ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Εισαγωγή στον προγραμματισμό των Sockets

στο UNIX

Γ. ΠΑΠΑΚΩΝΣΤΑΝΤΙΝΟΥ

Π.ΤΣΑΝΑΚΑΣ

Επιμέλεια: Π. Θεοδωρόπουλος
Σχήματα:Ι. Δροσίτης

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΕΧΟΜΕΝΑ i

I. ΕΙΣΑΓΩΓΗ ΣΤΗ ΧΡΗΣΗ ΤΟΥ ΜΗΧΑΝΙΣΜΟΥ ΤΩΝ SOCKETS ΣΤΟ UNIX……..3

1. ΓΕΝΙΚΑ 3

2. ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ ΓΙΑ ΤΗ ΧΡΗΣΗ ΤΩΝ SOCKETS 6

3. ΧΡΗΣΗ ΚΛΗΣΕΩΝ SOCKETS 9

3.1. Δημιουργία Socket 9
3.2. Ονομασία Socket 10
3.3. Σύνδεση 12
3.4. Επικοινωνία, Είσοδος και Έξοδος 15
3.5. Είσοδος/Έξοδος χωρίς αναμονή (Non blocking I/O) 15
3.6. Αναμονή εισόδου από πολλούς descriptors 16
3.7. Κλείσιμο socket 19

4. ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΕΞΥΠΗΡΕΤΗΤΗ (SERVER) 200

4.1. Παράδειγμα 1: Ένας απλός Εξυπηρετητής 20
4.2. Παράδειγμα 2: Εξυπηρετητής χωρίς αναμονή στην εξυπηρέτηση 21
4.3. Παράδειγμα 3: Εξυπηρετητής χωρίς αναμονή στην είσοδο/έξοδο 24
4.4. Παράδειγμα 4: Εξυπηρετητής χωρίς αναμονή Ε/Ε με χρήση σήματος SIGIO 26
4.5. Παράδειγμα 5: Εξυπηρετητής με δυνατότητα ελέγχου της λειτουργίας του 28

5. ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΠΕΛΑΤΗ (CLIENT) 32

5.1. Παράδειγμα 6: Πελάτης για τον Εξυπηρετητή Ώρας 32

6. ΧΡΗΣΗ DATAGRAM SOCKETS 34

II. ΒΙΒΛΙΟΓΡΑΦΙΑ 35

Ι. ΕΙΣΑΓΩΓΗ ΣΤΗ ΧΡΗΣΗ ΤΟΥ ΜΗΧΑΝΙΣΜΟΥ ΤΩΝ SOCKETS ΣΤΟ UNIX

1. ΓΕΝΙΚΑ

Σε ένα σύστημα που υποστηρίζει την πολυεπεξεργασία, όπως το UNIX, είναι απαραίτητη η επικοινωνία μεταξύ διεργασιών που τρέχουν παράλληλα, έτσι ώστε να επιτυγχάνεται συγχρονισμός καθώς και ανταλλαγή πληροφοριών, όταν αυτό είναι απαραίτητο. Γνωστοί τρόποι με τους οποίους επιτυγχάνεται αυτό είναι οι σηματοφορείς (semaphores), τα μηνύματα (messages) και οι σωληνώσεις (pipes). Οι παραπάνω τρόποι καθιστούν δυνατή την επικοινωνία μεταξύ διεργασιών με ένα σοβαρό περιορισμό: οι διεργασίες που επικοινωνούν πρέπει να εκτελούνται στο ίδιο μηχάνημα, ενώ επιπλέον στην περίπτωση των pipes πρέπει να έχουν κάποιο κοινό πρόγονο (π.χ. να είναι διεργασίες-παιδιά της ίδιας διεργασίας-γονέα). Τι γίνεται όμως στην περίπτωση που θέλουμε δύο εντελώς ανεξάρτητες διεργασίες που τρέχουν σε διαφορετικά υπολογιστικά συστήματα να επικοινωνήσουν; Για την επικοινωνία αυτή πρέπει να χρησιμοποιηθεί το επικοινωνιακό υποδίκτυο που ενώνει τις δύο μηχανές. Την επικοινωνία αυτή καθιστούν δυνατή τα sockets.

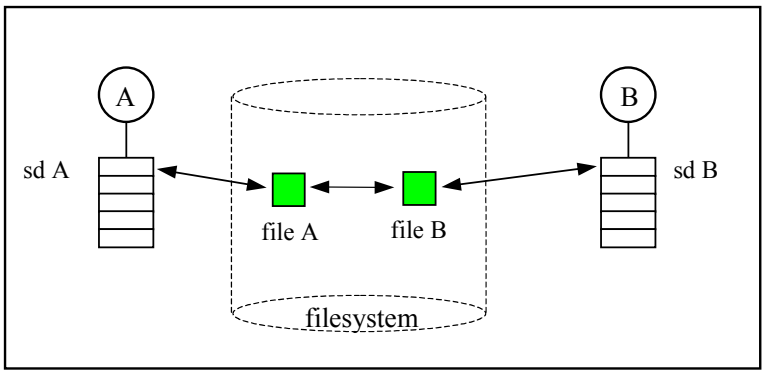
Ο μηχανισμός των sockets εμφανίσθηκε για πρώτη φορά στην έκδοση 4.2 BSD του Berkeley Unix αλλά στη συνέχεια γρήγορα υιοθετήθηκε από όλες τις παραλλαγές αυτού του λειτουργικού και αποτελεί σήμερα ένα standard για όλες τις εκδόσεις του δημοφιλούς συστήματος, ενώ έχει μεταφερθεί και σε άλλα λειτουργικά περιβάλλοντα π.χ. Winsock. Σκοπός του είναι η γενική επικοινωνία μεταξύ διεργασιών. Αυτή η επικοινωνία μπορεί να γίνεται ανάμεσα σε διεργασίες του ιδίου ή και διαφορετικών υπολογιστικών συστημάτων, καλύπτοντας έτσι το κενό της επικοινωνίας διεργασιών διαφορετικών χρηστών στο ίδιο σύστημα αλλά και διεργασιών που εκτελούνται σε διαφορετικά συστήματα και συνδέονται μεταξύ τους με δίκτυο. Ο μηχανισμός των sockets αποτελεί σήμερα το δημοφιλέστερο API (Application Programming Interface) προγραμματισμού και χρήσης του συνόλου των πρωτοκόλλων επικοινωνίας TCP/IP από τις εφαρμογές χρήστη.

Το socket ορίζεται ως το άκρο (endpoint) ενός καναλιού επικοινωνίας διεργασιών. Υπάρχει μια συγκεκριμένη διαδικασία δημιουργίας αυτού του καναλιού, μετά από αυτήν όμως οι διεργασίες μπορούν να επικοινωνούν με συνηθισμένες κλήσεις `read()` και `write()`. Η δημιουργία ενός socket δεν συνεπάγεται και τη δημιουργία του καναλιού. Για να επιτευχθεί κάτι τέτοιο, θα πρέπει δύο διεργασίες να δημιουργήσουν η κάθε μία ένα socket και έπειτα η μια από αυτές να περιμένει σύνδεση και η άλλη να τη ζητήσει. Με άλλα λόγια, η μια διεργασία αναλαμβάνει το ρόλο του εξυπηρετητή (server - αυτή που περιμένει τη σύνδεση) και η άλλη το ρόλο του πελάτη (client - αυτή που κάνει την αίτηση για σύνδεση).

Κάθε διεργασία χειρίζεται το socket μέσω ενός θετικού ακεραίου τον οποίο της επιστρέφει το σύστημα όταν δημιουργείται το socket. Ο ακέραιος αυτός, που ονομάζεται και socket descriptor, έχει παρόμοια λειτουργικότητα με ένα file descriptor, δεδομένου ότι επιτρέπεται να χρησιμοποιήσουμε τις κλήσεις του συστήματος αρχείων για να τον χειριστούμε.

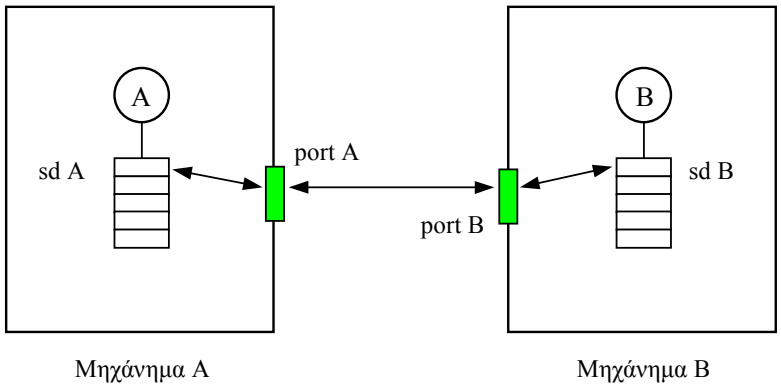
Είναι φανερό ότι είναι αναγκαίο τα sockets να έχουν κάποια διεύθυνση η οποία θα επιτρέπει την ταυτοποίησή τους. Τα sockets που δημιουργούνται από διαφορετικά προγράμματα χρησιμοποιούν "ονόματα" για να αναφερθούν μεταξύ τους και να γίνει η σύνδεση. Το πεδίο από το οποίο λαμβάνει όνομα το socket αποτελεί το **domain** ή **address family**. Δύο είναι τα domain που χρησιμοποιούνται από τις εφαρμογές για τη διευθυνσιοδότηση των sockets τους:

- **Unix domain (`AF_UNIX`)** στο οποίο κάθε socket ταυτοποιείται μέσω ενός ονόματος "αρχείου" (και επομένως πρέπει οι διεργασίες να έχουν πρόσβαση σε ένα κοινό σύστημα αρχείων). Κάθε κανάλι ταυτοποιείται μονοσήμαντα με την τριάδα {πρωτόκολλο / τοπικό αρχείο / απομακρυσμένο αρχείο}, όπου η λέξη *απομακρυσμένο* δηλώνει την άλλη διεργασία. Είναι φανερό ότι αυτός ο τρόπος δεν μπορεί να χρησιμοποιηθεί πάνω από το δίκτυο και δεν θα μας απασχολήσει στη συνέχεια.



Σχήμα 1: *Επικοινωνία στο UNIX domain*

- **Internel domain (`AF_INET`)** στο οποίο κάθε socket ταυτοποιείται από μια δυάδα που αποτελείται από την IP διεύθυνση (IP address) του συστήματος στο οποίο έχει δημιουργηθεί το socket και ένα θετικό ακέραιο ο οποίος ονομάζεται port number του socket και δηλώνει σε ποιά "θύρα" επικοινωνίας μέσα στο σύστημα αυτό έχει "συνδεθεί" το socket. Κάθε κανάλι επικοινωνίας ταυτοποιείται μονοσήμαντα με την πεντάδα {πρωτόκολλο / τοπική μηχανή / τοπικό port / απομακρυσμένη μηχανή / απομακρυσμένο port}.



Σχήμα 2: *Επικοινωνία στο Internet domain*

Πρέπει να παρατηρήσουμε ότι ενώ οι socket descriptors είναι τοπικοί για κάθε διεργασία (όπως ακριβώς και οι file descriptors), τα ονόματα των sockets είναι μοναδικά για κάθε μηχανή.

Όσον αφορά τη μετάδοση των δεδομένων πάνω στο κανάλι επικοινωνίας μετά τη δημιουργία του, υπάρχουν δύο τρόποι:

- **Stream communication** (**`SOCK_STREAM`**), όπου τα δεδομένα στέλνονται πάνω σε ήδη υπάρχον κανάλι επικοινωνίας (για αυτό και ο συγκεκριμένος τρόπος ονομάζεται **connection oriented**). Το πρωτόκολλο επικοινωνίας TCP μέσα στον πυρήνα του λειτουργικού συστήματος φροντίζει και αναλαμβάνει όλους τους ελέγχους για λάθη στη μετάδοση των δεδομένων καθώς και την επαναμετάδοση των πακέτων που χάθηκαν. Έτσι εξασφαλίζονται αξιόπιστα κανάλια επικοινωνίας στα οποία οι πληροφορίες φτάνουν με τη σωστή σειρά και πλήρεις, αν και με μικρότερη ταχύτητα. Αυτός ο τρόπος δεν κρατά τα όρια των μηνυμάτων που στέλνονται, όπως ακριβώς συμβαίνει και με τα pipes. Αυτό σημαίνει ότι κάθε ανάγνωση δεν επιστρέφει απαραίτητα τα δεδομένα μίας μόνο αποστολής δεδομένων, αλλά όσα δεδομένα είναι στον buffer του συστήματος από προηγούμενες αποστολές. Αυτόν τον τρόπο επικοινωνίας χρησιμοποιούν συνήθως εφαρμογές που έχουν μεταξύ τους μια καθορισμένη σχέση "πελάτη - εξυπηρετητή".

