ΕΡΓΑΣΤΗΡΙΟ 8

Υποδοχείς

Να πληκτρολογηθεί και να εκτελεστεί ο κώδικας των επόμενων παραδειγμάτων που επιδεικνύουν τη χρήση των σημάτων ως μέσο επικοινωνία μεταξύ διεργασιών.

Παράδειγμα 1. Σε αυτό το παράδειγμα αρχιτεκτονικής client - server με UNIX sockets, ο server λειτουργεί επαναληπτικά και επιτρέπει τη σύνδεση ενός πελάτη κάθε φορά. Ο πελάτης συνδέεται στο socket του server και αντιγράψει απλά την είσοδό του στον server ο οποίος και την εκτυπώνει.

(αρχεία un-client.c και un-server.c).

Κώδικας διακομιστή

Αρχείο un-server.c

```
// Source: https://www.danlj.org/mkj/lad/src/userver.c.html
// (Johnson & Troan, pp.298-299)

#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>
```

Σε αυτή την απλή εφαρμογή client – server, ο client χρησιμοποιώντας ένα UNIX Socket συνδέεται στον server και του αποστέλλει μηνύματα τα οποία εκτυπώνονται στην οθόνη του server. Η μεταφορά των δεδομένων γίνεται με τη συνάρτηση CopyData.

Η συνάρτηση CopyData αντιγράφει αρχεία από το socket from στο socket to μέσω μίας επαναληπτικής διαδικασίας κατά τη διάρκεια της οποίας διαβάζει ομάδες 1024 bytes κάθε φορά με τη συνάρτηση read από το άκρο from και τα γράφει με τη συνάρτηση write στο άκρο to. Ο επαναληπτικός βρόχος εκτελείται για όσο χρονικό διάστημα υπάρχουν δεδομένα για ανάγνωση και ως εκ τούτου η συνάντηση τερματίζεται όταν δεν υπάρχει τίποτε άλλο για να μεταφερθεί.

```
void copyData(int from, int to) {
   char buf[1024];
   int amount;
   while ((amount = read(from, buf, sizeof(buf))) > 0) {
     if (write(to, buf, amount) != amount) {
        perror("Message from write:");
        exit(1); }}
   if (amount < 0) {
      perror("Message from read:");
      exit(1); } }
```

Η διεργασία του διακομιστή αναμένει αιτήματα εισερχομένων συνδέσεων στο αρχείο ./sample-socket που υλοποιεί ένα Unix Socket (θυμηθείτε πως σε αυτόν τον τύπο δικτυακών υποδοχέων, ως διεύθυνση νοείται η διαδρομή στο δέντρο καταλόγων του εν λόγω αρχείου). Από τη στιγμή που λάβει χώρα η αποκατάσταση μιας σύνδεσης με ένα πελάτη, ο διακομιστής αντιγράφει δεδομένα από το socket στην προεπιλεγμένη έξοδο (ο περιγραφέας αρχείου της οποίας είναι ίσος με 1) μέχρι να λάβει χώρα τερματισμός της σύνδεσης από τον πελάτη. Από τη στιγμή που αυτή η σύνδεση τερματιστεί, ο διακομιστής δεν τερματίζει αλλά περιμένει τη σύνδεση ενός νέου πελάτη. Ωστόσο, σε κάθε περίπτωση μόνο ένας πελάτης μπορεί να συνδεθεί κάθε φορά.

```
int main(void) {
   struct sockaddr_un address;
   int sock, conn;
   socklen_t addrLength;
```

Σε αυτό το σημείο δημιουργείται το passive socket τύπου UNIX (*PF_UNIX*) για επικοινωνία με σύνδεση (*SOCK_STREAM*) στο οποίο ο διακομιστής αναμένει συνδέσεις πελατών.

```
   if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
       perror("Message from socket:");
       exit(1); }
```

Εάν το αρχείο του socket υπάρχει ήδη στο δίσκο αυτό διαγράφεται.

```
   unlink("./sample-socket");
```

Αρχικοποιούνται τα πεδία της δομής sockaddr_un (για Unix Sockets) και στη συνέχεια αυτή η δομή διαβιβάζεται ως όρισμα στην bind η οποία συσχετίζει την τοπική διεύθυνση addr (που έχει αρχικοποιηθεί προηγουμένως) με την ακέραια μεταβλητή sock που έχει επιστραφεί από την κλήση συστήματος socket.

```
   address.sun_family = AF_UNIX;          /* Unix domain socket */
   strcpy(address.sun_path, "./sample-socket");
   addrLength = sizeof(address.sun_family) + strlen(address.sun_path);
```

Καλείται η συνάρτηση bind

```
   if (bind(sock, (struct sockaddr *) &address, addrLength)){
      perror("Message from bind:");
      exit(1); }
```

Καλείται η συνάρτηση listen μέσω της οποίας ο διακομιστής αναμένει αιτήματα σύνδεσης πελάτη. Το μήκος της ουράς στην οποία τοποθετούνται τα εκκρεμή αιτήματα σύνδεσης ορίζεται ίσο με 5.

```
  if (listen(sock, 5)){
      perror("Message from listen:");
      exit(1); }
```

Μέσω μίας επαναληπτικής διαδικασίας η οποία τερματίζεται μόνο όταν σταματήσει η λειτουργία του ίδιου του διακομιστή, καλείται συνεχώς η συνάρτηση accept για να διαβάσει δεδομένα από τον πελάτη. Η accept δημιουργεί ένα νέο active socket με ονομα conn που συνομιλεί με το active socket του πελάτη (το socket που δημιουργεί η accept είναι διαφορετικό από το αρχικό passive socket που χρησιμοποιήθηκε ως όρισμα στη listen) και μέσω του οποίου ο πελάτης αποστέλλει τα δεδομένα του στον διακομιστή που τον εξυπηρετεί. Η αποστολή αυτών των δεδομένων και η παραλαβή τους από το διακομιστή, γίνεται καλώντας τη συνάρτηση CopyData (conn, 1) η οποία μεταφέρει δεδομένα από το socket conn στην προεπιλεγμένη έξοδο του διακομιστή που έχει file descriptor με τιμή ίση με 1. Μετά τον τερματισμό της λειτουργίας του πελάτη, ο διακομιστής αναμένει ενεργός αναμένοντας αιτήματα σύνδεσης ενός νέου πελάτη.

```
   while ((conn = accept(sock, (struct sockaddr *) &address, &addrLength)) >= 0) {
      printf("---- getting data\n");
      copyData(conn, 1);
      printf("---- done\n");
      close(conn); }
```

Εάν η κλήση της accept δεν είναι επιτυχής, η εφαρμοφή τερματίζει τη λειτουργία της με μήνυμα λάθους.

```
   if (conn < 0) {
      perror("Message from accept:"); exit(1); }
   close(sock);
```

Κώδικας πελάτη

Αρχείο un-client.c

```
// Source: https://www.danlj.org/mkj/lad/src/userver.c.html
// (Johnson & Troan, pp.298-299)

#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>

// Copies data from file descriptor 'from' to file descriptor 'to until nothing is left to be
// copied. Exits if an error occurs.

void copyData(int from, int to) {
   char buf[1024];
   int amount;
   while ((amount = read(from, buf, sizeof(buf))) > 0) {
     if (write(to, buf, amount) != amount) {
         perror("Message from write:"); exit(1); }}

   if (amount < 0) {
      perror("Message from read:");
      exit(1); } }
```

Ο κώδικας του πελάτη μοιάζει πάρα πολύ με τον κώδικα του διακομιστή και αυτό φυσικά είναι αναμενόμενο διότι αυτές οι δύο διεργασίες θα αρχικοποιηθούν με τον ίδιο ακριβώς τρόπο. Η συνάρτηση CopyData προφανώς θα πρέπει να αντιγραφεί και στον πηγαίο κώδικα του πελάτη (αφού ο διακομιστής και ο πελάτης αποτελούν δύο ανεξάρτητες εφαρμογές με το δικό της κώδικα η καθεμία και ως εκ τούτου η εν λόγω συνάντηση πρέπει να δηλωθεί σε αμφότερες τις εφαρμογές), ενώ προκειμένου να είναι δυνατή η επικοινωνία ανάμεσα στις δύο εφαρμογές, η συνάρτηση socket θα πρεπε σε αμφότερες να κληθεί με τον ίδιο ακριβώς τρόπο. Η δομή *sockaddr_un αρχικοποιείται και αυτή με τον ίδιο ακριβώς τρόπο, με τη διαφορά πως στην εφαρμογή του διακομιστή διαβιβάζεται ως όρισμα στη συνάρτηση bind, ενώ στην εφαρμογή του πελάτη διαβιβάζεται ως όρισμα στη συνάρτηση connect. Μετά την αποκατάσταση της σύνδεσης, καλείται η συνάρτηση CopyData προκειμένου να λάβει χώρα η ανταλλαγή των δεδομένων ανάμεσα στις διεργασίας του πελάτη και του διακομιστή. Παρατηρήστε πως ενώ στον πελάτη η CopyData καλείται μία και μοναδική φορά, στο διακομιστή καλείται μέσα από έναν επαναληπτικό βρόχο κάτι που είναι αναμενόμενο, διότι ο διακομιστής αναμένει συνεχώς αιτήματα σύνδεσης από πολλούς και διαφορετικούς πελάτες (αλλά μόνο από ένα πελάτη κάθε φορά).*

```
int main(void) {
   struct sockaddr_un address;
   int sock;
   socklen_t addrLength;
   if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0){
       perror("Message from socket:");
       exit(1); }
   address.sun_family = AF_UNIX;    /* Unix domain socket */
   strcpy(address.sun_path, "./sample-socket");
   addrLength = sizeof(address.sun_family) + strlen(address.sun_path);
   if (connect(sock, (struct sockaddr *) &address, addrLength)){
       perror("Message from connect:");
       exit(1); }
   copyData(0, sock);
   close(sock);
   return 0; }
```

