ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΣΠΟΥΔΩΝ ΣΤΑ ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

Μάθημα:

Δίκτυα Υπολογιστών

Θέμα Εργασίας:

Υλοποίηση Κατανεμημένης Εφαρμογής με Χρήση των Υπηρεσιών των Πρωτοκόλλων TCP και UDP

Υπεύθυνος Καθηγητής:

Θ. Αποστολόπουλος

Ομάδα Εργασίας:

Βασιλείου Θεώνη

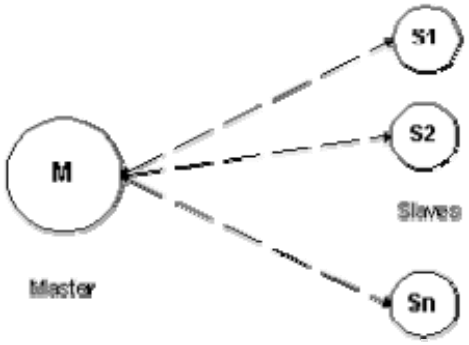
Γάκης Κωνσταντίνος

Οικονομίδης Δημήτριος

ΑΚΑΔ. ΕΤΟΣ 2001 – 2002

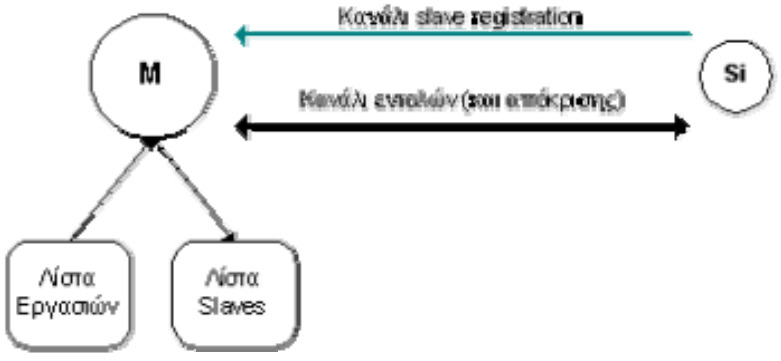
ΠΕΡΙΓΡΑΦΗ ΕΡΓΑΣΙΑΣ

Η παρούσα εφαρμογή πελάτη / εξυπηρετητή υλοποιήθηκε στα πλαίσια του μαθήματος «Δίκτυα Υπολογιστών» του Μ.Π.Σ. «Πληροφοριακά Συστήματα» του Ο.Π.Α. Πρόκειται για μια κατανεμημένη εφαρμογή που χρησιμοποιεί τις υπηρεσίες των πρωτοκόλλων TCP και UDP και κατά την οποία μια διεργασία master αναθέτει εργασίες (tasks) σε διεργασίες slaves. Η υλοποίηση της άσκησης έγινε σε περιβάλλον λειτουργικού συστήματος UNIX με την γλώσσα προγραμματισμού C και με την χρήση των λειτουργιών των BSD sockets.



Η λειτουργία της εφαρμογής έχει ως εξής:

1. Ο master ξεκινά και αναμένει εθελοντές slaves έχοντας μια (σταθερή) λίστα εργασιών προς διεκπεραίωση.
2. Ένας εθελοντής slave ξεκινά και δηλώνει τη παρουσία του στον master στέλνοντας κατάλληλο μήνυμα.
3. Ο master καταγράφει τον παραπάνω slave στη λίστα με τους διαθέσιμους (για την εκτέλεση εργασιών) slaves και ανοίγει ένα κανάλι επικοινωνίας μέσω του οποίου ελέγχει κατάλληλα τον slave (με χρήση συγκεκριμένων εντολών / αποκρίσεων).
4. Ο slave «υπακούει» στις εντολές του master (εκτελεί ή αναστέλλει εργασίες που του αναθέτει ο master, δίνει αναφορά της κατάστασής του, τερματίζει)
5. Ο master ενημερώνει κατάλληλα τις λίστες με τους διαθέσιμους slaves και τις προς εκτέλεση εργασίες (όταν εκτελείται επιτυχώς κάποια εργασία).
6. Ο master, με την επιτυχή εξάντληση της λίστας των προς διεκπεραίωση εργασιών, ειδοποιεί τους συνεργαζόμενους slaves να τερματίσουν τη λειτουργία τους.



Οι εργασίες που αναθέτει ο master στους slaves αφορούν λήψη εγγράφου web. Ο slave καλεί την εντολή wget του λειτουργικού συστήματος UNIX με παράμετρο τη διεύθυνση url που δέχεται από τον master.

Η επικοινωνία στο κανάλι slave registration γίνεται με χρήση του πρωτοκόλλου UDP. Ο master ακούει σε συγκεκριμένο port.

Το μήνυμα «παρουσίας» που στέλνει ο slave είναι ένα string (τελειώνει με χαρακτήρα αλλαγής γραμμής) της μορφής:

`HERE_I_AM` *n*,

όπου n το port number στο οποίο ακούει ο slave (αυτό που χρησιμοποιείται για το κανάλι εντολών).