- **Datagram communication** (**`SOCK_DGRAM`**), όπου τα δεδομένα μεταδίδονται σε μικρά αυτόνομα "πακέτα" που ονομάζονται datagrams και το κανάλι επικοινωνίας δημιουργείται για κάθε μήνυμα ξεχωριστά (**connectionless**), ενώ η παραλαβή του δεν είναι εγγυημένη. Ο έλεγχος μετάδοσης με τη σωστή σειρά και η ευθύνη για τον κατακερματισμό (fragmentation) και την επανασυναρμολόγηση (reassembly) των δεδομένων ανήκουν στις εφαρμογές. Με τον τρόπο αυτό, που αντιστοιχεί στο πρωτόκολλο επικοινωνίας UDP, επιτυγχάνονται υψηλές ταχύτητες μετάδοσης για εφαρμογές που επικοινωνούν με βάση μικρά αυτόνομα "πακέτα" πληροφορίας χωρίς ανάγκη απόλυτης αξιοπιστίας. Σημειώνεται ότι ο τρόπος αυτός κρατά τα όρια των μηνυμάτων και επομένως κάθε ανάγνωση επιστρέφει τα δεδομένα μιας αποστολής.

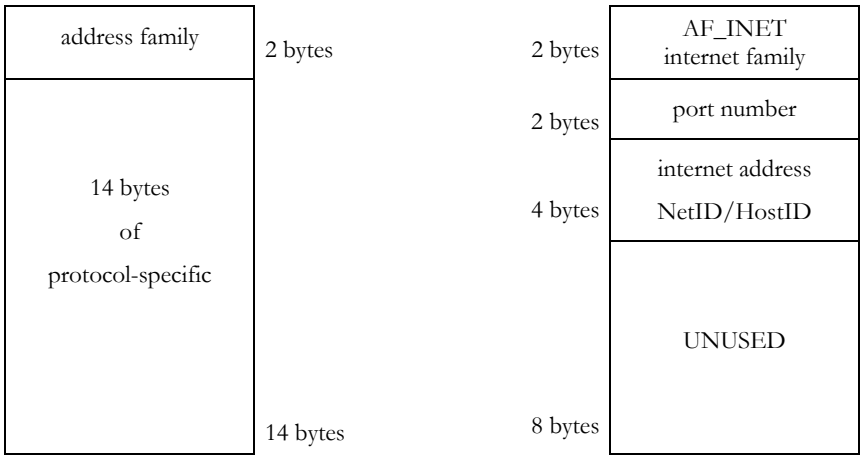
Ο δεύτερος τρόπος επικοινωνίας είναι πιο σαφώς πιο γρήγορος, αλλά αυξάνει την πολυπλοκότητα του προγράμματος, καθώς πρέπει να αναλάβει η εφαρμογή την επαναμετάδοση των πακέτων που χάθηκαν μέσω ενός μηχανισμού αναγνώρισης και timeout (βλ. παράρτημα). Εδώ θα ασχοληθούμε κυρίως με stream sockets στο internet domain, που είναι και ο πιο διαδεδομένος τρόπος υλοποίησης πολλών από τις υπηρεσίες που παρέχονται πάνω από δίκτυο (finger, e-mail, www).

Ακολουθεί περιγραφή του τρόπου χειρισμού του μηχανισμού μέσα από ρουτίνες της γλώσσας C. Οι ρουτίνες δεν είναι πλήρεις διότι δεν περιλαμβάνουν έλεγχο λαθών για τις τιμές επιστροφής των διαφόρων εντολών. Ωστόσο δίνουν μια εικόνα του τρόπου προγραμματισμού των sockets.

2. ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ ΓΙΑ ΤΗ ΧΡΗΣΗ ΤΩΝ SOCKETS

Στις επόμενες παραγράφους θα δούμε τις πρωτογενείς κλήσεις που υποστηρίζει το UNIX για τη χρήση των sockets. Σε πολλές από τις πρωτογενείς αυτές κλήσεις περνιούνται σαν παράμετροι δείκτες προς δομές τύπου `sockaddr`. Η δομή `sockaddr` είναι τελείως γενική και επιτρέπει στο UNIX να υποστηρίζει πολλά πρωτόκολλα επικοινωνιών όπως το XNS της Xerox και τα UNIX domain protocols. Στο σχήμα 3 φαίνεται η δομή `sockaddr` και η δομή `sockaddr_in` η οποία χρησιμοποιείται για τη διαχείριση πληροφοριών στα πρωτόκολλα internet. Ο τύπος `u_long` είναι ορισμένος σαν `unsigned long` και ο τύπος `u_short` σαν `unsigned short` μέσα στο αρχείο `<sys/types.h>`. Η σταθερά `AF_INET` είναι ορισμένη στο αρχείο `<sys/socket.h>` και χρησιμοποιείται για την αρχικοποίηση ενός socket σε κάποιο από τα πρωτόκολλα επικοινωνίας του internet. Όταν χρειάζεται να περαστεί σαν παράμετρος ένας δείκτης προς δομή τύπου `sockaddr_in` εκεί που μια πρωτογενής κλήση περιμένει δείκτη τύπου `sockaddr` κάνουμε casting, όπως φαίνεται και στο παράδειγμα που ακολουθεί:

```
Sockaddr    *addr;
sockaddr_in  host_addr;
        :
addr = (sockaddr *) &host_addr;
```




```
#include <sys/socket.h>

/*
 * Structure used by kernel
 * to store most addresses.
 */


struct sockaddr {
      u_short sa_family;
                /* address family */
      char    sa_data[14];
                /* up to 14 bytes of
                   direct address */
};
```

```
#include <netinet/in.h>

/*
 * Socket address, internet style.
 */

struct in_addr {
            u_long s_addr;
};
struct sockaddr_in {
      short          sin_family;
      u_short        sin_port;
      struct in_addr sin_addr;
      char           sin_zero[8];
};
```

Σχήμα 3: *Οι δομές sockaddr και sockaddr_in*

Το όνομα του μηχανήματος στο οποίο εκτελείται ένα πρόγραμμά μας μπορούμε να το βρούμε με την κλήση `gethostname()`, που έχει την παρακάτω μορφή:

```
int gethostname(char *name, int namelen);
```

Η παράμετρος name είναι ένας δείκτης σε πίνακα χαρακτήρων στον οποίο επιστρέφεται το όνομα του μηχανήματος στο οποίο εκτελείται η καλούσα διεργασία, όπως αυτό είναι γνωστό στο δίκτυο. Η δεύτερη παράμετρος καθορίζει το μέγιστο μήκος του πίνακα της πρώτης παραμέτρου. Η συνάρτηση τερματίζει αυτόματα τη συμβολοσειρά με ένα χαρακτήρα '\0'.

Πολλές φορές εμφανίζεται η ανάγκη να βρούμε τα port numbers για κάποιες συγκεκριμένες υπηρεσίες που προσφέρονται από το σύστημα, όπως για παράδειγμα του FTP. Πρέπει, δηλαδή, να θυμόμαστε ότι ο αριθμός θύρας για τον FTP server είναι 21; Πρέπει επίσης να θυμόμαστε την internet διεύθυνση του μηχανήματος στο οποίο τρέχει ο FTP server; Για την διευκόλυνση του προγραμματιστή υπάρχουν ευτυχώς μερικές συναρτήσεις βιβλιοθήκης του UNIX, που μας βοηθούν να ξεπεράσουμε κάποια τέτοια προβλήματα. Τα πρωτότυπά (prototypes) τους ορίζονται στο αρχείο <netdb.h> και φαίνονται παρακάτω:

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const char *addr, int len, int type);

struct servent *getservbyname(const char *name, const char *proto);
```

Η πρώτη παίρνει σαν όρισμα ένα string που περιέχει το όνομα της μηχανής και επιστρέφει ένα δείκτη προς μια δομή τύπου hostent(ry). Η δεύτερη παίρνει σαν πρώτο όρισμα ένα δείκτη σε μια δομή που περιέχει την διεύθυνση της μηχανής, όπως αυτή καθορίζεται από την τρίτη παράμετρο που δηλώνει την οικογένεια διευθύνσεων στην οποία ανήκει και επιστρέφει ένα δείκτη προς μια δομή τύπου hostent(ry). Η τρίτη παίρνει σαν ορίσματα δύο strings από τα οποία το πρώτο περιέχει το όνομα της υπηρεσίας (π.χ. FTP, TELNET) της οποίας ζητάμε τον αριθμό θύρας, ενώ το δεύτερο όρισμα μπορεί να είναι NULL ή να περιέχει το όνομα κάποιου πρωτοκόλλου (π.χ. TCP, UDP αφού κάποιες υπηρεσίες είναι διαθέσιμες είτε χρησιμοποιείται connection-oriented είτε connectionless πρωτόκολλο). Και στις δύο περιπτώσεις η συνάρτηση επιστρέφει ένα δείκτη προς μια δομή τύπου servent(ry). Οι δομές hostent και servent είναι οι παρακάτω:

```
struct  hostent {
        char    *h_name;         /* official name of host */
        char   **h_aliases;      /* alias list */
        int     h_addrtype;      /* host address type */
        int     h_length;        /* length of address */
        char   **h_addr_list;    /* list of addresses from name server */
#define h_addr  h_addr_list[0]   /* address, for backward compatiblity */
};