Τυπικό παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Παρατηρήστε πως ενώ ο διακομιστής στο αριστερό τερματικό έχει ξεκινήσει μία και μοναδική φορά αναμένοντας συνδέσεις, στο δεξί τερματικό εμφανίζονται περισσότεροι από ένας πελάτες οι οποίοι αποστέλλουν ένα ενδεικτικό μήνυμα στο διακομιστή και στη συνέχεια τερματίζουν τη λειτουργία τους με το συνδυασμό πλήκτρων Ctrl-C. Όταν τερματιστεί η σύνδεση με ένα πελάτη δια της χρήσεως του εν λόγω συνδυασμό πλήκτρων, ο διακομιστής αποκρίνεται εκτυπώνοντας το μήνυμα done στη συνέχεια εξακολουθεί να παραμένει ενεργός αναμένοντας αιτήματα άλλων πελατών. Ο διακομιστής τερματίζεται και αυτός με τον ίδιο τρόπο, δηλαδή με την το συνδυασμό πλήκτρων Ctrl-C.

```
amarg@amarg-vbox:~$ ./un-server
---- getting data
Hello from Client1
---- done
---- getting data
Hello from Client2
---- done
---- getting data
Hello from Client3
---- done
---- getting data
Hello from Client4
---- done
^C
amarg@amarg-vbox:~$
```

```
amarg@amarg-vbox:~$ ./un-client
Hello from Client1
^C
amarg@amarg-vbox:~$ ./un-client
Hello from Client2
^C
amarg@amarg-vbox:~$ ./un-client
Hello from Client3
^C
amarg@amarg-vbox:~$ ./un-client
Hello from Client4
^C
amarg@amarg-vbox:~$
```

Παράδειγμα 2. Σε αυτό το παράδειγμα αρχιτεκτονικής client – server με Unix sockets, ο server και ο client σχετίζονται με μία σχέση πατέρα (server) – παιδιού (client) που δημιουργείται μέσω της fork και επικοινωνούν στέλνοντας ένα μήνυμα ο ένας στον άλλο. (αρχείο fork-cl-srv.c).

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>

#define SOCKETNAME "MySocket"
```

Αυτό το δεύτερο παράδειγμα επικοινωνίας πελάτη με διακομιστή χρησιμοποιώντας Unix Sockets είναι ακριβώς το ίδιο με πριν, με τη διαφορά ότι ενώ προηγουμένως οι δύο διεργασίες ήταν εντελώς ανεξάρτητες μεταξύ τους σε αυτό το παράδειγμα σχετίζονται μεταξύ τους με σχέση πατέρα - παιδιού. Κατά συνέπεια εκτελείται ο ίδιος κώδικας.

```
int main(void) {

   struct sockaddr_un sa;
   int fd_skt, fd_client; char buf[100];
   int written; ssize_t readb;
```

Όπως και πριν, εάν το αρχείο MySocket υπάρχει από προηγούμενη εκτέλεση του κώδικα, διαγράφεται έτσι ώστε να δημιουργηθεί από την εφαρμογή

```
   (void) unlink (SOCKETNAME);
```

ενώ στη συνέχεια αρχικοποιείται η δομή *sockaddr_un sa η οποία θα χρησιμοποιηθεί στις συναρτήσεις connect του client και bind του server.*

```
   strcpy (sa.sun_path, SOCKETNAME);
   sa.sun_family = AF_UNIX;
```

Ο κώδικας της θυγατρικής διεργασίας (πελάτης)

Στο σημείο αυτό καλείται η συνάρτηση fork για τη δημιουργία της θυγατρικής διεργασίας. Εάν η επιστρεφόμενη τιμή αυτής της συνάρτησης είναι ίση με το μηδέν, τότε όπως έχουμε ήδη αναφέρει, ο κώδικας που αντιστοιχεί σε αυτή τη συνθήκη, είναι ο κώδικας της θυγατρικής διεργασίας. Στο πρώτο βήμα της διαδικασίας, η θυγατρική διεργασία καλεί τη συνάρτηση socket για να δημιουργήσει ένα socket για επικοινωνία με το διακομιστή και εφόσον όλα λειτουργήσουν χωρίς πρόβλημα, εκτυπώνει ένα ενημερωτικό μήνυμα.

```
if (fork() == 0) { /* client */
  if ((fd_skt = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
       perror("Message from socket [client] : ");
      exit(-1); }
  printf ("[Client] ==> Socket %d has been created\n", fd_skt);
```

Επειδή όπως είναι γνωστό, η γονική και η θυγατρική διεργασία εκτελούνται ταυτόχρονα, υπάρχει περίπτωση η θυγατρική διεργασία να καλέσει τη συνάρτηση connect πριν τη δημιουργία του socket από την γονική διεργασία. Εάν συμβεί αυτό, προφανώς η επικοινωνία μεταξύ των διεργασιών δεν είναι δυνατή. Η θυγατρική διεργασία διαπιστώνει την εμφάνιση αυτής της προβληματικής κατάστασης, ελέγχοντας την επιστρεφόμενη τιμή της συνάρτησης connect και εξετάζοντας εάν αυτή είναι ίση με ENOENT. Στον κώδικα που ακολουθεί, η θυγατρική διεργασία καλεί επαναληπτικά τη συνάρτηση connect. Εάν διαπιστώσει πως η εν λόγω συνάρτηση επιστρέφει την τιμή ENOENT, αναστέλλει τη λειτουργία της για ένα δευτερόλεπτο καλώντας τη συνάρτηση sleep και προσπαθεί να συνδεθεί εκ νέου, κατά την επόμενη επανάληψη. Όταν η γονική διεργασία δημιουργήσει το δικό της socket, τότε η συνάρτηση connect θα επιστρέψει κωδικό επιτυχίας, γεγονός που σημαίνει πως η σύνδεση ανάμεσα στις δύο διεργασίες έχει αποκατασταθεί.

```
while (connect(fd_skt, (struct sockaddr *)&sa, sizeof(sa)) == -1) {
   if (errno == ENOENT) {
      sleep(1); continue; }
   else {
    perror("Message from connect [client]"); exit(-2); }}
```

Μετά την αποκατάσταση της επικοινωνίας, ο πελάτης αρχικά γράφει στο server καλώντας τη συνάρτηση write

```
printf ("[Client] ==> Connection has been established .. let us write to server\n");
written = write(fd_skt, "Hello!", 7);
if (written==-1) {
    perror("Message from write [client]");  exit(-3); }
else printf ("[Client] ==> %d bytes written to server\n", written);
```

και στη συνέχεια διαβάζει από το server καλώντας τη συνάρτηση read.

```
printf ("[Client] ==> Now let us read from server\n");
readb = read(fd_skt, buf, sizeof(buf));
if (readb==-1) {
    perror("Message from read [client]");
    exit(-4); }
else printf ("[Client] ==> %zd bytes read from server\n", readb);
printf("Client got %s\n", buf);
```

Οι διαδικασίες ανάγνωσης και εγγραφής πραγματοποιούνται χρησιμοποιώντας το socket fd που δημιούργησε η connect το οποίο μετά τον τερματισμό της επικοινωνίας κλείνει με τη συνάρτηση close.

```
close(fd_skt);
exit(0); }
```

Ο κώδικας της γονικής διεργασίας (διακομιστής)

```
else { /* server */
```

Όπως και στο προηγούμενο παράδειγμα, έτσι και στην προκειμένη περίπτωση, ο διακομιστής καλεί τη συνάρτηση socket για να δημιουργήσει το δικτυακό υποδοχέα στον οποίο θα αναμένει συνδέσεις από τους πελάτες, τη συνάρτηση bind για να συσχετίσει τη διεύθυνση sa (που έχει αρχικοποιηθεί με τον τρόπο που το κάναμε προηγουμένως) με το εν λόγω socket και τη συνάρτηση listen η οποία του επιτρέπει να μπει σε κατάσταση αναμονής αιτημάτων σύνδεσης από τους πελάτες.

