -d" " -f1 | cat
ls -al /tmp/ | tr ";" "|" | grep "-" | tr -s " " | cut -d" " -f1 | cat
timeToStop
```

Αυτό που είναι σημαντικό να δούμε παρακάτω είναι ότι στον server γίνεται kill το ένα process στην συνέχεια γίνονται κάποιες εκτελέσεις ενώ όταν έρθει timeToStop πεθαίνουν όλα τα παιδιά μαζί με τον Parent process.

```
cs@cs-l ~/Desktop/SofTools/src master ● make rs
[ DEBUG ] {remoteServer.c:78}-(main:29864) [SERVER]-(Input Parameters) portNumber: 8080, numChildren: 4
[ DEBUG ] {remoteServer.c:99}-(runServer:29864) [SERVER] FATHER's PID 29864...
[ DEBUG ] {remoteServer.c:112}-(runServer:29864) [SERVER] (Create processes) i :  0 process ID :  29864 parent ID : 29863 child ID : 29865
[ DEBUG ] {remoteServer.c:112}-(runServer:29864) [SERVER] (Create processes) i :  1 process ID :  29864 parent ID : 29863 child ID : 29866
[ DEBUG ] {remoteServer.c:112}-(runServer:29864) [SERVER] (Create processes) i :  2 process ID :  29864 parent ID : 29863 child ID : 29867
[ DEBUG ] {remoteServer.c:112}-(runServer:29864) [SERVER] (Create processes) i :  3 process ID :  29864 parent ID : 29863 child ID : 29868
[ DEBUG ] {remoteServer.c:127}-(runServer:29864) [SERVER]-(Parent) Create Server Socket...
[ DEBUG ] {remoteServer.c:129}-(runServer:29864) [SERVER]-(Parent) Address reuse...
[ DEBUG ] {remoteServer.c:131}-(runServer:29864) [SERVER]-(Parent) Bind Server socket...
[ DEBUG ] {remoteServer.c:133}-(runServer:29864) [SERVER]-(Parent) Listen socket...
[ DEBUG ] {remoteServer.c:136}-(runServer:29864) [SERVER]-(Parent) Socket created successfully! In ip [0.0.0.0], port [8080], socket descriptor [5]
[ DEBUG ] {remoteServer.c:262}-(runParent:29864) [SERVER]-(Parent) Waiting client to connect...
[ DEBUG ] {remoteServer.c:277}-(runParent:29864) [SERVER]-(Parent) Client connected successfully!
[ DEBUG ] {remoteServer.c:183}-(runChild:29867) [SERVER]-(Child)    [Received] Line: [1], Send to Port: [8081], Addr: [127.0.0.1], Command: [end]
[ KILL ] {remoteServer.c:56}-(suicide:29867) [KILL] Process with PID: [29867] suicide.
[ DEBUG ] {remoteServer.c:308}-(readFromConnection:29864) [SERVER]-(Parent) Closing current connection...
[ DEBUG ] {remoteServer.c:262}-(runParent:29864) [SERVER]-(Parent) Waiting client to connect...
[ DEBUG ] {remoteServer.c:183}-(runChild:29868) [SERVER]-(Child)    [Received] Line: [2], Send to Port: [8081], Addr: [127.0.0.1], Command: [ls -al /tmp/ | tr ";" "|" | grep "-" | tr -s " " | cut -d" " -f1
| cat]
[ DEBUG ] {remoteServer.c:183}-(runChild:29866) [SERVER]-(Child)    [Received] Line: [4], Send to Port: [8081], Addr: [127.0.0.1], Command: [ls -al /tmp/ | tr ";" "|" | grep "-" | tr -s " " | cut -d" " -f1
| cat]
[ DEBUG ] {remoteServer.c:183}-(runChild:29865) [SERVER]-(Child)    [Received] Line: [3], Send to Port: [8081], Addr: [127.0.0.1], Command: [ls -al /tmp/ | tr ";" "|" | grep "-" | tr -s " " | cut -d" " -f1
| cat]
[ DEBUG ] {remoteServer.c:183}-(runChild:29865) [SERVER]-(Child)    [Received] Line: [7], Send to Port: [8081], Addr: [127.0.0.1], Command: [timeToStop]
[ KILL ] {remoteServer.c:56}-(suicide:29864) [KILL] Process with PID: [29864] suicide.
[ KILL ] {remoteServer.c:56}-(suicide:29868) [KILL] Process with PID: [29868] suicide.
[ KILL ] {remoteServer.c:56}-(suicide:29866) [KILL] Process with PID: [29866] suicide.
[ KILL ] {remoteServer.c:56}-(suicide:29865) [KILL] Process with PID: [29865] suicide.
```

Figure 4: `Server's debug`

Παρακάτω μπορούμε να δούμε ότι το αρχείο περιέχει αυτά που περιμέναμε άρα έχει εκτελέσει την εντολή όπως θα έπρεπε ενώ μπορούμε να δούμε ότι δεν υπάρχει ενεργό κάποιο process του Server.

```
cs@cs-l ~/Desktop/SofTools/src master ● cat output.8081.2
drwxr-xr-x
-rw-------
drwxrwxrwt
drwx------
drwxrwxrwt
srwxrwxr-x
drwx------
drwx------
prwx------
prwx------
prwx------
prwx------
drwx------
-rw-rw-r--
drwx------
drwx------
drwx------
drwx------
drwxrwxrwt
-rw-rw-r--
drwx------
drwx------
-r--r--r--
drwxrwxrwt
drwxrwxrwt
cs@cs-l ~/Desktop/SofTools/src master ● ps -a
  PID TTY          TIME CMD
30846 pts/2    00:00:00 ps
```

Figure 5: `Some usefull info`