struct  servent {
        char    *s_name;         /* official service name */
        char   **s_aliases;      /* alias list */
        int     s_port;          /* port # */
        char    *s_proto;        /* protocol to use */
};
```

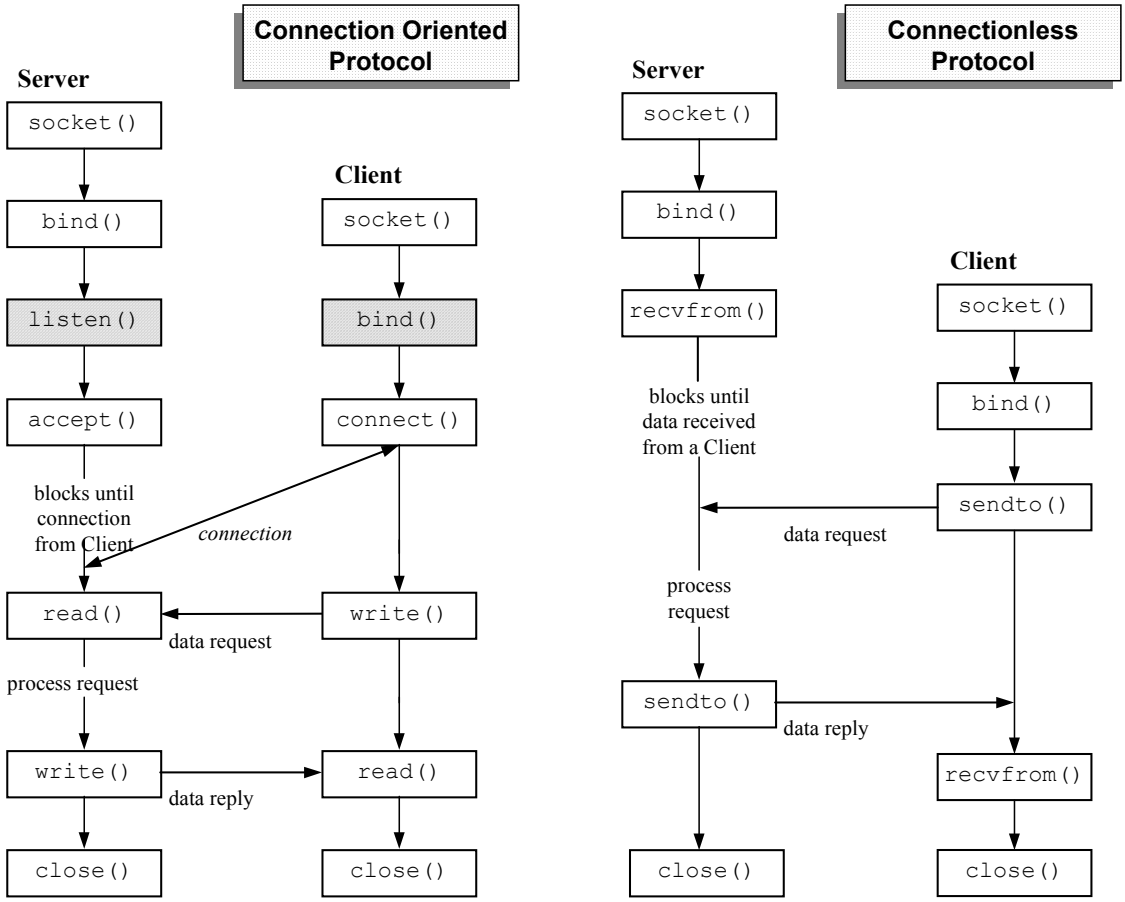
Στη δομή hostent, το πεδίο h_name περιέχει το όνομα του μηχανήματος στο δίκτυο, το h_addrtype έχει την τιμή AF_INET για τα πρωτόκολλα internet και για τον ίδιο λόγο το h_length έχει πάντα την τιμή 4. Αυτό όμως που μας ενδιαφέρει περισσότερο είναι το πεδίο h_addr_list το οποίο είναι ένας πίνακας από δείκτες όχι σε χαρακτήρες αλλά σε δομές τύπου in_addr (Σχήμα 3). Ο αριθμός των στοιχείων του πίνακα δεν είναι γνωστός, αλλά γνωρίζουμε ότι το τελευταίο του στοιχείο είναι ένας δείκτης NULL. Όπως θα έχει γίνει ήδη κατανοητό αυτές είναι οι internet διευθύνσεις όλων των δικτύων στα οποία ίσως συμμετέχει το μηχάνημα. Επειδή στη γενικότερη περίπτωση η μηχανή θα ανήκει σε ένα μόνο δίκτυο έχει ορισθεί ένα macro ώστε να είναι ευκολότερη η προσπέλαση του πρώτου (και μοναδικού στην περίπτωση αυτή) στοιχείου του πίνακα h_addr_list. Το μοναδικό σημείο που ίσως έμεινε αδιευκρίνιστο είναι το όνομα που περνιέται σαν παράμετρος στην gethostbyname(). Συνηθίζεται να δίνονται ονόματα στις διάφορες μηχανές ενός δικτύου. Τα ονόματα αυτά υπάρχουν συνήθως σε μια βάση δεδομένων στο αρχείο /etc/hosts. Η gethostbyname() κοιτάζει τις καταχωρήσεις στη βάση και επιστρέφει τα ζητούμενα δεδομένα μέσω της δομής hostent.

Τα πράγματα είναι εντελώς ανάλογα στη δομή `servent` όπου το πεδίο `s_name` είναι το όνομα της υπηρεσίας, το `s_port` είναι το port number με το οποίο συνδέεται η υπηρεσία και το `s_proto` είναι το πρωτόκολλο επικοινωνίας κάτω από το οποίο παρέχεται η συγκεκριμένη υπηρεσία. Οι πληροφορίες οι οποίες επιστρέφει η συνάρτηση `getservbyname()` είναι αποθηκευμένες στο αρχείο `/etc/services`.

Όταν δημιουργούμε κάποια καινούργια υπηρεσία, η οποία δεν υπάρχει στο αρχείο `/etc/services` τότε πρέπει να γνωστοποιήσουμε τον αριθμό θύρας με τον οποίο ταυτοποιείται η παροχή της υπηρεσίας μας, αλλιώς δεν θα είναι σε θέση να προσπελασθεί από κανένα χρήστη.

3. ΧΡΗΣΗ ΚΛΗΣΕΩΝ SOCKETS

Όπως είπαμε και πιο πριν, η σύνδεση δύο διεργασιών είναι μια μη συμμετρική διαδικασία, καθώς η μία περιμένει συνδέσεις (server) ενώ η άλλη κάνει αίτηση για σύνδεση. Στο σημείο αυτό θα εξετάσουμε τα διάφορα στάδια για την δημιουργία των sockets σε μία διεργασία. Ο κύκλος ζωής ενός τέτοιου socket μπορεί να περιγραφεί από το παρακάτω σχήμα, το οποίο είναι αρκετά γενικό και η αναλυτική εξήγησή του αποτελεί το αντικείμενο των επομένων παραγράφων.



Σχήμα 4: *Ο κύκλος ζωής ενός socket στο UNIX*

Στη συνέχεια παρουσιάζονται οι κλήσεις συστήματος του UNIX που υλοποιούν τα παραπάνω στάδια. Σημειώνεται ότι όλες οι κλήσεις επιστρέφουν -1 σε περίπτωση λάθους και θέτουν κατάλληλα την μεταβλητή `errno`.

3.1. Δημιουργία Socket

Η αρχικοποίηση ενός socket περιλαμβάνει καταρχήν τη δημιουργία του με μια κλήση `socket()`:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Η παράμετρος `domain` μπορεί να είναι `AF_INET` ή `AF_UNIX` και προσδιορίζει την οικογένεια διευθύνσεων του socket. Η παράμετρος `type` προσδιορίζει τον τρόπο μετάδοσης δεδομένων για το socket και μπορεί να είναι `SOCK_STREAM` ή `SOCK_DGRAM`. Τέλος, η παράμετρος `protocol` προσδιορίζει ένα πρωτόκολλο το οποίο θα υλοποιήσει την επικοινωνία πάνω στο κανάλι. Η παράμετρος αυτή μπορεί να προκύψει από μια κλήση στην `getprotoent()` για να διαλέξουμε ένα συγκεκριμένο πρωτόκολλο ή μπορεί να είναι 0 για να κάνει το σύστημα την επιλογή. Αν όλα πάνε καλά και η κλήση επιτύχει δημιουργείται ένα socket και επιστρέφεται στο πρόγραμμα ένας ακέραιος socket descriptor που έχει ίδια χρήση με τον file descriptor που επιστρέφει η κλήση `open()` του συστήματος αρχείων: μέσω αυτού του descriptor προσπελαύνεται στη συνέχεια το socket. Ωστόσο στην αρχική αυτή φάση το socket δεν έχει διεύθυνση. Αυτό γίνεται στη συνέχεια με τη χρήση της κλήσης `bind()`.

Ο τρόπος λοιπόν που θα χρησιμοποιήσουμε την `socket` είναι:

```
int sd;

sd = socket(AF_INET, SOCK_STREAM, 0);
```

3.2. Ονομασία Socket

Με χρήση της κλήσης `bind()` δίνεται διεύθυνση σε ένα socket:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sd, const struct sockaddr *addr, int addrlen);
```

Η πρώτη παράμετρος `sd` προσδιορίζει τον socket descriptor που έχουμε δημιουργήσει με κλήση στην `socket()`. Η τρίτη παράμετρος είναι το μέγεθος της δομής `struct sockaddr_in` η οποία περνάει με δείκτη ως δεύτερη παράμετρος και η οποία έχει παρουσιασθεί στην παράγραφο 2. Τα πεδία της δομής που μας ενδιαφέρουν είναι τα `sin_family`, `sin_port`, `sin_addr`.

Το πρώτο πεδίο `sin_family` πρέπει να είναι ίδιο με την παράμετρο `domain` κατά την δημιουργία του socket. Το πεδίο `sin_port` καθορίζει το port number του socket και δεν πρέπει να χρησιμοποιείται από κάποιο άλλο socket. Οι αριθμοί από 1 ως 1023 είναι δεσμευμένοι για υπηρεσίες του συστήματος (κυρίως εξυπηρετητές συστήματος) και συνήθως χρησιμοποιούνται οι αριθμοί πάνω από το 5000 για τα προγράμματα των χρηστών (εξυπηρετητές και άλλες εφαρμογές χρηστών). Μπορεί όμως να έχει την τιμή 0, οπότε διαλέγει το σύστημα ένα ελεύθερο port (> 1024).

Το τελευταίο από τα πεδία είναι η internet διεύθυνση της μηχανής στην οποία θα δέχεται συνδέσεις το socket. Αν π.χ. μία μηχανή είναι συνδεδεμένη ταυτόχρονα σε περισσότερα από ένα δίκτυα, τότε το socket μπορεί να προσδιορίσει από ποιο δίκτυο μόνο θα δέχεται συνδέσεις. Συνήθως όμως χρησιμοποιείται η τιμή `INADDR_ANY`, που σημαίνει ότι δεχόμαστε συνδέσεις από παντού.

Οι δύο τελευταίες παράμετροι είναι αναγκαίο να έχουν μια μορφή η οποία θα είναι αποδεκτή από οποιαδήποτε μηχανή συνδέεται σε δίκτυο, ώστε να μη υπάρχουν ασυμβατότητες. Για τον σκοπό αυτό υπάρχουν οι συναρτήσεις βιβλιοθήκης `htonl()` (host to network - long) και `htons()` (host to network - short) οι οποίες μετατρέπουν το όρισμά τους (`long` και `short` αντίστοιχα) στην γενικά αποδεκτή μορφή. Υπάρχουν και οι αντίστροφες συναρτήσεις `ntohl()` (network to host - long) και `ntohs()` (network to host - short) οι οποίες μετατρέπουν το όρισμά τους (`long` και `short` αντίστοιχα) από την γενικά αποδεκτή μορφή στην αναπαράσταση που χρησιμοποιεί εσωτερικά η συγκεκριμένη μηχανή. Ο τρόπος λοιπόν που θα χρησιμοποιηθεί η `bind()` είναι:

```
int sd=socket(AF_INET, SOCK_STREAM, 0);
…
struct sockaddr_in sin;
sin.sin_family      = AF_INET;
sin.sin_port        = htons(0);         /* Let the system choose */
sin.sin_addr.s_addr= htonl(INADDR_ANY);

bind(sd, &sin, sizeof(sin));
```

Αν πρόκειται για διεργασία-πελάτη είναι δυνατό να παραληφθούν όλα τα βήματα που αφορούν στη κλήση `bind()`, καθώς το λειτουργικό θα δεσμεύσει αυτόματα κάποιο ελεύθερο port number του συστήματος στην πρώτη μεταγενέστερη κλήση που θα γίνει για το συγκεκριμένο socket descriptor.

Ακολουθεί ένα πιο ολοκληρωμένο παράδειγμα, όπου στο socket δίνεται η IP διεύθυνση του τρέχοντος συστήματος και port number το 5678:

```
#define PORTNUM      5678
          /* Με PORTNUM 0 το σύστημα θα διαλέξει ένα ελεύθερο port number ανάμεσα στο 1024
          και το 5000 και θα το αντιστοιχίσει στο socket κατά την κλήση bind() */

struct sockaddr_in sa;          /* Η διεύθυνση του socket */
char   host[MAXHOSTNAME+1];     /* Το όνομα του συστήματος */
struct hostent *hostp;          / * Η διεύθυνση του συστήματος */

bzero(&sa, sizeof(struct sockaddr_in));
          /* Μηδενίζονται τα περιεχόμενα της διεύθυνσης του socket */
(void) gethostname(host, MAXHOSTNAME);
          /* Ποιό είναι το όνομα του παρόντος συστήματος. Αν δεν ενδιαφέρει το παρόν αλλά
          κάποιο άλλο σύστημα τότε η κλήση αυτή παραλείπεται και χρησιμοποιείται μόνο η
          επόμενη, όπου βέβαια host είναι το όνομα του άλλου συστήματος */
hostp = gethostbyname(host);
          /* Ποιά είναι η διεύθυνση του συστήματος host */
          /* Στη συνέχεια ορίζεται η διεύθυνση του socket σε 3 βήματα */
sa.sin_family = AF_INET;                    /* Internet */
sa.sin_port   = htons((u_short) PORTNUM);          /* Port number */
          /* Η κλήση htons() και η συγγενής της htonl() μετατρέπει το short και long
          αντίστοιχα όρισμα σύμφωνα με το "byte ordering" που χρησιμοποιείται στο δίκτυο ώστε
          να αντιμετωπίζονται τυχόν προβλήματα που προξενούνται από το "endianess" των
          συστημάτων. Οι αντίστροφες κλήσεις είναι οι ntohs() και η ntohl() */
bcopy(hostp->h_addr, &sa.sin_addr, hostp->h_length);
          /* Internet address του συστήματος. Είναι επίσης δυνατό, αν ενδιαφέρει το παρόν
          σύστημα να αποφευχθούν οι κλήσεις gethostname() και gethostbyname() και να
          χρησιμοποιηθεί η μορφή
          sa.sin_addr.s_addr = htonl(INADDR_ANY);
          που σημαίνει "IP διεύθυνση του socket είναι οποιαδήποτε διεύθυνση αντιστοιχεί στο παρόν
          σύστημα" */
(void) bind(sd, (struct sockaddr *)&sa, sizeof(sa));
          /* Με την κλήση αυτή το socket αποκτά διεύθυνση */
```