```
if ((fd_skt = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
    perror("Message from socket [server]");
    exit(-1); }
printf ("[Server] ==> Socket %d has been created\n", fd_skt);
if (bind(fd_skt, (struct sockaddr *) &sa, sizeof(sa))){
   perror("Message from bind [server]");
   exit(-2); }
printf ("[Server] ==> Socket %d has been bound to address\n", fd_skt);
if (listen(fd_skt, 5)){
   perror("Message from listen [server]");
   exit(-3); }

printf ("[Server] ==> Listening for incoming messages\n");
```

Στη συνέχεια καλεί τη συνάρτηση accept για να δημιουργήσει μία σύνδεση εξυπηρέτησης με τον πελάτη.

```
if ((fd_client = accept(fd_skt, NULL, 0))<0) {
   perror("Message from accept [server]");
   exit(-4); }
```

Μετά την αποκατάσταση της επικοινωνίας, ο διακομιστής αρχικά διαβάζει από τον client καλώντας τη συνάρτηση read

```
printf ("[Server] ==> let us read from client via socket %d\n", fd_client);
  readb = read(fd_client, buf, sizeof(buf));
  if (readb == -1) {
      perror("Message from read [server] : ");
      exit(-5); }
  printf("Server got %s ==> %zd bytes read from client\n", buf, readb);
```

και στη συνέχεια γράφει στον client καλώντας τη συνάρτηση write.

```
printf ("[Server] ==> let us write to client\n");
if (write(fd_client, "Goodbye!", 9)==-1) {
    perror("Message from write [server]");
    exit(-6); }
```

Στο τέλος της διαδικασίας κλείνει τα sockets που χρησιμοποιήθηκαν.

```
close(fd_skt);
close(fd_client);
exit(0); }}
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ./fork-cl-srv
[Server] ==> Socket 3 has been created
[Server] ==> Socket 3 has been bound to address
[Server] ==> Listening for incoming messages
[Client] ==> Socket 3 has been created
[Server] ==> let us read from client via socket 4
[Client] ==> Connection has been established .. let us write to server
Server got Hello! ==> 7 bytes read from client
[Server] ==> let us write to client
amarg@amarg-vbox:~$ [Client] ==> 7 bytes written to server
[Client] ==> Now let us read from server
[Client] ==> 9 bytes read from server
Client got Goodbye!
```

Παράδειγμα 3. Σε αυτό το παράδειγμα αρχιτεκτονικής client – server με TCP / IP sockets, ο client αποστέλλει στον server ένα μήνυμα που παραλαμβάνεται και εκτυπώνεται από αυτόν. (αρχεία server1-tcp.c και client1-tcp.c).

Σε αυτό το παράδειγμα όπως και στο προηγούμενο ο ο πελάτης αποστέλλει ένα μήνυμα στον διακομιστή ο οποίος το παραλαμβάνει και το εκτυπώνει η διαφορά είναι που στην προκειμένη περίπτωση δεν χρησιμοποιούνται unix αλλά tcp IP sockets ωστόσο η λογική είναι ακριβώς η ίδια. Το socket που χρησιμοποιείται είναι SOCK_STREAM και κατά συνέπεια χρησιμοποιείται επικοινωνία με σύνδεση.

Κώδικας διακομιστή

Αρχείο server1-tcp.c

```
#include <stdio.h>
#include <sys/socket.h> // AF_INET and SOCK_STREAM
#include <stdlib.h>     // EXIT_SUCCESS, EXIT_FAILURE
#include <netinet/in.h> // INADDR_ANY
#include <string.h>     // bzero
#include <unistd.h>     // read and write
```

#define PORT 8080 ← αριθμός θύρας για σύνδεση = 8080 (συνήθως για proxy service)

```
int main(int argc, char const *argv[]) {
   int server_fd, new_socket, valread;
   struct sockaddr_in address;
   int opt = 1;
   int addrlen = sizeof(address);
   char buffer[1024] = {0};
   char *hello = "Hello from server";
```

Δημιουργία του socket στο server AF_INET → IPv4, SOCK_STREAM → TCP

```
   if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
      perror("socket failed");
      exit(EXIT_FAILURE); }
```

Εδώ μπορούμε να ορίσουμε κάποιες παραμέτρους λειτουργίας του socket. Είναι αρκετά τεχνικό και εξειδικευμένο κομμάτι και δεν έχει νόημα να ασχοληθούμε περισσότερο σε αυτό το εισαγωγικό επίπεδο (εκείνο που γίνεται εδώ είναι να επιτραπεί η εκκίνηση πολλαπλών στιγμιότυπων διακομιστής στην ίδια θύρα υπό την προϋπόθεση που σχετίζονται με διαφορετική τοπική διεύθυνση IP).

```
if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,   &opt, sizeof(opt))) {
   perror("setsockopt");
   exit(EXIT_FAILURE); }
```

Όπως και πριν αρχικοποιούμε με τον κατάλληλο τρόπο τα διάφορα πεδία της δομής address που τώρα είναι τύπου *sockaddr_in* . Η τιμή AF_INET υποδηλώνει πως θα χρησιμοποιηθεί το πρωτόκολλο IPv4, η τιμή *INADDR_ANY υποδηλώνει πως το socket δεν θα δέχεται αιτήσεις σύνδεσης από μία συγκεκριμένη διεύθυνση αλλά από κάθε διεύθυνση, ενώ το πεδίο port λαμβάνει την τιμή της θύρας στην οποία θα αναμένονται αιτήσεις σύνδεσης, που είναι η θύρα 8080.*

```
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons (PORT);
```

Τα υπόλοιπα είναι ακριβώς ίδια με τα προηγούμενα παραδείγματα αφού τα Unix sockets και τα TCP / IP sockets δουλεύουν με τον ίδιο ακριβώς τρόπο. Αρκετά καλούμε bind για να συσχετίσουμε το socket που δημιουργήθηκε πιο πάνω με την παραπάνω διεύθυνση. Στη συνέχεια καλούμε τη listen έτσι ώστε ο διακομιστής να είναι σε θέση να τεθεί σε κατάσταση αναμονής αιτημάτων σύνδεσης από πελάτες (παρατηρήστε πως στην προκειμένη περίπτωση το δεύτερο όρισμα είναι ισο με 3 και κατά συνέπεια στην ουρά αναμονής εκκρεμών αιτημάτων σύνδεσης μπορούν να εκχωρηθούν μέχρι τρία τέτοια αιτήματα), ενώ τέλος καλείται η accept που αποκαθιστά την επικοινωνία με τον πελάτη και επιστρέφει το socket μέσω του οποίου θα πραγματοποιηθεί η διαδικασία της επικοινωνίας (σημειώστε πως όλοι οι διακομιστές δουλεύουν παντού και πάντα με τον ίδιο ακριβώς τρόπο δηλαδή καλούνε με τη σειρά τις συναρτήσεις socket, bind, listen και accept (ενώ κάτι αντίστοιχο ισχύει και για τους πελάτες, οι οποίοι καλούν παντού και πάντα τις συναρτήσεις socket και connect) και αυτό σημαίνει πως εάν εάν κατανοηθεί πλήρως ένα παράδειγμα επικοινωνίας πελάτη διακομιστή θα κατανοηθούν και όλα τα υπόλοιπα παρόμοια παραδείγματα αφού τα πάντα λειτουργούν ακριβώς με τον ίδιο τρόπο). Μετά την αποκατάσταση της σύνδεσης καλείται η συνάρτηση read προκειμένου ο διακομιστής να διαβάσει το μήνυμα που του αποστέλλει ο πελάτης χρησιμοποιώντας ως περιγραφέα αρχείου το socket που επέστρεψε η access. Μετά την παραλαβή του μηνύματος, αυτό εκτυπώνεται στην οθόνη.

```
if  (bind (server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
   perror("bind failed");
   exit(EXIT_FAILURE); }

if (listen(server_fd, 3) < 0) {
   perror("listen");
   exit(EXIT_FAILURE); }

if ((new_socket = accept (server_fd, (struct sockaddr *)&address,
                                    (socklen_t*)&addrlen)) < 0) {
   perror("accept");
   exit(EXIT_FAILURE); }

valread = read (new_socket, buffer, 1024);