Είναι πολλές φορές συνηθισμένο κατά τη φάση της ανάπτυξης ενός προγράμματος εξυπηρετητή το φαινόμενο η διεργασία-εξυπηρετητής να έχει ανώμαλο τερματισμό. Στην περίπτωση αυτή τα sockets, που η διεργασία είχε δημιουργήσει, παραμένουν σε μια κατάσταση timeout από το λειτουργικό σύστημα. Έτσι παρόλου που κανένας δεν χρησιμοποιεί το συγκεκριμένο port number, όταν προσπαθήσουμε να επανεκτελέσουμε το πρόγραμμα-εξυπηρετητή αυτό τερματίζεται, καθώς αποτυγχάνει η κλήση της `bind()` με το διαγνωστικό μήνυμα: "Address already in use". Το πρόβλημα αυτό λύνεται με τη χρήση της κλήσης `setsockopt()` που έχει την παρακάτω μορφή:

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int sd, int level, int optname, void *optval, int *optlen);
int setsockopt(int sd, int level, int optname, void *optval, int optlen);
```

Η παράμετρος `sd` δηλώνει το socket στο οποίο θέλουμε να επέμβουμε ενώ η παράμετρος `level` έχει για sockets πάντα την τιμή `SOL_SOCKET`. Η κλήση `setsockopt()` είναι αρκετά γενική και μπορεί να καθορίσει πληθώρα χαρακτηριστικών των sockets. Η κλήση της πρέπει να γίνει μετά την κλήση της

socket() αλλά πριν την κλήση της bind() και ένα παράδειγμα χρήσης της, το οποίο μας βοηθά να ξεπεράσουμε το παραπάνω πρόβλημα, δίνεται παρακάτω:

```
int  enable=1;

setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int));
```

3.3. Σύνδεση

Μια διεργασία-πελάτης μπορεί να ζητήσει σύνδεση με κάποιον εξυπηρετητή με χρήση της κλήσης connect():

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int sd, struct sockaddr *addr, int addrlen);
```

Η σημασία των ορισμάτων είναι εντελώς ανάλογη με την περίπτωση της bind(). Έτσι το πρώτο όρισμα είναι ο socket descriptor που έχουμε δημιουργήσει (και με τον οποίο θα έχουμε πρόσβαση στο κανάλι επικοινωνίας), το δεύτερο όρισμα προσδιορίζει το όνομα του socket του εξυπηρετητή και το τρίτο όρισμα είναι το μέγεθος του δευτέρου ορίσματος σε bytes. Η περιγραφή που κάναμε για την bind() ισχύει και εδώ με την διαφορά ότι το addr.sin_addr.s_addr πρέπει να περιέχει την διεύθυνση της μηχανής στην οποία τρέχει ο server (στην περίπτωση της bind() περιείχε την τιμή INADDR_ANY) και το addr.sin_port το port number που ακούει ο server. Για να βρούμε την διεύθυνση μιας μηχανής από το όνομά της χρησιμοποιούμε την συνάρτηση βιβλιοθήκης gethostbyname(), η οποία παίρνει ως όρισμα το όνομα της μηχανής και επιστρέφει ένα δείκτη σε δομή hostent, την δήλωση της οποίας επαναλαμβάνουμε παρακάτω:

```
struct  hostent {
        char    *h_name;                /* official name of host */
        char    **h_aliases;            /* alias list */
        int     h_addrtype;             /* host address type */
        int     h_length;               /* length of address */
        char    **h_addr_list;          /* list of addresses from name server */
#define h_addr  h_addr_list[0]          /* address, for backward compatiblity */
};
```

Το πεδίο h_addr θα περιέχει την διεύθυνση της μηχανής σε network byte order (δεν χρειάζεται δηλ. να χρησιμοποιήσουμε την htonl() όπως κάναμε στην bind()).

Αν λοιπόν, η διεργασία-πελάτης θέλει να συνδεθεί με μια διεργασία-εξυπηρετητή που βρίσκεται σε κάποιο σύστημα host και κάποιο port number portnum, θα πρέπει να χρησιμοποιήσει τον ακόλουθο τρόπο:

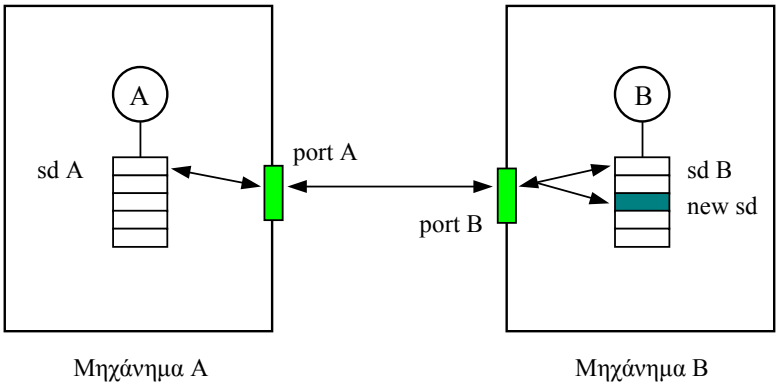
```
hostp = gethostbyname(host);
sa.sin_family = AF_INET;
sa.sin_port   = htons((u_short) portnum);
bcopy(hostp->h_addr, &sa.sin_addr, hostp->h_length);
          /* Σε αυτό το σημείο στο sa υπάρχει η πλήρης διεύθυνση του ζητούμενου εξυπηρετητή,
          οπότε με χρήση της connect() ζητείται σύνδεση */
(void) connect(sd, &sa, sizeof(sa));  /* Κλήση σύνδεσης */
          /* Στο σημείο αυτό -αν βέβαια η connect() έχει επιτύχει- το socket sd του πελάτη έχει
          συνδεθεί με ένα αξιόπιστο κανάλι επικοινωνίας με το socket που περιγράφεται στο sa */
```

Αφού μια διεργασία-εξυπηρετητής διευθυνσιοδοτήσει το socket της με χρήση της bind() μπορεί να δεχτεί κλήσεις σύνδεσης με χρήση της κλήσης accept():

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int sd, struct sockaddr *addr, int *addrlen);
```

Με την κλήση αυτή γίνεται αποδοχή κάποιας αίτησης σύνδεσης που περιμένει στην ουρά του socket descriptor `sd`. Αν η ουρά είναι άδεια, τότε η κλήση μπλοκάρει την καλούσα διεργασία μέχρι να εμφανιστεί κάποια αίτηση. Μόλις γίνει η αποδοχή, τα πεδία της δεύτερης παραμέτρου περιέχουν πληροφορίες για το socket που έκανε την αίτηση σύνδεσης (δηλ. port number και IP διεύθυνση) και επιστρέφεται ένας καινούριος socket descriptor, ο οποίος αποτελεί το άκρο του καινούργιου καναλιού που δημιουργήθηκε.



Σχήμα 5: *Δημιουργία καναλιού επικοινωνίας μέσω της κλήσης* `accept()`

Μπορούμε να παρατηρήσουμε ότι έχουμε πολύπλεξη στο επίπεδο του πρωτοκόλλου. Ενώ, δηλαδή, όλα τα sockets "βρίσκονται" στο ίδιο port, τα κανάλια επικοινωνίας δεν συγχέονται μεταξύ τους αφού όπως είδαμε ταυτοποιούνται και από το port number της διεργασίας του άλλου (απομακρυσμένου) άκρου, το οποίο είναι σίγουρα μοναδικό στο μοντέλο client-server.

Παράδειγμα χρήσης της κλήσης `accept()` σε κώδικα εξυπηρετητή είναι το ακόλουθο:

```
struct sockadd_in in_sa;
          /* Η διεύθυνση του socket με το οποίο έγινε σύνδεση */
int  new_sd, len;

len = sizeof(in_sa);
new_sd = accept(sd, &in_sa, &len);      /* Αποδοχή κλήσης σύνδεσης */
          /* Ο εξυπηρετητής μπλοκάρεται στην accept() και περιμένει κλήσεις σύνδεσης που
          αντιστοιχούν σε connect() από την πλευρά του πελάτη. Μόλις γίνει αυτό το
          connect() και φτάσει αυτή η κλήση σύνδεσης, η accept() δημιουργεί ένα νέο socket,
          new_sd, και το συνδέει με το socket του πελάτη, οπότε δημιουργείται το κανάλι
          επικοινωνίας new_sd (εξυπηρετητή) ↔sd (πελάτη). Ταυτόχρονα το αρχικό socket sd του
          εξυπηρετητή μένει ελεύθερο για νέο συνδυασμό accept() (εξυπηρετητή) - connect()
          (πελάτη) ώστε πολλοί πελάτες να μπορούν να συνδεθούν στον ίδιο εξυπηρετητή. Για όλους
          αυτούς τους πελάτες ο εξυπηρετητής δέχεται κλήσεις σύνδεσης στο socket του sd και για
          κάθε έναν τους δημιουργεί νέο socket new_sd που συνδέεται με το αντίστοιχο του πελάτη
          */
```

Από την ώρα που η `accept()` επιστρέφει με δημιουργημένο το νέο socket και εγκατεστημένο το κανάλι επικοινωνίας με τον πελάτη, μέχρι να ξανακληθεί η `accept()`, ο εξυπηρετητής δεν μπορεί να δεχτεί άλλες κλήσεις. Πρέπει δηλ. ο εξυπηρετητής να βρίσκεται "μέσα" στην `accept()` για να είναι δυνατή η αποδοχή κλήσεων σύνδεσης. Αυτές οι κλήσεις που γίνονται όσο ο εξυπηρετητής βρίσκεται εκτός της `accept()` τοποθετούνται σε μια ουρά ώστε να είναι διαθέσιμες όταν ξανακληθεί η `accept()`. Το default μήκος της ουράς είναι 5 κλήσεις σύνδεσης αλλά με χρήση της κλήσης `listen()` μπορεί να αλλάξει.

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int sd, int backlog);
```

Η δεύτερη παράμετρος `backlog` προσδιορίζει το μήκος της ουράς συνδέσεων για τον socket descriptor `sd` και καθορίζει ότι θα δέχεται συνδέσεις. Το λειτουργικό σύστημα επιβάλλει ένα μέγιστο στο μήκος της ουράς συνδέσεων. Αν η παράμετρος του μήκους είναι μεγαλύτερη από αυτό, το μήκος της ουράς τίθεται χωρίς προειδοποίηση σε αυτό το μέγιστο.

Ουσιαστικά η κλήση αρχικοποιεί την ουρά συνδέσεων και οποιεσδήποτε αιτήσεις συνδέσεων γίνουν (ενώ ο εξυπηρετητής είναι απασχολημένος) θα παραμένουν στην ουρά. Αν η ουρά γεμίσει τότε οι καινούριες αιτήσεις για σύνδεση θα χαθούν. Για παράδειγμα η παρακάτω κλήση:

```
listen(sd, num);
```

ορίζει το μήκος της ουράς σε `num` αναμένουσες κλήσεις σύνδεσης.

Με τον τρόπο αυτό για να δημιουργηθεί το κανάλι επικοινωνίας ο πελάτης κάνει:

```
socket();
bind();          /* Προαιρετικό */
connect();
```

και ο εξυπηρετητής κάνει:

```
socket();
bind();
listen();        /* Προαιρετικό */
for ( ; ; )
{
    accept();
    ...    /* Επεξεργασία της αίτησης, π.χ. fork() */
}
```

Ο συνήθης τρόπος επεξεργασίας της αίτησης από τον εξυπηρετητή περιλαμβάνει ένα `fork()`, ώστε ο πελάτης να επικοινωνεί με ένα "αφοσιωμένο" αντίγραφο του εξυπηρετητή, ενώ ο "αρχικός" εξυπηρετητής μπορεί να δέχεται νέες κλήσεις σύνδεσης. Στην περίπτωση που χρησιμοποιείται `fork()` θα πρέπει να αποφεύγεται η δημιουργία "zombie-processes". Οι διεργασίες αυτές είναι διεργασίες που τελείωσαν αλλά κανένας δεν έκανε `wait()` για αυτές. Έχουν ελευθερώσει όλη την μνήμη που κατείχαν αλλά παραμένουν στον πίνακα διεργασιών. Zombie μπορεί λοιπόν να δημιουργηθεί αν ο εξυπηρετητής δημιουργήσει ένα παιδί για να "ασχοληθεί" με κάποιο πελάτη αλλά όταν ο πελάτης και το παιδί τελειώσει ο γονέας του δεν κάνει `wait()`. Για να αποφευχθεί αυτό χρειάζεται ένα τμήμα κώδικα στο οποίο ο γονέας θα καθορίζει πως σε περίπτωση σήματος SIGCHLD το οποίο σηματοδοτεί το τέλος κάποιου παιδιού θα γίνεται αυτόματα `wait()`. Ο κώδικας αυτός έχει περίπου ως εξής:

```
void  wait_for_child();

signal(SIGCHLD, wait_for_child);

void wait_for_child()
{
     (void) wait((int *) 0);
     return;
}
```

Πάντως δεν είναι αναγκαίο να γίνεται `fork()` κάθε φορά που γίνεται `accept()`. Μπορεί η εξυπηρέτηση του παιδιού να γίνεται άμεσα από τη διεργασία του εξυπηρετητή, αν διασφαλισθεί πως αυτό θα γίνει γρήγορα, καθόσον όσο ο εξυπηρετητής βρίσκεται εκτός της `accept()` οι κλήσεις σύνδεσης των άλλων πελατών αναμένουν. Αν μάλιστα το μήκος της ουράς, που μπορεί να καθορίσθηκε με τη `listen()` ή να παραμένει στην default τιμή 5, ξεπεραστεί, τότε οι αιτήσεις των πελατών, όπως ήδη αναφέραμε, απορρίπτονται.