printf("%s\n",buffer );
return 0; }
```

Στην εφαρμογή του πελάτη πραγματοποιείται ακριβώς η ίδια διαδικασία και η ίδια αρχικοποίηση μεταβλητών. Με άλλα λόγια, χρησιμοποιείται και εδώ η θύρα 8080 αφού προφανώς για να επικοινωνήσουν δύο διεργασίες θα πρέπει να χρησιμοποιήσουν την ίδια θύρα. Η συνάρτηση socket καλείται ακριβώς με τις ίδιες παραμέτρους με την αντίστοιχη συνάρτηση του διακομιστή, δηλαδή με ορίσματα τις τιμές AF_INET (που υποδηλώνει τη χρήση του πρωτοκόλλου IPv4) και SOCK_STREAM που υποδηλώνει τη χρήση TCP socket και υπηρεσία με σύνδεση. Μετά την κατάλληλη αρχικοποίηση και τη δημιουργία του socket, καλείται η συνάρτηση connect για τη σύνδεση στο διακομιστή και λαμβάνει χώρα η αποστολή του

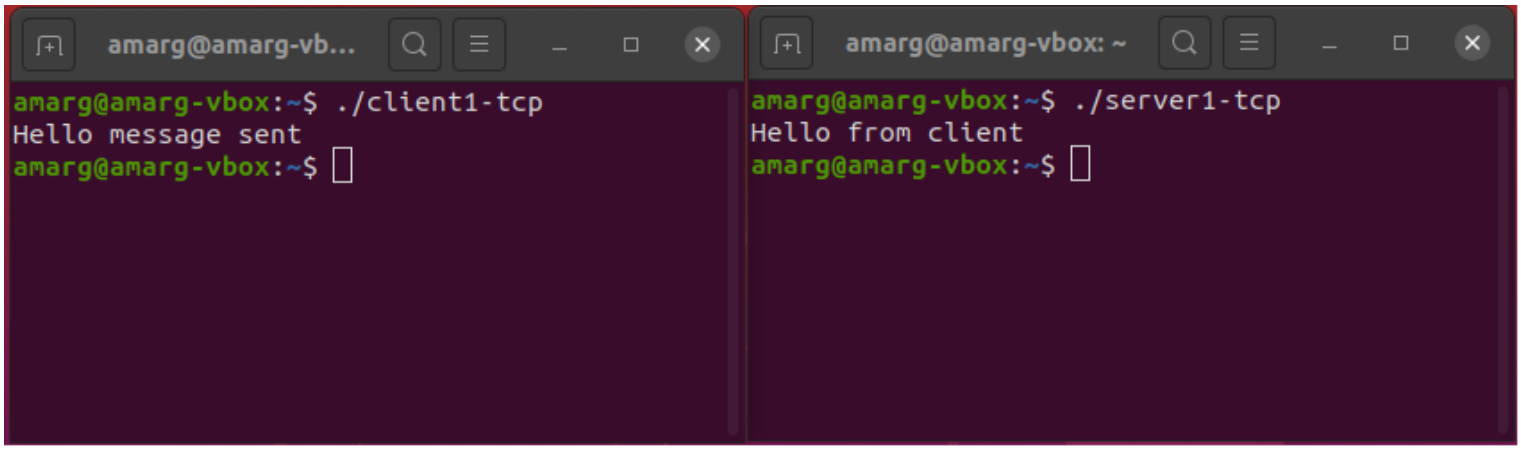
μηνύματος στο διακομιστή χρησιμοποιώντας τη συνάρτηση send. Η μοναδική νέα συνάρτηση που βλέπετε εδώ είναι η *inet_pton η οποία μετατρεπει μία διεύθυνση από μορφή κειμένου (για παράδειγμα 127.0.0.1) σε δυαδική μορφή (σε έναν δυαδικό αριθμό μήκους 32 bits) προκειμένου να χρησιμοποιηθεί στη συνάρτηση connect (που απαιτεί δυαδικό όρισμα).*

Κώδικας πελάτη

Αρχείο client1-tcp.c

```
#include <stdio.h>
#include <sys/socket.h> // AF_INET and SOCK_STREAM
#include <arpa/inet.h>  // storage size of serv_addr
#include <unistd.h>     // getpid, getppid, fork
#include <string.h>     // bzero

#define PORT 8080

int main(int argc, char const *argv[]) {
   int sock = 0, valread;
   struct sockaddr_in serv_addr;
   char *hello = "Hello from client";
   char buffer[1024] = {0};

   if ((sock = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
      printf("\n Socket creation error \n");
      return -1; }

   serv_addr.sin_family = AF_INET;
   serv_addr.sin_port = htons(PORT);

   // Convert IPv4 and IPv6 addresses from text to binary form

   if (inet_pton (AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0) {
      printf("\nInvalid address/ Address not supported \n");
      return -1; }

   if (connect (sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
      printf("\nConnection Failed \n");
      return -1; }

   send (sock , hello , strlen(hello) , 0 );
   printf("Hello message sent\n");
   return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Λαμβάνοντας υπόψη πως ο πελάτης και ο διακομιστής αποτελούν δύο ανεξάρτητες μεταξύ τους διεργασίες για την καθεμία εκ των οποίων υπάρχει ξεχωριστός πηγαίος κώδικας και απαιτείται ξεχωριστή διαδικασία μεταγλώττισης και δημιουργίας του εκτελέσιμου αρχείου, είναι αυτονόητο πως οι δύο αυτές διεργασίες θα εκτελεστούν η καθεμία στο δικό της τερματικό όπως συμβαίνει συνήθως σε αυτές τις περιπτώσεις (μόνο όταν ο πελάτης και ο διακομιστής συνδέονται με σχέση γονικής - θυγατρικής διεργασίας εκτελείται ο ίδιος κώδικας και ως εκ τούτου χρειαζόμαστε μόνο ένα τερματικό - σε όλες τις άλλες περιπτώσεις απαιτείται η χρήση δύο τερματικών).



Παράδειγμα 4. Η εφαρμογή πελάτης συνδέεται σε έναν απομακρυσμένο Web server (port 80) η IP διεύθυνση του οποίου δίδεται από το χρήστη και διαβάζει τις 1000 πρώτες γραμμές του κώδικα HTML της κεντρικής σελίδας τις οποίες και εκτυπώνει. (αρχείο inetExample.c).

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
```

Όπως είναι γνωστό, η επικοινωνία με μία ιστοσελίδα του παγκόσμιου διαδικτύου, στηρίζεται στη χρήση του πρωτοκόλλου HTTP (Hhypertext Transfer Protocol) το οποίο μεταξύ άλλων προσφέρει τις δικές του εντολές για την ανταλλαγή πληροφοριών ανάμεσα στον Web Client και στο Web Server. Σε αυτό το παράδειγμα, θα χρησιμοποιήσουμε την εντολή GET του πρωτοκόλλου HTTP 1.0 για να διαβάσουμε τμήμα του κώδικα HTML της κεντρικής σελίδας του απομακρυσμένου δικτυακού τόπου. Χωρίς να είμαστε σε θέση σε αυτό το σημείο να δώσουμε πολλές πληροφορίες σχετικά με την ακριβή σύνταξη των εντολών του εν λόγω πρωτοκόλλου, αναφέρουμε απλά πώς το αίτημα (REQUEST) που θα πρέπει να στείλουμε στο διακομιστή για να πραγματοποιήσουμε αυτή τη διαδικασία θα πρέπει να έχει τη μορφή *"GET / HTTP/1.0\r\n\r\n" (για όποιον ενδιαφέρεται να ενημερωθεί σχετικά με τον τρόπο λειτουργίας και τα χαρακτηριστικά αυτού του πρωτοκόλλου υπάρχουν αναρίθμητες διευθύνσεις στο διαδίκτυο με τις σχετικές πληροφορίες όπως είναι για παράδειγμα η διεύθυνση https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol).

```
#define REQUEST "GET / HTTP/1.0\r\n\r\n"
```

Λαμβάνοντας υπόψη πως αυτή τη φορά θα συνδεθούμε σε έναν απομακρυσμένο διακομιστή ο κώδικας του οποίου προφανώς έχει γραφεί από κάποιον άλλο, το μόνο το οποίο θα κάνουμε είναι να γράψουμε τον κώδικα του πελάτη ο οποίος θα συνδεθεί σε αυτόν τον διακομιστή. Ο διακομιστής υπάρχει κάπου και απλά συνδεόμαστε σε αυτόν για να ζητήσουμε κάποια εξυπηρέτηση. Για να το κάνουμε βέβαια αυτό, θα πρέπει να γνωρίζουμε το πρωτόκολλο στο οποίο στηρίζεται η λειτουργία του (για παράδειγμα αν είναι Web server, FTP server, Email server, κ.τ.λ), έτσι ώστε να ξέρουμε με ποιον τρόπο θα επικοινωνήσουμε μαζί του. Στο εν λόγω παράδειγμα, ο απομακρυσμένος διακομιστής είναι ένας web server ο οποίος χρησιμοποιεί το πρωτόκολλο HTTP. Εάν λοιπόν θέλουμε να κατασκευάσουμε μία πραγματική επαγγελματική εφαρμογή και όχι ένα μικρό πρόγραμμα επίδειξης σαν και αυτό που παρουσιάζεται εδώ, θα πρέπει να γνωρίζουμε πλήρως τον τρόπο λειτουργίας και τις εντολές αυτού του πρωτοκόλλου.

```
int main(int argc, char * argv[]) {
   struct sockaddr_in sa;
   int fd_skt;
   char buf[1000];
   ssize_t nread;
```

Στην προκειμένη περίπτωση για να εκτελεστεί ο κώδικας θα πρέπει να δώσουμε από τη γραμμή εντολών την διεύθυνση του απομακρυσμένου διακομιστή στον οποίο θα συνδεθεί η εφαρμογή πελάτης μας. Αυτό σημαίνει πως η σωστή χρήση της εφαρμογής απαιτεί την καταχώρηση ακριβώς ενός ορίσματος, δηλαδή την εκχώρηση από το σύστημα στη μεταβλητή

argc της τιμής 2. Εάν δεν ισχύει αυτό, τότε το πρόγραμμα τερματίζεται εκτυπώνονται στο κατάλληλο ενημερωτικό μήνυμα χρήσης (πρόκειται για έναν έλεγχο που τον έχουμε κάνει αρκετές φορές στο παρελθόν και ως εκ τούτου θεωρείται γνωστός).

```
if (argc!=2) {
  printf ("Usage: inetExample xxx.xxx.xxx.xxx\n");
  exit (-1); }
```

Η αρχικοποίηση της δομής *sockaddr_in* γίνεται όπως και πριν, χρησιμοποιώντας ως όρισμα το *AF_INET* (αφού θα χρησιμοποιήσουμε το δικτυακό πρωτόκολλο *IPv4*) ενώ ως αριθμός θύρας χρησιμοποιούμε τον αριθμό *80* αφού είναι γνωστό πως αυτός είναι ο αριθμός θύρας του πρωτοκόλλου HTTP. Λαμβάνοντας υπόψη πως η διεύθυνση στην οποία θα συνδεθούμε θα δοθεί από το χρήστη μέσω του πληκτρολογίου, είναι προφανές πως αυτή θα ανακτηθεί από το όρισμα *argv[1]* το οποίο χρησιμοποιείται για αυτόν ακριβώς τον λόγο.

```
sa.sin_family = AF_INET;
sa.sin_port = htons(80);
sa.sin_addr.s_addr =  inet_addr(argv[1]);
```

Οι εφαρμογές διακομιστή, είτε υλοποιούνται από εμάς είτε έχουν υλοποιηθεί από άλλους, είναι απομακρυσμένοι και απλά συνδεόμαστε σε αυτούς, δουλεύουν παντού και πάντα με τον ίδιο τρόπο, δηλαδή καλώντας τις συναρτήσεις socket, bind, listen και accept με τον τρόπο που περιγράψαμε στα προηγούμενα παραδείγματα. Ο πελάτης λοιπόν σε αυτό το παράδειγμα, θα κάνει ό,τι έκανε και πριν, δηλαδή θα καλέσει τη συνάρτηση socket με τις ίδιες ρυθμίσεις με αυτές του διακομιστή, έτσι ώστε τελικά να μπορέσει να επικοινωνήσει μαζί του και θα συνδεθεί σε αυτόν χρησιμοποιώντας τη συνάρτηση connect επίσης με τον τρόπο που είδαμε (γενικά είναι καλό να γνωρίζετε, πως όλες αυτές οι εφαρμογές πελάτη διακομιστή είναι εύκολες στον προγραμματισμό τους, όσον αφορά το κομμάτι της διασύνδεσης, διότι κάνουν όλες ακριβώς τα ίδια πράγματα).

```
if ((fd_skt = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Message from socket [client] : ");
    exit(-1); }

if (connect (fd_skt, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
   printf("\nConnection Failed \n");
   return -1; }
```

Μετά τη σύνδεση στο διακομιστή, ο πελάτης του αποστέλλει το αίτημα *REQUEST* με τιμή *"GET / HTTP/1.0\r\n\r\n"* γνωστοποιώντας του πως θέλει να διαβάσει δεδομένα από αυτόν. Όπως και στις προηγούμενες εφαρμογές, η αποστολή του μηνύματος πραγματοποιείται με τη συνάρτηση *write* και χρησιμοποιώντας τον περιγραφέα αρχείου *fd_skt* που επέστρεψε η συνάρτηση *connect*.

```
write (fd_skt, REQUEST, strlen(REQUEST));
```

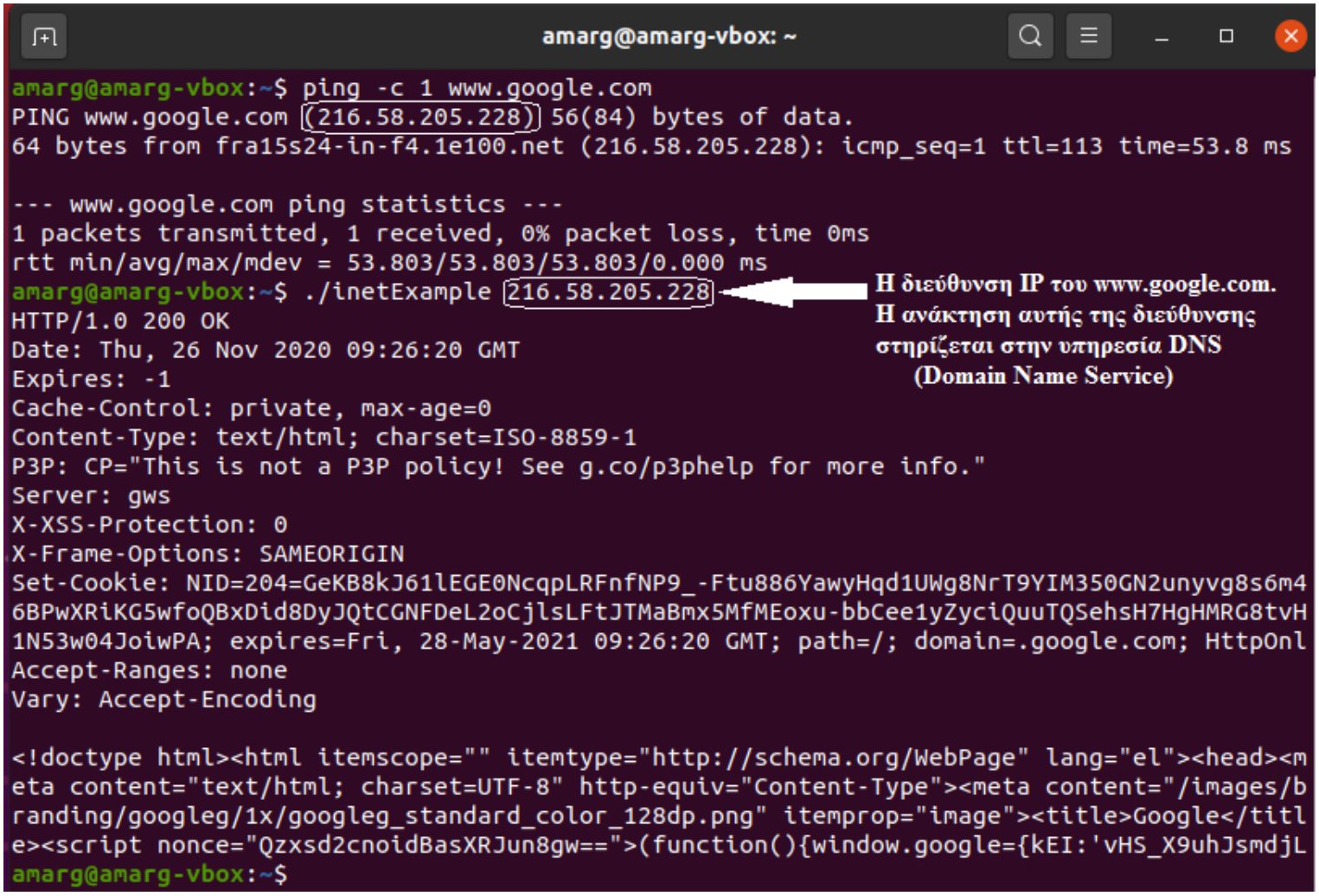
Ο διακομιστής αποκρίνεται στο αίτημα του πελάτη επιτρέποντάς του να διαβάσει δεδομένα κάτι που γίνεται από τον πελάτη καλώντας τη συνάρτηση read. Το πλήθος των δεδομένων που θα διαβάσει ο πελάτης υπαγορεύεται από το μέγεθος της περιοχής μνήμης στην οποία θα αποθηκευτούν τα δεδομένα που διαβάζονται και το οποίο στην προκειμένη περίπτωση είναι ίσο με 1000. Εάν θέλετε να διαβάσετε περισσότερα ή λιγότερα δεδομένα τροποποιήστε την τιμή αυτού του αριθμού.

```
nread = read (fd_skt, buf, sizeof(buf));
```

Η εφαρμογή ολοκληρώνεται με την εκτύπωση των δεδομένων από τον πελάτη στην οθόνη του τερματικού του, κάτι που δίνετε καλώντας τη συνάρτηση write με περιγραφέα αρχείου τον *STDOUT_FILENO που παραπέμπει στην προεπιλεγμένη έξοδο. Κατά τον τερματισμό της διαδικασίας κλείνει και το socket επικοινωνίας με το διακομιστή.*

```
(void) write (STDOUT_FILENO, buf, nread);
close(fd_skt);
exit(0); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Επειδή αυτή η εφαρμογή δέχεται ως όρισμα αριθμητική και όχι συμβολική διεύθυνση IP, αρχικά καλούμε την εντολή ping με όρισμα τη συμβολική διεύθυνση την οποία θέλουμε να χρησιμοποιήσουμε, έτσι ώστε να ανακτήσουμε την αριθμητική διεύθυνση IP του απομακρυσμένου διακομιστή.



Παράδειγμα 5. Ο χρήστης καλεί την ping με μία συμβολική διεύθυνση για να ανακτήσει την πραγματική IP διεύθυνση την οποία στη συνέχεια καταχωρεί ως όρισμα στην εφαρμογή addrInfo. (αρχείο addrInfo.c).