3.4. Επικοινωνία, Είσοδος και Έξοδος

Όπως αναφέραμε, ο socket descriptor έχει την ίδια λειτουργικότητα με τον file descriptor. Έτσι, όταν το κανάλι επικοινωνίας πελάτη-εξυπηρετητή εγκατασταθεί, οι δύο διεργασίες μπορούν να επικοινωνήσουν με χρήση των κλήσεων `read()` και `write()` προκειμένου να διαβάσουν ή να γράψουν από το socket, όπου για τον πελάτη θα χρησιμοποιείται αντί για file descriptor ο socket descriptor `sd`, ενώ για τον εξυπηρετητή θα χρησιμοποιείται ο socket descriptor `new_sd` που προέρχεται από την `accept()`.

Μπορούμε βέβαια να χρησιμοποιήσουμε και την συνάρτηση `fdopen()` της βιβλιοθήκης για να συσχετίσουμε τον socket descriptor με ένα IO stream όπως ακριβώς είναι το `stdio` και να χρησιμοποιήσουμε τις `fprintf()` και `fscanf()` για είσοδο και έξοδο. Σε αυτήν την περίπτωση θα πρέπει να δώσουμε προσοχή στο buffering των IO streams που προσφέρει η βιβλιοθήκη και το οποίο μπορεί να επηρεάσει την μετάδοση των δεδομένων. Ίσως θα ήταν λοιπόν χρήσιμο να χρησιμοποιηθεί η συνάρτηση `setbuf()` για να απενεργοποιήσουμε το buffering ή η `fflush()` για να αδειάζουμε τον buffer.

Υπάρχει, ωστόσο, μια σημαντική διαφορά ανάμεσα στην επικοινωνία αυτή και στην συνήθη ανάγνωση και εγγραφή από το σύστημα αρχείων: τόσο το `write()` όσο και το `read()` είναι δυνατό να μην γράψουν/διαβάσουν όσους χαρακτήρες τους ζητηθούν. Αυτό διότι είναι δυνατό να έχει γεμίσει ο buffer του δικτύου (στην περίπτωση του `write()`) ή να μην έχουν "έλθει" πάνω από το δίκτυο όλα τα δεδομένα (για το `read()`). Για το λόγο αυτό θα πρέπει η κλήση να "επιμένει" ωσότου γράψει/διαβάσει όλα τα ζητούμενα δεδομένα. Για παράδειγμα για το `read()` θα είναι περίπου:

```
int  sread( int sd, char *buf, int count )
{
     int   n, total = 0;
     char *p = buf;

     while ( total < count )
     {
          n = read(sd, p, count - total);
          if ( n < 0 )     /* Λάθος */
               return(-1);
          total += n;
          p += n;          /* Διάβασμα στο ελεύθερο τμήμα του buffer */
     }
     return(total);
}
```

Ολοκληρωμένα παραδείγματα χρήσης των παραπάνω κλήσεων δίνονται στις παραγράφους 4.1 και 4.2.

3.5. Είσοδος/Έξοδος χωρίς αναμονή (Non blocking I/O)

Όπως έχουμε ήδη αναφέρει η `accept()` μπλοκάρει την καλούσα διεργασία μέχρι να έρθει κάποια σύνδεση. Επίσης, όταν προσπαθούμε να διαβάσουμε από ένα socket στο οποίο δεν υπάρχουν δεδομένα, πάλι μπλοκάρεται η καλούσα διεργασία. Αυτό που θέλουμε είναι κάποιος τρόπος να ελέγχουμε αν υπάρχουν δεδομένα και αν όχι να εκτελούμε κάποια άλλη ενέργεια αντί να περιμένουμε. Αν θέλουμε ο έλεγχος να γίνεται με σύγχρονο τρόπο (σε καθορισμένο σημείο του προγράμματος), τότε χρησιμοποιούμε την `fcntl()` για να θέσουμε το socket σε non blocking mode ως εξής:

```
fcntl(sd, F_SETFL, FNDELAY);
```

Αν δεν υπάρχουν δεδομένα για διάβασμα από το socket τότε η `read()` θα επιστρέψει -1 και θα θέσει την μεταβλητή `errno` σε `EWOULDBLOCK`. Με τον ίδιο ακριβώς τρόπο συμπεριφέρεται και η `accept()` όταν δεν υπάρχει σύνδεση. Ο άλλος τρόπος (ασύγχρονος) είναι να στέλνεται το σήμα `SIGIO` στην διεργασία κάθε φορά που υπάρχουν δεδομένα για ανάγνωση στο socket. Πρέπει δηλ. μέσω της `signal()` να φροντίσουμε να εκτελείται η κατάλληλη συνάρτηση που θα χειρίζεται τα καινούργια δεδομένα. Για να λάβουμε το σήμα `SIGIO` θα πρέπει πρώτα να θέσουμε τον αριθμό διεργασίας του socket (στην οποία στέλνεται το σήμα) στη δική μας διεργασία, και έπειτα να επιτρέψουμε στο socket να λειτουργεί ασύγχρονα.

Οι δύο προηγούμενες τεχνικές επιδεικνύονται στα παραδείγματα των παραγράφων 4.3 και 4.4 αντίστοιχα, τα οποία αποτελούν κατάλληλα τροποποιημένες εκδόσεις του προγράμματος του παραδείγματος 2.

3.6. Αναμονή εισόδου από πολλούς descriptors

Ένα πρόβλημα που παρουσιάζεται αρκετές φορές είναι η ανάγκη να έχουμε είσοδο από πολλούς descriptors. Για παράδειγμα, μια διεργασία-εξυπηρετητής μπορεί να έχει πολλούς ταυτόχρονα ενεργούς πελάτες. Ωστόσο, δεν είναι γνωστό εκ των προτέρων ποιός πελάτης θα ζητήσει εξυπηρέτηση. Επίσης, όταν ένας πελάτης ζητήσει εξυπηρέτηση, ο εξυπηρετητής πρέπει να γνωρίζει ποιός είναι αυτός. Το πρόβλημα, λοιπόν, εντοπίζεται στο πώς μια διεργασία που μπορεί να δεχθεί είσοδο (input) από περισσότερες από μια πηγές είναι δυνατό να διαπιστώσει ποιά από τις πηγές αυτές είναι που προκάλεσε την είσοδο.

Δυο κύριες μέθοδοι υπάρχουν για τη λύση αυτού του προβλήματος. Στην πρώτη, το λεγόμενο polling, οι διαθέσιμες πηγές εισόδου έχουν τεθεί σε non blocking mode και στη συνέχεια η διεργασία-εξυπηρετητής τις ελέγχει μία προς μία, για να διαπιστώσει αν έχουν έτοιμα δεδομένα. Όποια βρεθεί να έχει, εξυπηρετείται. Αυτός ο τρόπος όμως είναι αρκετά άκομψος και σημαίνει ότι θα πρέπει να κάνουμε μία κλήση συστήματος για κάθε descriptor που θέλουμε να ελέγξουμε. Το πρόβλημα με το polling, δηλαδή, είναι πως έχουμε μία χρονική επιβάρυνση που δεν μπορούμε να αγνοήσουμε, καθώς καταναλώνει χρόνο CPU χωρίς αναγκαστικά να παράγει χρήσιμο έργο, να εξυπηρετεί δηλ. πελάτες. Απλά ελέγχει τις πηγές εισόδου (μοντέλο busy wait). Στη δεύτερη μέθοδο, που μπορεί να θεωρηθεί interrupt driven, ο εξυπηρετητής περιμένει μέχρι να εκπληρωθεί η συνθήκη "διαθέσιμη είσοδος" και στη συνέχεια ελέγχει ποιός εκπλήρωσε τη συνθήκη.

Στην περίπτωση των sockets μια πιθανή λύση του προβλήματος της αναμονής εισόδου, όταν ο εξυπηρετητής περιμένει δεδομένα σε πολλά sockets, είναι να υπάρχει μια διεργασία ανά socket. Καθεμία από αυτές τις διεργασίες εκτελεί ένα `accept()` ή ένα `read()`, ανάλογα αν το socket δεν είναι ή είναι συνδεδεμένο. Προφανώς αυτή η λύση δεν είναι polling, διότι οι κλήσεις αυτές μπλοκάρουν την καλούσα διεργασία. Ωστόσο, είναι αρκετά ακριβή σε μνήμη και CPU, καθώς απαιτείται ένας εξυπηρετητής-παιδί ανά socket.

Μια κομψότερη λύση είναι η κλήση `select()`. Η κλήση αυτή επιτρέπει στην καλούσα διεργασία να πετύχει έλεγχο των πολλαπλών πηγών εισόδου αλλά ακόμα και τυχόν πολλαπλών αποδεκτών εξόδου, δηλ. ποιά από τις διεργασίες από τις οποίες η καλούσα δέχεται δεδομένα έχει μερικά διαθέσιμα αλλά και ποιά από τις διεργασίες στις οποίες η καλούσα διεργασία στέλνει δεδομένα είναι έτοιμη να τα δεχτεί. Επίσης η κλήση αυτή δίνει τη δυνατότητα καθορισμού ενός χρονικού διαστήματος για το οποίο η `select()` θα περιμένει πριν επιστρέψει. Αν το χρονικό αυτό διάστημα είναι 0 τότε έχουμε "καθαρόαιμο" polling όπου η καλούσα διεργασία συνέχεια ελέγχει τις διάφορες πηγές εισόδου/εξόδου. Αν είναι διάφορο από το 0 τότε πριν ξαναελέγξει περιμένει για το χρονικό αυτό διάστημα, επιβάλλοντας μικρότερο φορτίο στο σύστημα. Αν τέλος δεν καθορίζεται χρονικό διάστημα τότε η `select()` περιμένει συνέχεια, μέχρι να εμφανιστεί δυνατότητα εισόδου ή εξόδου. Σε αυτή την περίπτωση γίνεται interrupt driven.

Οι πηγές εισόδου/εξόδου είναι file ή socket descriptors. Η διεργασία πριν καλέσει τη `select()` έχει ανοίξει μερικούς descriptors με `open()` (προκειμένου για αρχεία) ή με το συνδυασμό `socket()`, `bind()`, πιθανά `listen()` και ίσως `accept()`(προκειμένου για sockets), όπως αναφέρθηκε και παραπάνω. Αν έχει γίνει `accept()` τότε κάποιος πελάτης έχει ήδη συνδεθεί σε αυτό το socket (πρόκειται για το "νέο" socket που επιστρέφει η κλήση `accept()`) και ο εξυπηρετητής περιμένει είσοδο από αντίστοιχο `write()` του πελάτη. Αν δεν έχει γίνει `accept()` τότε ο εξυπηρετητής περιμένει αντίστοιχο `connect()` από κάποιον πελάτη για να κάνει στη συνέχεια `accept()`. Στην περίπτωση αυτή θεωρείται πως η διεργασία αναμένει είσοδο.