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>
```

Αυτό το παράδειγμα επιδεικνύει τον τρόπο χρήσης της εντολής getaddrInfo. Η εντολή δέχεται ως όρισμα (1) μία διεύθυνση IP (στη μορφή xxx.xxx.xxx.xxx), (2) μία υπηρεσία (η οποία περιγράφεται από έναν αριθμό θύρας) και (3) έναν δείκτη σε μία δομή *addrinfo* (στο παράδειγμά μας αυτό το όρισμα είναι το hint) η οποία καθορίζει τα κριτήρια επιλογής των πληροφοριών που επιστρέφονται. Η συνάρτηση επιστρέφει μέσω του τέταρτου ορίσματος που δέχεται (αυτό το όρισμα στο παράδειγμά μας είναι το όρισμα infop) μία ή περισσότερες δομές τύπου *addrinfo* η καθεμία των οποίων περιέχει μία διεύθυνση IP που να μπορεί να χρησιμοποιηθεί σε μία κλήση της bind ή της connect.

Επειδή η συνάρτηση δέχεται ως όρισμα αριθμητική διεύθυνση IP της μορφής xxx.xxx.xxx.xxx ενώ οι χρήστες συνήθως χρησιμοποιούν για λόγους ευκολίας συμβολικές διευθύνσεις (για παράδειγμα, www.uth.gr), αρχικά όπως θα δούμε κατά την εκτέλεση του προγράμματος θα καλέσουμε την εντολή ping όπως κάναμε και στο προηγούμενο παράδειγμα, η οποία για την συμβολική διεύθυνση που θέλουμε να χρησιμοποιήσουμε, θα μας επιστρέψει την πραγματική αριθμητική διεύθυνση η οποία θα διαβιβαστεί στην εφαρμογή ως όρισμα από τη γραμμή εντολών. Αντιλαμβάνεστε πως πλέον έχουμε

φύγει από τις εφαρμογές client-server (αν και θα επιστρέψουμε στη συνέχεια), αφού αυτό το παράδειγμα επιδεικνύει απλά τον τρόπο χρήσης μιας συνάρτησης ανάκτησης πληροφοριών. Το TCP/IP προσφέρει πάρα πολλές τέτοιες συναρτήσεις ανάκτησης δικτυακών πληροφοριών και η getaddressinfo είναι απλά μία από αυτές.

```
int main (int argc, char * argv[]) {
   int retVal; char err[50];
   struct addrinfo *infop = NULL, hint;
```

Σε αυτό το σημείο αρχικοποιούμε τη δομή hint η οποία θα αποτελέσει το τρίτο όρισμα της εντολής. Αρχικά θέτουμε σε όλα τα πεδία της την τιμή 0 και στη συνέχεια εκχωρούμε στα κατάλληλα πεδία τις τιμές AF_INET και SOCK_STREAM για να ορίσουμε πως ενδιαφέρομαστε για διευθύνσεις που χρησιμοποιούν το πρωτόκολλο IPV4 και TCP Sockets και κατά συνέπεια προσφέρουν επικοινωνία με σύνδεση.

```
   memset(&hint, 0, sizeof(hint));
   hint.ai_family = AF_INET;
   hint.ai_socktype = SOCK_STREAM;
```

Εδώ πραγματοποιούμε το συνήθη έλεγχο ορθής χρήσης αφού για να εκτελεστεί ο κώδικας θα πρέπει να δώσουμε από τη γραμμή εντολών την διεύθυνση του απομακρυσμένου δικτυακού τόπου. Εάν λοιπόν δεν ισχύει η συνθήκη argc = 2, τότε το πρόγραμμα τερματίζεται εκτυπώνονται στο κατάλληλο ενημερωτικό μήνυμα χρήσης

```
 if (argc!=2) {
      printf ("Usage: inetExample xxx.xxx.xxx.xxx\n");
      exit (-1); }
```

Στο σημείο αυτό καλείται η συνάρτηση με πρώτο όρισμα την αριθμητική διεύθυνση IP που καταχώρησε ο χρήστης, δεύτερο όρισμα τον αριθμό θύρας 80 (που παραπέμπει σε πρωτόκολλο HTTP) – προσέξτε πως ο αριθμός μπαίνει σε εισαγωγικά γιατί το δεύτερο όρισμα της συνάρτησης δηλώνεται ως const char *service και κατά συνέπεια δεν είναι αριθμός αλλά συμβολοσειρά και τρίτο όρισμα τη δομή hint που αρχικοποιήσαμε παραπάνω. Η συνάρτηση επιστρέφει τη ζητούμενη πληροφορία στο τέταρτο όρισμα infop.

```
retVal = getaddrinfo(argv[1], "80", &hint, &infop);
```

Εάν η επιστρεφόμενη τιμή της συνάρτησης δεν είναι μηδέν, αυτό σημαίνει πως έλαβε χώρα κάποιο σφάλμα και ως εκ τούτου το πρόγραμμα τερματίζεται εκτυπώνοντας το κατάλληλο μήνυμα σφάλματος.

```
if (retVal!=0) {
      strcpy (err,gai_strerror(retVal));
      printf ("Error found ==> %s\n", err);
      exit (-1); }
```

Εάν η εντολή επιστρέψει με επιτυχία, εκτυπώνει τις τιμές των πληροφοριών που ανέκτησε. Επειδή στη γενική περίπτωση η εντολή μπορεί να επιστρέψει περισσότερες από μία δομές addrinfo, η εκτύπωση των πληροφοριών γίνεται μέσα από έναν επαναληπτικό βρόχο στον οποίο σε κάθε κύκλο επανάληψης εκτυπώνεται το περιεχόμενο της καθεμιάς από αυτές τις δομές. Αυτές οι δομές αποτελούν τους κόμβους μιας λίστας με τον επόμενο κόμβο που προσπελαύνεται να προσδιορίζεται από τον δείκτη next και τον επαναληπτικό βρόχο να εκτελείται για όσο χρονικό διάστημα ο δείκτης next δεν έχει τιμή NULL και κατά συνέπεια υπάρχει επόμενη δομή για εκτύπωση.

```
for ( ; infop != NULL; infop = infop->ai_next) {
      struct sockaddr_in *sa = (struct sockaddr_in *)infop->ai_addr;
      printf("%s port: %d protocol: %d\n", inet_ntoa(sa->sin_addr),
           ntohs(sa->sin_port), infop->ai_protocol); }
   exit(0);}
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια. Επειδή αυτή η εφαρμογή δέχεται ως όρισμα αριθμητική και όχι συμβολική διεύθυνση IP, αρχικά όπως και πριν καλούμε την εντολή ping με όρισμα τη συμβολική διεύθυνση την οποία θέλουμε να χρησιμοποιήσουμε, έτσι ώστε να ανακτήσουμε την αριθμητική διεύθυνση IP του απομακρυσμένου υπολογιστή.

```
amarg@amarg-vbox:~$ ping www.uth.gr
PING zeus.uth.gr (194.177.200.17) 56(84) bytes of data.
64 bytes from zeus.uth.gr (194.177.200.17): icmp_seq=1 ttl=53 time=39.0 ms
amarg@amarg-vbox:~$ ./addrInfo 194.177.200.17
194.177.200.17 port: 80 protocol: 6
amarg@amarg-vbox:~$
```

Παράδειγμα 6. Ανταλλαγή ενός μηνύματος μεταξύ σταθμών με αυτοδύναμα πακέτα.
(αρχεία server2-udp.c και client2-udp.c).

Στο εν λόγω παράδειγμα, επιδεικνύεται η αποκατάσταση μιας επικοινωνίας χωρίς σύνδεση (ή ασυνδεσμικής επικοινωνίας) στην οποία χρησιμοποιούνται αυτοδύναμα πακέτα (datagrams). Με άλλα λόγια, δεν απαιτείται η προγενέστερη αποκατάσταση μιας σύνδεσης ανάμεσα στα δύο άκρα και για το λόγο αυτό δεν χρησιμοποιούνται οι συναρτήσεις listen και accept. Στις συνδέσεις αυτού του είδους, ο πελάτης ορίζει απλά τη διεύθυνση στην οποία θέλει να αποστείλει το πακέτο και το αποστέλλει χωρίς να υπάρχει καμία εγγύηση και χωρίς να ελέγχει καν εάν το πακέτο έφτασε με επιτυχία. Κατά την παραλαβή, ο πελάτης καθορίζει από που θέλει να παραλάβει το πακέτο και ενημερώνεται για την κατάσταση του αποστολέα. Σε αυτού του τύπου την επικοινωνία δεν υφίσταται αρχιτεκτονική client-server αλλά όλοι οι κόμβοι είναι ομότιμοι δηλαδή λειτουργούν με τον ίδιο τρόπο (ωστόσο, χρησιμοποιούμε αυτούς τους όρους στα ονόματα των αρχείων απλά και μόνο για να τα ξεχωρίσουμε μεταξύ τους – αλλά σε εισαγωγικά για να τονίσουμε την ομοτιμία των κόμβων).

Κώδικας «διακομιστή»

Αρχείο server2-udp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT    8080
#define MAXLINE 1024

int main() {
    int sockfd, len, n;
    char buffer[MAXLINE];
    char *hello = "Hello from server";
    struct sockaddr_in servaddr, cliaddr;
```

Παρά τη διαφορετική φύση της επικοινωνίας, τα πάντα λειτουργούν όπως πριν. Οι δύο διεργασίες καλούν τη συνάρτηση socket για να δημιουργήσουν την κατάλληλη υποδομή για τη μεταξύ τους επικοινωνία, μόνο που αυτή τη φορά χρησιμοποιείται ως όρισμα στη συνάρτηση η σταθερά SOCK_DGRAM για να δείξουμε πως στην προκειμένη περίπτωση η επικοινωνία που αποκαθίσταται είναι επικοινωνία χωρίς σύνδεση.

```
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE); }
```

Στο επόμενο βήμα αρχικοποιείται με τον τρόπο που είδαμε και στα προηγούμενα παραδείγματαη η δομή servaddr προκειμένου να διαβιβαστεί ως όρισμα στη συνάρτηση bind η οποία λειτουργεί και αυτή με τον τρόπο που έχουμε περιγράψει στα προηγούμενα παραδείγματα.

```
memset(&servaddr, 0, sizeof(servaddr));
memset(&cliaddr, 0, sizeof(cliaddr));
servaddr.sin_family    = AF_INET; // IPv4
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons(PORT);

if (bind (sockfd, (const struct sockaddr *)&servaddr,  sizeof(servaddr)) < 0) {
    perror("bind failed");
    exit(EXIT_FAILURE); }
```

Επειδή πρόκειται για επικοινωνία χωρίς σύνδεση δεν καλούνται (όπως αναφέραμε πιο πάνω) οι συναρτήσεις listen και accept οπότε οι δύο διεργασίες προχωρούν απευθείας στην αποστολή και λήψη δεδομένων, χρησιμοποιώντας τις συναρτήσεις sendto και recvfrom. Η διεργασία «διακομιστής» πρώτα διαβάζει το μήνυμα του «πελάτη» και στη συνέχεια αποστέλλει το δικό της μήνυμα σε αυτόν, ενώ η διεργασία «πελάτης» ο πηγαίος κώδικας της οποίας ακολουθεί στη συνέχεια, πραγματοποιεί πρώτα τη διαδικασία αποστολής του μηνύματος στο διακομιστή και στη συνέχεια παραλαμβάνει το μήνυμα που στάλθηκε από αυτόν.

```
n = recvfrom (sockfd, (char *)buffer, MAXLINE, MSG_WAITALL,  (struct sockaddr *) &cliaddr, &len);
buffer[n] = '\0';
printf("Client : %s\n", buffer);

sendto (sockfd, (const char *)hello, strlen(hello), MSG_CONFIRM,  (const struct sockaddr *) &cliaddr, len);

printf("Hello message sent.\n");
return 0; }
```

Κώδικας «πελάτη»

Αρχείο client2-udp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

#define PORT     8080
#define MAXLINE 1024

int main() {
    int sockfd, len, n;
    char buffer[MAXLINE];
    char *hello = "Hello from client";
    struct sockaddr_in     servaddr;
```

Αρχικά καλείται η socket με όρισμα SOCK_DGRAM για τη δημιουργία του socket επικοινωνίας.

```
    if ((sockfd = socket (AF_INET, SOCK_DGRAM, 0)) < 0) {
```

```
    perror("socket creation failed");
    exit(EXIT_FAILURE); }
```

Αμέσως μετά αρχικοποιείται η δομή servaddr προκειμένου να διαβιβαστεί ως όρισμα στη συνάρτηση sendto.

```
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(PORT);
servaddr.sin_addr.s_addr = INADDR_ANY;
```

Ακολουθούν οι διαδικασίες αποστολής και παραλαβής δεδομένων από την ομότιμη διεργασία.

```
sendto (sockfd, (const char *)hello, strlen(hello), MSG_CONFIRM, (const struct sockaddr *) &servaddr, sizeof(servaddr));
printf("Hello message sent.\n");

n = recvfrom (sockfd, (char *)buffer, MAXLINE, MSG_WAITALL,
            (struct sockaddr *) &servaddr, &len);
buffer[n] = '\0';
printf("Server : %s\n", buffer);
close(sockfd);
return 0; }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
amarg@amarg-vbox:~$ ./client2-udp
Hello message sent.
Server : Hello from server
```

```
amarg@amarg-vbox:~$ ./server2-udp
Client : Hello from client
Hello message sent.
```

Παράδειγμα 7. Σε αυτό το παράδειγμα αρχιτεκτονικής client – server με Unix sockets, η συνάρτηση run_server μέσα από ένα infinite loop καλεί συνεχώς την accept για να συλλάβει αιτήματα σύνδεσης από τέσσερις διεργασίες – πελάτες (που υλοποιούνται από τη συνάρτηση run_client), χρησιμοποιώντας τη select. Στη συνέχεια επικοινωνεί με τον καθέναν από τους πελάτες ανταλλάσσοντας με αυτόν ένα μήνυμα. (αρχείο one-serv-n-cls-unix.c).

```
#include <stdio.h>
#include <sys/socket.h> // AF_INET and SOCK_STREAM
#include <arpa/inet.h>  // storage size of serv_addr
#include <unistd.h>     // getpid, getppid, fork
#include <string.h>     // bzero
#include <stdlib.h>
#include <errno.h>
#include <sys/un.h>

#define SOCKETNAME "MySocket"
```

Σε αυτό το παράδειγμα επιδεικνύεται η χρήση της select η οποία αποτελεί τη βασική συνάρτηση επικοινωνίας με πολλές τερματικές μονάδες ταυτόχρονα. Στην προκειμένη περίπτωση η εφαρμογή συνίσταται στη χρήση μίας συνάρτηση διακομιστή με όνομα run_server η οποία δέχεται αιτήματα εξυπηρέτησης από πελάτες χρησιμοποιώντας την accept με την επιλογή του πελάτη που θα συνδεθεί να στηρίζεται στη χρήση της select. Στη συνέχεια ο διακομιστής επικοινωνεί με τον καθέναν από αυτούς τους πελάτες ανταλλάσσοντας με αυτόν ένα μήνυμα. Όσον αφορά στους πελάτες αυτοί αποτελούν θυγατρικές διεργασίες της διεργασίας του διακομιστή και ως εκ τούτου δημιουργούνται με την κλήση της συνάρτησης fork. Ο καθένας από αυτούς τους πελάτες εκτελεί τη συνάρτηση run_client. Η επικοινωνία ανάμεσα στις διεργασίες στηρίζεται στη χρήση Unix Sockets.

Κώδικας διακομιστή

// Ο κώδικας του server

```
static int run_server(struct sockaddr_un *sap) {

   int fd_skt, fd_client, fd_hwm = 0, fd;
   char buf[100];
   fd_set set, read_set;
   ssize_t nread;
```

Ο διακομιστής αρχικά εκτυπώνει στην οθόνη το PID του.

```
   printf ("Server pid is %d\n", getpid());
```

Ο διακομιστής καλεί ως συνήθως τη συνάρτηση socket για να δημιουργήσει το socket με ορίσματα UNIX_SOCK (αφού θα χρησιμοποιηθεί Unix Socket και SOCK_STREAM αφού η εφαρμογή χρησιμοποιεί επικοινωνία με σύνδεση).

```
   if ((fd_skt = socket (AF_UNIX, SOCK_STREAM, 0)) < 0) {
      printf("\n Socket creation error \n"); return -1; }
```

Παρατηρήστε πως στην προκειμένη περίπτωση η διεύθυνση sockaddr_un *sap που περνά ως όρισμα στη συνάρτηση bind για να συσχετιστεί με το socket που δημιουργήθηκε από τη συνάρτηση socket διαβιβάζεται ως όρισμα στη συνάρτηση run_server αφού πρώτα αρχικοποιηθεί μέσα στη συνάρτηση main.

```
   if (bind (fd_skt, (struct sockaddr *) sap, sizeof(*sap)) < 0) {
      perror("Bind"); exit(EXIT_FAILURE); }
```

Ο διακομιστής τίθεται σε καθεστώς αναμονής εισερχομένων αιτημάτων σύνδεσης καλώντας τη συνάρτηση listen. Η σταθερά SOMAXCONN ορίζεται στο αρχείο socket.h στην τιμή 128 αλλά γενικά τίθεται σε μικρότερη τιμή.

```
   if (listen (fd_skt, SOMAXCONN)<0) {
      perror("Listen"); exit(EXIT_FAILURE); }
```

Οι μακροεντολές FD_ZERO και FD_SET λειτουργούν με παρόμοιο τρόπο με τον οποίο λειτουργούν οι συναρτήσεις sigemptyset και sigaddset που είδαμε στην περίπτωση των σημάτων. Ειδικότερα, η μακροεντολή FD_ZERO αρχικοποιεί το σύνολο των file descriptors εκχωρώντας σε όλα τα bits την τιμή μηδέν, ενώ η FD_SET ενεργοποιεί το bit του συνόλου fd_set που αντιστοιχεί στον περιγραφέα αρχείου που δέχεται ως όρισμα.

```
   if (fd_skt > fd_hwm) fd_hwm = fd_skt;
   FD_ZERO (&set);
   FD_SET (fd_skt, &set);
```