Η πλήρης σύνταξη της κλήσης `select()` δίνεται παρακάτω:

```
#include <sys/time.h>
#include <sys/types.h>

int select(int nfds, fd_set *read_ds, fd_set *write_ds,
                          fd_set *except_ds, struct timeval *timeout);
```

Η `select()` δέχεται σαν παραμέτρους δείκτες σε πεδία από bits (bit fields) τύπου `fd_set` που σημαίνουν "αν το i-οστό bit είναι 1 τότε η `select()` αναμένει είσοδο/έξοδο από τον i-οστό descriptor". Πιο συγκεκριμένα, κάθε δομή `fd_set` δουλεύει σαν μάσκα και δηλώνει ποιοί είναι οι descriptors (file ή/και

socket) που θα ελεγχθούν. Η `select()` ελέγχει κάθε μάσκα και αφαιρεί από αυτή όσους descriptor δεν βρίσκει έτοιμους. Η πρώτη παράμετρος είναι η αριθμητική τιμή του μεγαλύτερου descriptor αυξημένη κατά 1. Όσοι descriptor είναι στην `read_ds` θα ελεγχθούν αν είναι έτοιμοι για ανάγνωση δεδομένων, στην `write_ds` για γράψιμο και στην `except_ds` για ορισμένες καταστάσεις που είναι δυνατόν να συμβούν στα sockets. Εδώ μας ενδιαφέρει κυρίως η `read_ds`. Αν κάποια κατάσταση δεν μας ενδιαφέρει, μπορούμε απλώς να περάσουμε NULL στην αντίστοιχη παράμετρο.

Ο χειρισμός των δομών `fd_set` και των πεδίων τους γίνεται από κατάλληλες συναρτήσεις βιβλιοθήκης (συνήθως υλοποιημένες ως macros στο αρχείο `<sys/select.h>` και το οποίο γίνεται include στο `<sys/types.h>`). Αυτές είναι οι:

- `void FD_ZERO(fd_set &fdset);` καθαρίζει την μάσκα `fdset`.
- `void FD_CLR(int fd, fd_set &fdset);` αφαιρεί τον descriptor `fd` από την μάσκα `fdset`.
- `void FD_SET(int fd, fd_set &fdset);` προσθέτει τον descriptor `fd` στην μάσκα `fdset`.
- `int FD_ISSET(int fd, fd_set &fdset);` επιστρέφει 1 αν ο `fd` είναι στην `fdset`.
  Χρησιμοποιείται μετά την επιστροφή της `select()` για να ελεγχθεί αν ο `fd` είναι έτοιμος.

Η τελευταία παράμετρος είναι δείκτης σε μια δομή `struct timeval` και δηλώνει το μέγιστο χρονικό διάστημα αναμονής που θα περιμένει η `select()` μέχρι να επιστρέψει αν κανένας descriptor δεν είναι έτοιμος και καθορίζεται από μια μεταβλητή του τύπου:

```
#include <sys/time.h>

struct timeval {
     long tv_sec;          /* seconds (sec) αναμονής */
     long tv_usec;         /* microseconds (usec) αναμονής */
     /* Δηλ. η συνολική αναμονή είναι tv_sec sec + tv_usec usec */
};
```

την οποία αρχικοποιεί το πρόγραμμα προτού καλέσει την κλήση `select()`. Αν ο δείκτης είναι NULL τότε δεν υπάρχει χρονικό timeout και η `select()` γίνεται interrupt driven, θα μπλοκάρει δηλαδή την καλούσα διεργασία μέχρι τουλάχιστον ένας descriptor να είναι έτοιμος. Αν κανένας descriptor δεν είναι έτοιμος και λήξει το timeout τότε η `select()` επιστρέφει 0. Σε περίπτωση λάθους επιστρέφει -1 και θέτει τη μεταβλητή `errno` στην κατάλληλη τιμή. Σε κάθε άλλη περίπτωση επιστρέφει τον αριθμό των έτοιμων descriptors και τροποποιεί τα πεδία bits με τέτοιο τρόπο ώστε να σημαίνουν "αν το i-οστό bit είναι 1 τότε ο i-οστός descriptor είναι έτοιμος για είσοδο/έξοδο".

Σημειώνουμε ότι το μέγιστο πλήθος αρχείων που μπορεί να έχει ταυτόχρονα ανοικτά μια διεργασία είναι παράμετρος του συστήματος και μπορεί να βρεθεί εύκολα με μια κλήση στη συνάρτηση:

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
```

Η συνάρτηση `getrlimit()` είναι γενική και μας επιτρέφει στην δεύτερη παράμετρο που είναι δείκτης σε μια δομή `rlimit` τα μέγιστα όρια για το συγκεκριμένο αγαθό που ζητάμε στην πρώτη παράγραφο `resource`. Για παράδειγμα με την παρακάτω κλήση:

```
getrlimit(RLIMIT_NOFILE, &rlp);
```

στο πεδίο `rlp.rlim_cur` μας επιστρέφεται ο μέγιστος αριθμός αρχείων που μπορεί κάθε διεργασία να έχει ταυτόχρονα ανοικτά (το πεδίο δηλώνει πάντα το "current soft limit").

Η κλήση της `select()` ακολουθεί σε ένα σύνθετο παράδειγμα. Στο παράδειγμα αυτό υποτίθεται πως οι descriptors (file ή socket δεν έχει σημασία) του προγράμματος βρίσκονται σε ένα πίνακα `struct descr_status descr[]`. Ο πίνακας αυτός αρχικοποιείται σε προηγούμενο στάδιο του προγράμματος με κατάλληλες κλήσεις `open()` και `socket()`:

```
#define  DESCR_READ      1
#define  DESCR_WRITE     2
#define  DESCR_ACCEPT    3
#define  DESCRSIZ        4

struct  descr_status {
   int  descriptor, status;
} descr[DESCRSIZ];

descr[0].descriptor = open(…);   descr[0].status = DESCR_READ;
descr[1].descriptor = socket(…); descr[1].status = DESCR_READ;
descr[2].descriptor = socket(…); descr[2].status = DESCR_ACCEPT;
descr[3].descriptor = open(…);   descr[3].status = DESCR_WRITE;
```

Ο πίνακας `descr[]` καταγράφει δηλ. την τιμή και την κατάσταση των descriptors που ενδιαφέρουν. Ο descriptor `descr[2].descriptor` περιμένει κάποιο `connect()` από πελάτη, οι `descr[0].descriptor` και `descr[1].descriptor` κάποιο `write()` από πελάτη (ο 1 είναι socket descriptor, χωρίς αυτό να έχει σημασία για τη `select()`) και ο `descr[3].descriptor` `read()` από πελάτη. Δηλαδή, τα στοιχεία 0, 1 και 2 του πίνακα αναμένουν είσοδο, ενώ το 3 έξοδο:

```
#include  <sys/time.h>
          /* Επιπλέον include file εκτός αυτών που χρειάζονται για τις άλλες κλήσεις sockets */

struct timeval tval;
fd_set input_fds, output_fds;

tval.sec = 1; tval.usec = 1000;
     /* Η select() θα περιμένει 1.001 sec (1 sec και 1000 usec = 1 msec).
        Αν tval.sec = tval.usec = 0 τότε έχουμε "καθαρόαιμο" polling με busy wait */

     /* Μηδενίζονται και αρχικοποιούνται κατάλληλα τα πεδία bits των descriptors
        που θα χρησιμοποιηθούν για είσοδο και έξοδο */

FD_ZERO(&input_fds); FD_ZERO(&output_fds);
FD_SET(descr[0].descriptor, &input_fds);
FD_SET(descr[1].descriptor, &input_fds);
FD_SET(descr[2].descriptor, &input_fds);
FD_SET(descr[3].descriptor, &input_fds);

while ( (r = select(DESCRSIZ, &input_fds, &output_fds,
                                          (fd_set *) 0, &tval)) >= 0 )
{
     /* Η select() επιστρέφει 0 αν γίνει timeout, -1 αν συμβεί λάθος και μεγαλύτερο από 0 αν υπάρξει
     δυνατότητα εισόδου/εξόδου. Ο αριθμός που επιστρέφεται είναι ο αριθμός των descriptors για τους
     οποίους υπάρχει αυτή η δυνατότητα. Το πρώτο όρισμα της select() είναι το μέγιστο μέγεθος των
     πινάκων input_fds, output_fds. Αν &tval είναι (struct timeval *) 0 (NULL)
     τότε δεν υπάρχει timeout και η κλήση γίνεται interrupt driven. Το 4ο όρισμα είναι οι descriptors που
     σχετίζονται με ειδικές συνθήκες (exceptional conditions) και είναι σχεδόν πάντα (fd_set *) 0
     (NULL ) */

     if ( r > 0 )
     {
          for (i=0; i<DESCRSIZ; i++)
              if ( FD_ISSET(descr[i].descriptor, &input_fds) ||
              FD_ISSET(descr[i].descriptor, &output_fds))
                  switch (descr[i].status)
                  {
                    case DESCR_ACCEPT: accept(descr[i].descriptor, …); …
                    case DESCR_READ:   read(descr[i].descriptor, …); …
                    case DESCR_WRITE:  write(descr[i].descriptor, …); …
                  }
```

```
            /* Ξαναρχικοποιούνται τα πεδία bits για την επόμενη κλήση */
            FD_ZERO(&input_fds); FD_ZERO(&output_fds);
            FD_SET(descr[0].descriptor, &input_fds);
            FD_SET(descr[1].descriptor, &input_fds);
            FD_SET(descr[2].descriptor, &input_fds);
            FD_SET(descr[3].descriptor, &output_fds);
        }
}
```

Η αλήθεια είναι πως η κλήση `select()`, παρότι λύνει με έξυπνο τρόπο ένα δύσκολο πρόβλημα, έχει μια σχετική πολυπλοκότητα. Το παραπάνω παράδειγμα μπορεί να απλοποιηθεί αρκετά, ιδίως όταν είναι σίγουρο πως μόνο μια ενέργεια (π.χ. `read()`) θα χρειαστεί.

3.7. Κλείσιμο socket

Το κλείσιμο ενός καναλιού επικοινωνίας είναι απλή υπόθεση: αρκεί να κληθεί η `close()` πάνω στα δυο sockets-άκρα του καναλιού.

```
#include <unistd.h>

int close(int sd);
```

Μόλις κλείσει ένα socket τότε καταστρέφεται και το κανάλι επικοινωνίας με το οποίο αυτό σχετιζόταν (άμεσα ή μετά από λίγο). Αν μια διεργασία προσπαθήσει να διαβάσει ή να γράψει δεδομένα από ένα socket το οποίο έχει κλείσει, τότε θα λάβει το σήμα `SIGPIPE` με αποτέλεσμα να διακοπεί η εκτέλεσή της, αν δεν έχει φροντίσει για τον χειρισμό ή την απενεργοποίηση του σήματος μέσα από την συνάρτηση `signal()`. Αν μέχρι να κλείσει το κανάλι επικοινωνίας στέλνονται δεδομένα, τότε αυτά θα συσσωρεύονται στο buffer του συστήματος ώσπου να απορριφθούν μετά το κλείσιμο της σύνδεσης. Για να αποφύγουμε κάτι τέτοιο μπορούμε να χρησιμοποιήσουμε την κλήση:

```
int shutdown(int sd, int how);
```

πριν από την `close()`, όπου ο `sd` δηλώνει τον socket descriptor και το δεύτερο όρισμα `how` μπορεί να πάρει τις τιμές 0 αν δεν ενδιαφερόμαστε να διαβάσουμε δεδομένα από τον `sd`, 1 αν δεν ενδιαφερόμαστε να στείλουμε δεδομένα και 2 για τον συνδυασμό των δύο.

4. ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΕΞΥΠΗΡΕΤΗΤΗ (SERVER)

4.1. Παράδειγμα 1: Ένας απλός Εξυπηρετητής

Στο σημείο αυτό θα επιχειρήσουμε να υλοποιήσουμε έναν απλό server ο οποίος θα αναφέρει την ώρα της μηχανής στην οποία θα τρέχει. Η δομή του προγράμματος ακολουθεί το σχήμα που δώσαμε στην προηγούμενη ενότητα, δηλ. δημιουργούμε το socket, το ονομάζουμε και περιμένουμε συνδέσεις από τους πελάτες. Όλα τα παραδείγματα που παρουσιάζονται είναι διαθέσιμα και σε ηλεκτρονική μορφή στο `ftp://ftp.dsclab.ece.ntua.gr/pub/sockets`

```
/*
 * Simple time server - Version 1: Serial execution
 * Alkis
 */

#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>

extern int daemonize(void);
static void serve(int);

#define PORT 6060

main( int argc, char **argv )
{
      int lsd;                /* Listening socket */
      struct sockaddr_in sin;       /* Binding struct */
      int sin_size=sizeof(sin);
      int sd;                       /* Socket to accept new connexion */

      /* Create listening socket */

      if ( (lsd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
            fprintf(stderr,"%s: cannot create listening socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Bind socket to port */

      sin.sin_family        = AF_INET;
      sin.sin_port          = htons(PORT);
      sin.sin_addr.s_addr = htonl(INADDR_ANY);

      if ( bind(lsd, &sin, sin_size) < 0 ) {
            fprintf(stderr, "%s: cannot bind listening socket: ", argv[0]);
            perror(0);
            exit(1);
      }
```

```
      /* Initiate a listen queue */

      if ( listen(lsd, 5) < 0 ) {
            fprintf(stderr, "%s: cannot listen on socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Become a daemon */
#ifndef DEBUG
      if ( daemonize() < 0 ) {
            fprintf(stderr, "%s: cannot disconnect \
                              from controlling terminal: ", argv[0]);
            perror(0);
            exit(1);
      }
#endif
      /* Take care of the SIGPIPE signal - ignore it */
      signal(SIGPIPE, SIG_IGN);

      /* Ready to accept connexions */
      while ( 1 ) {
            if ( (sd=accept(lsd, &sin, &sin_size)) < 0 )
                  exit(errno);
            serve(sd);
            shutdown(sd, 2);
            close(sd);
      }
}


/* Tell the time to socket descriptor sd */

void serve(int sd)
{
      time_t local_time;
      char *time_string;

      time(&local_time);
      time_string = ctime(&local_time);

      write(sd, time_string, strlen(time_string));

      return;
}
```

4.2. Παράδειγμα 2: Εξυπηρετητής χωρίς αναμονή στην εξυπηρέτηση

Το προηγούμενο πρόγραμμα είχε ένα αδύνατο σημείο: όσο εξυπηρετούσε ένα πελάτη δεν είχε την δυνατότητα να δεχθεί καινούργιες συνδέσεις, παρόλο που κάθε εξυπηρέτηση είναι ανεξάρτητη από τις προηγούμενες. Το πρόβλημα αυτό λύνεται με την δημιουργία μιας καινούργιας διεργασίας, η οποία θα αναλαμβάνει να εξυπηρετήσει τον πελάτη, ενώ ο server θα περιμένει για καινούργιες συνδέσεις.