Η διαδικασία που πραγματοποιείται στον κεντρικό βρόχο της συνάρτησης του διακομιστή, έχει ως εξής. Καταρχήν παρατηρήστε πως ο βρόχος είναι ατέρμων, δηλαδή εκτελείται επ' άπειρον και αυτό διότι ένας διακομιστής υποτίθεται πως είναι συνεχώς ενεργός, αναμένοντας αιτήματα σύνδεσης. Με άλλα λόγια, για να ολοκληρωθεί η διαδικασία θα πρέπει ο χρήστης να τερματίσει τη λειτουργία της συνάρτησης του διακομιστή πατώντας τον συνδυασμό πλήκτρων Ctrl-C.

```
  while (1) {
      read_set = set;
```

Παρατηρήστε πως η select στην προκειμένη περίπτωση, δέχεται μόνο ένα όρισμα τύπου fd_set το οποίο αντιστοιχεί σε διαδικασία ανάγνωσης (διότι ενδιαφέρεται μόνο να διαβάσει δεδομένα από τους πελάτες και όχι να στείλει δεδομένα σε αυτούς, ή να αναφέρει μηνύματα σφάλματος). Το πρώτο όρισμα της select σύμφωνα με τη θεωρία είναι ίσο με fd_skt+1.

```
if (select(fd_hwm+1, &read_set, NULL, NULL, NULL)==-1) {
   perror("Select"); exit(EXIT_FAILURE); }
```

Μέσα από έναν δεύτερο ένθετο βρόχο for, η συνάρτηση εξετάζει έναν προς έναν τους περιγραφείς αρχείων που έχουν τιμές από 0 έως fd_hwm.

```
for (fd = 0; fd <= fd_hwm; fd++)
```

Εάν συναντήσει περιγραφέα fd που ανήκει στο σύνολο read_set (οπότε η FD_ISSET επιστρέφει true)

```
    if (FD_ISSET(fd, &read_set)) {
```

ΚΑΙ αυτός ο περιγραφέας είναι ο fd_skt που επέστρεψε η socket και χρησιμοποιείται στην bind και στη listen ΤΟΤΕ

```
        if (fd == fd_skt) {
```

καλεί τη συνάρτηση accept με πρώτο όρισμα αυτόν τον περιγραφέα η οποία επιστρέφει τον περιγραφέα fd_client για το active socket μέσω του οποίου θα πραγματοποιηθεί η επικοινωνία με τον πελάτη (εάν η accept δεν λειτουργήσει με επιτυχία η εφαρμογή τερματίζεται με μήνυμα σφάλματος)

```
          if ((fd_client = accept(fd_skt, NULL, 0))<0) {
             perror("Select"); exit(EXIT_FAILURE); }
```

Ενεργοποιεί το bit του συνόλου fd_set που αντιστοιχεί στον περιγραφέα fd_client

```
          FD_SET (fd_client, &set);

          if (fd_client > fd_hwm) fd_hwm = fd_client; }

        else {
```

Εάν η read από το αρχείο με περιγραφέα fd αποτύχει και ως εκ τούτου η συνάρτηση επιστρέψει αρνητικό αριθμό, η διαδικασία τερματίζεται με μήνυμα σφάλματος.

```
          if ((nread = read(fd, buf, sizeof(buf)))<0) {
              perror("Read"); exit(EXIT_FAILURE); }
```

Εάν δεν υπάρχουν δεδομένα για ανάγνωση τότε η παραπάνω διαδικασία αναιρείται δηλαδή το bit που αντιστοιχεί σε αυτόν τον περιγραφέα τίθεται από την FD_CLR στη μηδενική τιμή, δηλαδή απενεργοποιείται ενώ η τιμή του fd_hwm μειώνεται κατά μία μονάδα.

```
        if (nread == 0) {
            FD_CLR (fd, &set);
            if (fd == fd_hwm) fd_hwm--;
            close(fd); }
```

Διαφορετικά, ο περιγραφέας διαβάζει το μήνυμα του πελάτη, το εκτυπώνει στην οθόνη και του απσοτέλει το μήνυμα GoodBye !!. Εάν η διαδικασία αποστολής (δηλαδή εγγραφής στο fd) αποτύχει, η διαδικασία τερματίζεται με μήνυμα σφάλματος.

```
          else {
             printf("Server got \"%s\"\n", buf);
             if (write(fd, "Goodbye!", 9)==-1) {
                perror("Write"); exit(EXIT_FAILURE); }}}}}

close(fd_skt);
close(fd_client);
```

```
   printf ("Server terminates\n");
   return 1; }
```

Κώδικας πελάτη

// Ο κώδικας του client

Η συνάρτηση του πελάτη run_client καλεί τη fork για να δημιουργήσει μία διεργασία και περιορίζεται στον κώδικα της θυγατρικής διεργασίας η οποία αντιστοιχεί στη συνθήκη pid = 0.

```
static int run_client(struct sockaddr_un *sap) {

   if (fork() == 0) {
      int fd_skt;
      char buf[100];
```

Αρχικά καλείται η συνάρτηση socket με τα ίδια ορίσματα με αυτά που κλήθηκε στη συνάρτηση run_server έτσι ώστε οι δύο διεργασίες να μπορέσουν να επικοινωνήσουν μεταξύ τους

```
      if ((fd_skt = socket(AF_UNIX, SOCK_STREAM, 0)) < 0) {
         printf("\n Socket creation error \n"); return -1; }
```

και στη συνέχεια καλείται επαναληπτικά η connect μέχρι τελικά η διεργασία πελάτης να καταφέρει να συνδεθεί στο διακομιστή (αυτό γίνεται διότι όπως σχολιάσαμε και σε άλλο παράδειγμα, υπάρχει περίπτωση όταν η θυγατρική διεργασία αιτηθεί σύνδεση, να μην έχει δημιουργηθεί ακόμη το άλλο άκρο της σύνδεσης στη διεργασία του διακομιστή και ως εκ τούτου η σύνδεση να μην είναι δυνατή).

```
      while (connect(fd_skt, (struct sockaddr *)sap, sizeof(*sap)) == -1) {
         if (errno == ENOENT) {
            sleep(1);
            continue; }
         else {
          perror("Message from connect [client]");
          exit(-2); }}
```

Μετά την αποκατάσταση της σύνδεσης, η θυγατρική διεργασία δημιουργεί και διαμορφώνει κατάλληλα ένα μήνυμα (που περιέχει το PID της) το οποίο στη συνέχεια στέλνει με τη συνάρτηση write στο διακομιστή χρησιμοποιώντας το fd_skt που επέστρεψε η socket.

```
      snprintf (buf, sizeof(buf), "Hello from %ld!", (long)getpid());
      if (write (fd_skt, buf, strlen(buf)+1) < 0) {
         perror("Write"); exit(EXIT_FAILURE); }
```

Στη συνέχεια διαβάζει με τη συνάρτηση read το μήνυμα που της έστειλε η συνάρτηση του διακομιστή το οποίο εκτυπώνει στην οθόνη της και κλείνει το socket της επικοινωνίας.

```
      if ((read(fd_skt, buf, sizeof(buf))) < 0) {
         perror("Read"); exit(EXIT_FAILURE); }
      printf("Client %d got %s\n", getpid(), buf);
      close(fd_skt);

   printf ("Client %d terminates\n", getpid());
   exit(EXIT_SUCCESS); }
   return 1; }
```

Η συνάρτηση main αρχικοποιεί τη διεύθυνση sa τύπου sockaddr_un στις τιμές SOCKETNAME και AF_UNIX. Στη συνέχεια μέσα από ένα βρόχο τεσσάρων επαναλήψεων καλεί τη συνάρτηση run_client για να δημιουργήσει τις τέσσερις θυγατρικές διεργασίες – πελάτες και στη συνέχεια μία φορά τη συνάρτηση run_server για να δημιουργήσει τη διεργασία του διακομιστή. Αμφότερες οι συναρτήσεις παίρνουν ως όρισμα ένα δείκτη στη δομή sa.

```
int main(void) {
   struct sockaddr_un sa;
   int nclient;
   (void)unlink(SOCKETNAME);
   strcpy(sa.sun_path, SOCKETNAME);
   sa.sun_family = AF_UNIX;
   for (nclient = 1; nclient <= 4; nclient++)
      run_client(&sa);
   run_server(&sa);
   exit(EXIT_SUCCESS); }
```

Παράδειγμα εξόδου της εφαρμογής ακολουθεί στη συνέχεια.

```
Server pid is 3082
Server got "Hello from 3086!"
Server got "Hello from 3085!"
Server got "Hello from 3087!"
Client 3086 got Goodbye!
Client 3086 terminates
Client 3087 got Goodbye!
Client 3085 got Goodbye!
Server got "Hello from 3084!"
Client 3087 terminates
Client 3085 terminates
Client 3084 got Goodbye!
Client 3084 terminates
```