```
/*
 * Simple time server - Version 2: Forks at every connexion
 * Alkis
 */
```

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

extern int daemonize(void);
static void serve(int);

#define PORT 6060

#ifdef DEBUG
#define MSG(x) puts(x);
#else
#define MSG(x) ;
#endif

main( int argc, char **argv )
{
      int lsd;                /* Listening socket */
      struct sockaddr_in sin;       /* Binding struct */
      int sin_size=sizeof(sin);
      int sd;                       /* Socket to accept new connexion */

      /* Create listening socket */

      if ( (lsd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
            fprintf(stderr,"%s: cannot create listening socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Bind socket to port */

      sin.sin_family        = AF_INET;
      sin.sin_port          = htons(PORT);
      sin.sin_addr.s_addr = htonl(INADDR_ANY);

      if ( bind(lsd, &sin, sin_size) < 0 ) {
            fprintf(stderr, "%s: cannot bind listening socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Initiate a listen queue */

      if ( listen(lsd, 5) < 0 ) {
            fprintf(stderr, "%s: cannot listen on socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Become a daemon */
#ifndef DEBUG
      if ( daemonize() < 0 ) {
            fprintf(stderr, "%s: cannot disconnect \
                              from controlling terminal: ", argv[0]);
            perror(0);
```

```
            exit(1);
      }
#endif
      /* Take care of SIGPIPE signal - ignore it */
      signal(SIGPIPE, SIG_IGN);

      /* Take care of SIGCHLD - ignore it */
      signal(SIGCHLD, SIG_IGN);

      /* Ready to accept connexions */
      while ( 1 ) {

MSG("Before connexion")

            if ( (sd=accept(lsd, &sin, &sin_size)) < 0 )
                  exit(errno);

MSG("connexion")
            switch ( fork() )
            {
                  case  0:    /* Child */
                        serve(sd);
                        shutdown(sd, 2);
                        close(sd);
                        exit(0);
                  case -1:    /* Error */
                  default:    /* Parent */
                        close(sd);
                        break;
            }
      }
}


/* Tell the time to socket descriptor sd */

void serve(int sd)
{
      time_t local_time;
      char *time_string;

      time(&local_time);
      time_string = ctime(&local_time);

      write(sd, time_string, strlen(time_string));

      return;
}
```

Η μόνη διαφορά με το προηγούμενο πρόγραμμα βρίσκεται στην εξυπηρέτηση της σύνδεσης. Αυτό που πρέπει να προσέχουμε σε αυτές τις περιπτώσεις είναι ο χειρισμός του σήματος SIGCHLD που θα σταλεί στην διεργασία πατέρα μόλις τελειώσει η διεργασία παιδί. Η εξ' ορισμού συμπεριφορά του σήματος είναι να τερματιστεί η διεργασία. Γι' αυτό το λόγο, φροντίζουμε να το αγνοήσουμε μέσω της συνάρτησης `signal()`. Εναλλακτικά, μπορούμε να θέσουμε έναν signal handler για το συγκεκριμένο σήμα (πάλι μέσω της `signal()`), ο οποίος όταν εκτελείται θα καλεί την `wait()` για τη διεργασία παιδί.

4.3. Παράδειγμα 3: Εξυπηρετητής χωρίς αναμονή στην είσοδο/έξοδο

```
/*
 * Simple time server - Version 3: Non blocking IO
 * Alkis
 */

#include <sys/types.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <fcntl.h>

extern int daemonize(void);
static void serve(int);

#define PORT 6060

#ifdef DEBUG
#define MSG(x) puts(x);
#else
#define MSG(x) ;
#endif

main( int argc, char **argv )
{
      int lsd;                /* Listening socket */
      struct sockaddr_in sin;       /* Binding struct */
      int sin_size=sizeof(sin);
      int sd;                       /* Socket to accept new connexion */

      /* Create listening socket */

      if ( (lsd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
            fprintf(stderr,"%s: cannot create listening socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Set Non blocking mode */

      if ( fcntl(sd, F_SETFL, O_NDELAY) < 0 ) {
            fprintf(stderr, "%s: cannot set non blocking mode: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Bind socket to port */

      sin.sin_family        = AF_INET;
      sin.sin_port          = htons(PORT);
      sin.sin_addr.s_addr = htonl(INADDR_ANY);

      if ( bind(lsd, &sin, sin_size) < 0 ) {
            fprintf(stderr, "%s: cannot bind listening socket: ", argv[0]);
```

```
            perror(0);
            exit(1);
      }

      /* Initiate a listen queue */

      if ( listen(lsd, 5) < 0 ) {
            fprintf(stderr, "%s: cannot listen on socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Become a daemon */
#ifndef DEBUG
      if ( daemonize() < 0 ) {
            fprintf(stderr, "%s: cannot disconnect \
                              from controlling terminal: ", argv[0]);
            perror(0);
            exit(1);
      }
#endif
      /* Take care of SIGPIPE signal - ignore it */
      signal(SIGPIPE, SIG_IGN);

      /* Take care of SIGCHLD - ignore it */
      signal(SIGCHLD, SIG_IGN);

      /* Ready to accept connexions */
      while ( 1 ) {

MSG("Before connexion")

            if ( (sd=accept(lsd, &sin, &sin_size)) < 0 ) {
                  if ( errno == EWOULDBLOCK ) {
                        /* Perform some other task */
                        continue;
                  }
                  else exit(errno);
            }

MSG("connexion")
            switch ( fork() )
            {
                  case  0:    /* Child */
                        serve(sd);
                        shutdown(sd, 2);
                        close(sd);
                        exit(0);
                  case -1:    /* Error */
                  default:    /* Parent */
                        close(sd);
                        break;
            }
      }
}


/* Tell the time to socket descriptor sd */

void serve(int sd)
{
      time_t local_time;
      char *time_string;
```

```
	time(&local_time);
	time_string = ctime(&local_time);

	write(sd, time_string, strlen(time_string));

	return;
}
```

4.4. Παράδειγμα 4: Εξυπηρετητής χωρίς αναμονή Ε/Ε με χρήση σήματος SIGIO

```
/*
 * Simple time server - Version 4: Non blocking IO with SIGIO
 * Alkis
 */

#include <sys/types.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <fcntl.h>

extern int daemonize(void);
static void serve(int);
static void handle_connexions(void);

#define PORT 6060

#ifdef DEBUG
#define MSG(x) puts(x);
#else
#define MSG(x) ;
#endif

int lsd;                /* Listening socket */
struct sockaddr_in sin;       /* Binding struct */
int sin_size=sizeof(sin);

main( int argc, char **argv )
{
	int my_pid=getpid();

	/* Create listening socket */

	if ( (lsd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
		fprintf(stderr,"%s: cannot create listening socket: ", argv[0]);
		perror(0);
		exit(1);
	}

	/* Receive SIGIO when socket is ready */
```

```
      if ( fcntl(lsd, F_SETOWN, getpid()) < 0 ) {
            fprintf(stderr, "%s: cannot receive SIGIO: ", argv[0]);
            perror(0);
            exit(1);
      }

      if ( fcntl(lsd, F_SETFL, FASYNC) < 0 ) {
            fprintf(stderr, "%s: cannot receive SIGIO: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Bind socket to port */

      sin.sin_family        = AF_INET;
      sin.sin_port          = htons(PORT);
      sin.sin_addr.s_addr = htonl(INADDR_ANY);

      if ( bind(lsd, &sin, sin_size) < 0 ) {
            fprintf(stderr, "%s: cannot bind listening socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Initiate a listen queue */

      if ( listen(lsd, 5) < 0 ) {
            fprintf(stderr, "%s: cannot listen on socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Become a daemon */
#ifndef DEBUG
      if ( daemonize() < 0 ) {
            fprintf(stderr, "%s: cannot disconnect \
                              from controlling terminal: ", argv[0]);
            perror(0);
            exit(1);
      }
#endif
      /* Take care of SIGPIPE signal - ignore it */
      signal(SIGPIPE,SIG_IGN);

      /* Take care of SIGIO */
      signal(SIGIO,handle_connexions);

      /* Take care of SIGCHLD - ignore it */
      signal(SIGCHLD,SIG_IGN);

      /* Ready to accept connexions */
      while ( 1 ) {
            /* Other staff here */
      }
}


/* Tell the time to socket descriptor sd */

void serve(int sd)
{
      time_t local_time;
      char *time_string;
```

```
      time(&local_time);
      time_string = ctime(&local_time);

      write(sd, time_string, strlen(time_string));

      return;
}

void handle_connexions(void)
{
      int sd;                        /* Socket to accept new connexion */

      if ( (sd=accept(lsd, &sin, &sin_size)) < 0 )
            exit(errno);

MSG("connexion")
      switch ( fork() )
      {
            case  0:    /* Child */
                  serve(sd);
                  shutdown(sd, 2);
                  close(sd);
                  exit(0);
            case -1:    /* Error */
            default:    /* Parent */
                  close(sd);
                  break;
      }
}
```

4.5. Παράδειγμα 5: Εξυπηρετητής με δυνατότητα ελέγχου της λειτουργίας του

Στο παράδειγμα αυτό θα τροποποιήσουμε τον server ώστε να δέχεται εντολές από τον χρήστη. Επειδή οι εξυπηρετητές τρέχουν συνήθως σαν δαίμονες (διεργασίες που εκτελούνται στο παρασκήνιο), δεν είναι δυνατόν να χρησιμοποιηθεί η standard είσοδος. Μία επιλογή είναι να ανοίξουμε άλλο ένα socket ειδικά για αυτό το σκοπό. Εδώ θα χρησιμοποιήσουμε ένα FIFO αρχείο (named pipe).

```
/*
 * Simple time server - Version 5: Accept user input
 * Alkis
 */

#include <sys/types.h>
#include <sys/time.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <fcntl.h>

/* States */
#define RUN    0
#define PAUSE  1
```

```
extern int daemonize(void);
static void serve(int);

#define PORT      6060
#define BUFSIZE   1024

#define COMMAND_PIPE "/tmp/srvpipe"

#ifdef DEBUG
#define MSG(x) puts(x);
#else
#define MSG(x) ;
#endif

main( int argc, char **argv )
{
      int state;
      int lsd;                /* Listening socket */
      int command;                  /* Command file descriptor */
      struct sockaddr_in sin;       /* Binding struct */
      int sin_size=sizeof(sin);
      int sd;                       /* Socket to accept new connexion */
      fd_set mask;
      char buffer[BUFSIZE];

      /* Open commands pipe */

      if ( (command=open(COMMAND_PIPE, O_RDONLY|O_NDELAY)) < 0 ) {
            fprintf(stderr, "%s: cannot open pipe: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Create listening socket */

      if ( (lsd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
            fprintf(stderr,"%s: cannot create listening socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Bind socket to port */

      sin.sin_family        = AF_INET;
      sin.sin_port          = htons(PORT);
      sin.sin_addr.s_addr = htonl(INADDR_ANY);

      if ( bind(lsd, &sin, sin_size) < 0 ) {
            fprintf(stderr, "%s: cannot bind listening socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Initiate a listen queue */

      if ( listen(lsd, 5) < 0 ) {
            fprintf(stderr, "%s: cannot listen on socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Become a daemon */
#ifndef DEBUG
```

```
      if ( daemonize() < 0 ) {
            fprintf(stderr, "%s: cannot disconnect \
                              from controlling terminal: ", argv[0]);
            perror(0);
            exit(1);
      }
#endif
      /* Take care of SIGPIPE signal - ignore it */
      signal(SIGPIPE, SIG_IGN);

      /* Take care of SIGCHLD - ignore it */
      signal(SIGCHLD, SIG_IGN);

      /* Ready to accept connexions */
      state=RUN;

      while ( 1 ) {

MSG("Before connexion")

            FD_ZERO(&mask);
            if ( state == RUN )
                  FD_SET(lsd, &mask);
            FD_SET(command, &mask);

            select(64, &mask, 0, 0, 0);

            if ( FD_ISSET(lsd, &mask) ) {
                  if ( (sd=accept(lsd, &sin, &sin_size)) < 0 )
                        exit(errno);

MSG("connexion")
                  switch ( fork() )
                  {
                        case  0:    /* Child */
                              serve(sd);
                              shutdown(sd, 2);
                              close(sd);
                              exit(0);
                        case -1:    /* Error */
                        default:    /* Parent */
                              close(sd);
                              break;
                  }
            }

            if ( FD_ISSET(command, &mask) ) {

                  if ( read(command, &buffer, BUFSIZE) < 0 )
                        exit(errno);

                  if ( !strcmp(buffer, "shutdown\n") ) {
                        shutdown(lsd, 2);
                        close(lsd);
                        close(command);
                        exit(0);
                  }
                  else if ( !strcmp(buffer, "pause\n") )
                        state=PAUSE;
                  else if ( !strcmp(buffer, "continue\n") )
                        state=RUN;
            }
      }
```

```
}


/* Tell the time to socket descriptor sd */

void serve(int sd)
{
      time_t local_time;
      char *time_string;

      time(&local_time);
      time_string = ctime(&local_time);

      write(sd, time_string, strlen(time_string));

      return;
}
```

Ο έλεγχος του server γίνεται από μηνύματα που γράφουμε στο αρχείο `/tmp/srvpipe` (το οποίο θα πρέπει να δημιουργήσουμε μόνοι μας με την εντολή `mknod`). Για να εμποδίσουμε νέες συνδέσεις αρκεί να χρησιμοποιήσουμε την παρακάτω εντολή στον φλοιό:

```
echo pause > /tmp/srvpipe
```

και για να επιτρέψουμε ξανά τις συνδέσεις:

```
echo continue > /tmp/srvpipe
```

Τέλος, με την εντολή `shutdown` σταματάμε το πρόγραμμα. Θα πρέπει να προσέξουμε ότι τα pipes όπως και τα stream sockets δεν διατηρούν τα όρια των μηνυμάτων. Αν δηλ. γράψουμε:

```
/usr/5bin/echo 'pause\\ncontinue' > /tmp/srvpipe
```

(χρησιμοποιούμε την Sys V έκδοση της `echo` αφού δέχεται τα escape codes της C) θα διαβάσουμε όλη τη συμβολοσειρά και όχι μόνο το string μέχρι το πρώτο newline.

5. ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΠΕΛΑΤΗ (CLIENT)

Σε αυτό το σημείο θα εξετάσουμε την δομή ενός πελάτη που συνδέεται σε συγκεκριμένο εξυπηρετητή. Το διάγραμμα ροής δεν διαφέρει γενικά από αυτό που είχαμε δεί στην περίπτωση του εξυπηρετητή, εκτός βέβαια από την αίτηση σύνδεσης και το ότι ο πελάτης δεν είναι υποχρεωμένος να καλέσει την κλήση `bind()` στον socket descriptor που έχει δημιουργήσει. Το λειτουργικό σύστημα θα κάνει μόνο του αυτή την ενέργεια όταν ο πελάτης προσπαθήσει να συνδεθεί στον εξυπηρετητή. Το μόνο λοιπόν που πρέπει να ορίσουμε για το πρόγραμμα πελάτη είναι η κλήση `connect()` για σύνδεση με κάποιο συγκεκριμένο εξυπηρετητή.

5.1. Παράδειγμα 6: Πελάτης για τον Εξυπηρετητή Ώρας

Ως παράδειγμα θα υλοποιήσουμε τον πελάτη ο οποίος θα συνδέεται στον εξυπηρετητή που δείξαμε στα προηγούμενα παραδείγματα.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define BUFSIZE         1024
#define SERVER_PORT     6060

main( int argc, char **argv )
{
      int sd;                       /* Socket descriptor */
      struct sockaddr_in server;    /* Server to connect */
      struct hostent *server_host;  /* Host info */
      char buf[BUFSIZE];
      int nbytes;

      /* Create socket */
      if ( (sd=socket(AF_INET, SOCK_STREAM, 0)) < 0 ) {
            fprintf(stderr, "%s: cannot create socket: ", argv[0]);
            perror(0);
            exit(1);
      }

      /* Get info on host */
      if ( (server_host=gethostbyname(argv[1])) == NULL ) {
            fprintf(stderr, "%s: unknown host %s\n", argv[0], argv[1]);
            exit(1);
      }

      /* Set up struct sockaddr_in */
      server.sin_family = AF_INET;
      server.sin_port   = SERVER_PORT;
      bcopy((char*)server_host->h_addr, (char*)&server.sin_addr,
                  server_host->h_length);

      /* Connect */
      if ( connect(sd, &server, sizeof(server)) < 0 ) {
            fprintf(stderr, "%s: cannot connect to server: ", argv[0]);
            perror(0);
            exit(1);
      }
```

```
	/* Get date */

	if ( (nbytes=read(sd, buf, BUFSIZE-1)) <= 0 ) {
		fprintf(stderr, "%s: read failed: ", argv[0]);
		perror(0);
		exit(1);
	}

	buf[nbytes] = 0;
	printf("Date on host %s is: %s\n", argv[1], buf);

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

6. ΧΡΗΣΗ DATAGRAM SOCKETS

Τα sockets που παρουσιάστηκαν μέχρι στιγμής είναι "stream sockets" τύπου SOCK_STREAM. Εγκαθιστούν δηλ. ένα "μόνιμο" κανάλι αξιόπιστης (με χρήση του πρωτοκόλλου TCP) επικοινωνίας ανάμεσα σε δύο διεργασίες. Χρησιμοποιούνται σε υπηρεσίες "connection oriented", όπου δηλ. πριν την αποστολή οποιουδήποτε μηνύματος πρέπει να εγκατασταθεί με χρήση του συνδυασμού `connect()` / `accept()` το κανάλι. Υπάρχουν όμως και "datagram sockets" τύπου SOCK_DGRAM για τα οποία δεν χρειάζεται πρότερη εγκατάσταση καναλιού επικοινωνίας. Αυτή η υπηρεσία είναι "connectionless". Στην περίπτωση αυτή οι πληροφορίες στέλνονται σαν αυτόνομα πακέτα πάνω από το δίκτυο (με χρήση του πρωτοκόλλου UDP). Επιπλέον δεν είναι εγγυημένη η αξιόπιστη μεταφορά τους και η διατήρηση της σωστής σειράς σε περίπτωση που κάποιο σύνολο πληροφοριών είναι πολύ μεγάλο για να χωρέσει σε ένα μόνο πακέτο, οπότε κατατέμνεται (fragmentation). Για αυτές τις λειτουργίες πρέπει να φροντίσει η εφαρμογή. Πλεονέκτημα της μετάδοσης αυτής είναι, εκτός από την απουσία ανάγκης εγκατάστασης καναλιού επικοινωνίας, η μεγάλη ταχύτητα που προσφέρει. Αν η επικοινωνία stream sockets μπορεί να παρομοιασθεί με τηλεφωνική επικοινωνία όπου πριν την συνομιλία προηγείται η εγκατάσταση του καναλιού (κλήση αριθμού κτλ.), η επικοινωνία datagram sockets μοιάζει περισσότερο με ταχυδρομική επικοινωνία, όπου κάθε γράμμα-πακέτο πληροφορίας είναι αυτόνομο και ανεξάρτητο σε σχέση με όλα τα άλλα. Αν απαιτείται συνεχής αξιόπιστη ροή δεδομένων με τη σωστή σειρά και ιδίως σε περιβάλλον client - server, είναι προτιμότερη η επικοινωνία stream. Η επικοινωνία datagram υπερέχει όταν ζητείται ταχεία κίνηση αυτόνομων μικρών πακέτων πληροφορίας, κυρίως σε περιβάλλον peer-to-peer (αλλά και σε client - server).

Για να χρησιμοποιηθούν datagram sockets πρέπει να κληθεί η `socket()` με παράμετρο SOCK_DGRAM αντί για SOCK_STREAM. Στη συνέχεια πρέπει να κληθεί η `bind()`, είτε πρόκειται για πελάτη είτε για εξυπηρετητή, ώστε να δοθεί διεύθυνση στο socket. Η αποστολή πληροφορίας γίνεται με την κλήση `sendto()` και η λήψη της με τη `recvfrom()`.

Ο αποστολέας χρησιμοποιεί τη `sendto()` ως εξής:

```
int  sd;       /* Το datagram socket του αποστολέα μετά από socket() και bind() */
char buf[BUFSIZ];          /* Ο buffer όπου βρίσκονται τα δεδομένα προς αποστολή */
int  len;                  /* Το μήκος του buffer */
struct sockaddr_in addr;   /* Η διεύθυνση του παραλήπτη */
struct hostent  *hostp;    /* Πληροφορίες για το σύστημα προορισμού */

addr.sin_family = AF_INET;
addr.sin_port   = htons(PORTNUM);
bcopy(hostp->h_addr, &addr.sin_addr, hostp->h_length);
len = sizeof(buf);
n = sendto(sd, buf, len, 0, &addr, sizeof(addr));
```

Αντίστοιχα για τον παραλήπτη είναι η `recvfrom()`:

```
int  sd;          /* Το datagram socket του παραλήπτη μετά από socket() και bind() */
char buf[BUFSIZ];   / * Ο buffer όπου θα τοποθετηθούν τα δεδομένα */
int  len;         /* Το μήκος του buffer */
struct sockaddr_in addr;
                  /* Το λειτουργικό σύστημα θα τοποθετήσει εδώ τη διεύθυνση του αποστολέα */
int  addrlen;     /* Το μήκος της διεύθυνσης του αποστολέα. Αλλάζει από το Λ.Σ. */

len     = sizeof(buf);
addrlen = sizeof(addr);
n = recvfrom(sd, buf, len, 0, &addr, &addrlen);
```

Η κλήση `sendto()`, όπως και η `recvfrom()`, επιστρέφει τον αριθμό των bytes που γράφτηκαν ή διαβάστηκαν και -1 αν γίνει λάθος.

ΙΙ. ΒΙΒΛΙΟΓΡΑΦΙΑ

[ΠΑΠΑ91] Λειτουργικά Συστήματα, Μέρος Ι: Αρχές Λειτουργίας, Γ. Κ. Παπακωνσταντίνου, Ν. Α. Μπιλάλης, Π. Δ. Τσανάκας, Εκδόσεις Συμμετρία, Αθήνα 1991.

[BROW94] UNIX Distributed Programming, Chris Brown, Prentice Hall, 1994.

[COFF90] UNIX System V Release 4: The Complete Reference, Stephen Coffin, Osborne McGraw-Hill, 1990.

[LEFF89] The Design and Implementation of the 4.3BSD UNIX Operating System, S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, Reading, MA, Addison-Wesley, 1989.

[STEV90] Unix Network Programming, W. R. Stevens, Prentice Hall, 1990.

[STEV92] Advanced Programming in the UNIX Environment, W. R. Stevens, Addison-Wesley, 1992.

[TANE91] Δίκτυα Υπολογιστών, Andrew S. Tanenbaum, Δεύτερη Έκδοση, Prentice Hall, Ελληνική Έκδοση Παπασωτηρίου 1991.

[TANE92] Modern Operating Systems, Andrew S. Tanenbaum, Prentice Hall, 1992.

[COUL01] Distributed Operating Systems: Concepts and Design, G. Coulouris, J. Dollimore, T.Kindberg, 3rd Edition, Addison-Wesley 2001

[GAY01] Linux Socket Programming by Example, W. Gay, QUE, 2001