Το κανάλι εντολών είναι μια σύνδεση TCP την οποία ανοίγει ο master (κάνει active open). Κάθε εντολή που στέλνει ο master προς το slave είναι string (τελειώνει με χαρακτήρα αλλαγής γραμμής) της μορφής:

`εντολή [όρισμα]`

Το σύνολο των εντολών δίνεται στον παρακάτω πίνακα:

Εντολή master	*Περιγραφή εντολής (για τον slave)*	*Απόκριση slave*
`GET url`	Ανάθεση εργασίας λήψης εγγράφου από τη διεύθυνση url	`OK` ή `LATER`
`STOP`	Αναστολή εργασίας	`OK`
`EXIT`	Αναστολή εργασίας και τερματισμός λειτουργίας	`OK`
`STATUS`	Αναφορά κατάστασης	`BUSY` ή `IDLE`

Ο slave, μετά την ολοκλήρωση κάθε εργασίας που έχει αναλάβει στέλνει στον master, μέσω του καναλιού registration, ένα μήνυμα (string) της μορφής:

`THANK_YOU` *n* *(στην περίπτωση επιτυχίας)*

ή

`SORRY` *n* *(στην περίπτωση αποτυχίας),*

όπου n το port number στο οποίο ακούει ο slave (παρόμοια με το μήνυμα ”HERE_I_AM”).

Η λίστα με τους εθελοντές όπως και η λίστα με τα αρχεία (τα οποία διαβάζονται στην αρχή εκτέλεσης του προγράμματος) υλοποιούνται ως δυναμικές λίστες με την χρήση δεικτών(pointers) και δομών(structs) της γλώσσας C. Η συγκεκριμένη περιγραφή της δομής τους περιλαμβάνεται στο αρχείο επικεφαλίδων inet.h.

Ο master, κατά τη λειτουργία του εμφανίζει πληροφορίες σχετικά με τους slaves που ελέγχει, όπως IP διεύθυνση και port number του slave, συνολικό αριθμό εργασιών που ανέθεσε στον slave, το όνομα του τελευταίου url που ανέθεσε στον slave κ.α.

Η λίστα με τις προς εκτέλεση εργασίες (δηλαδή η λίστα με τα urls) δίνεται ως αρχείο ASCII με ένα url ανά γραμμή. Το όνομα του αρχείου δίνεται ως παράμετρο κατά τη κλήση του master.

Η κλήση του master γίνεται ως εξής:

`master filename [port]`,

όπου

- `filename` : το όνομα του αρχείου που περιέχει τη λίστα των εργασιών
- `port` : προαιρετικά ο αριθμός του port στο οποίο αναμένει μηνύματα ο master (εάν δεν παρέχεται τότε χρησιμοποιείται η οριζόμενη από το ίδιο το πρόγραμμα στο αρχείο επικεφαλίδων).

Η κλήση του slave γίνεται ως εξής:

```
slave slave_port [master_addr [master_port]],
```

όπου

- `slave_port` : ο αριθμός του port στο οποίο αναμένει σύνδεση ο slave (για το κανάλι εντολών). Αν είναι 0, ο slave χρησιμοποιεί οποιοδήποτε port είναι διαθέσιμο από το σύστημα.
- `master_addr` : προαιρετικά η IP διεύθυνση του master (διαφορετικά θα ορίζεται από το ίδιο το πρόγραμμα ως σταθερή).
- `master_port` : προαιρετικά ο αριθμός του port στο οποίο αναμένει μηνύματα ο master (εάν δεν παρέχεται τότε χρησιμοποιείται η οριζόμενη από το ίδιο το πρόγραμμα στο αρχείο επικεφαλίδων).

Κατά την λειτουργία τόσο του master όσο και του slave, εμφανίζονται στις οθόνες των υπολογιστών όπου εκτελούνται πληροφορίες σχετικά με τις συνδέσεις, τα μηνύματα που ανταλλάσσονται, οι εγγραφές νέων slaves (όσον αφορά τον master) ή η πορεία των εργασιών (όσον αφορά τον slave).

Ειδικότερα όσον αφορά τον master ο χρήστης έρχεται σε επαφή με δύο μενού επιλογών: Ένα γενικό και ένα ειδικό για κάποιον slave που έχει επιλέξει. Η δομή τους μαζί με κάποιες εξηγήσεις, παρατίθεται παρακάτω:

Πρώτα το γενικό μενού επιλογών :

```
**********************************
*    GENERAL COMMANDS*
**********************************
* 1. PRINT TASK LIST          : Εκτύπωση της λίστας των εργασιών
* 2. PRINT SLAVE LIST         : Εκτύπωση της λίστας των slaves
* 3. SELECT SLAVE             : Επιλογή slave για αποστολή εντολών
* 4. QUIT PROGRAM             : Έξοδος από το πρόγραμμα
**********************************
```

Εφόσον ο χρήστης επιλέξει το 3, εμφανίζεται η λίστα όλων των slaves, από όπου επιλέγοντας έναν από αυτούς δίνοντας το ID του, κατόπιν εμφανίζεται το μενού ειδικών επιλογών για τον συγκεκριμένο slave :

```
***************************
* SLAVE COMMANDS*
***************************
* 1. GET url                  : Λήψη εγγράφου από τη διεύθυνση url
* 2. STOP                     : Αναστολή εργασίας
* 3. EXIT                     : Αναστολή εργασίας και τερματισμός λειτουργίας
* 4. STATUS                   : Αναφορά κατάστασης slave
***************************
```

Το αρχείο των επικεφαλίδων(inet.h)

```
/*********************************************************
* inet.h                                                 *
* Definitions File for master-slave application *
**********************************************/

/* Include all the necessary system header files */
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/file.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>
#include <math.h>
#include <memory.h>

/* Make the appropriate definitions for the application */
#define GET                1
#define STOP               2
#define EXIT               3
#define STATUS             4

#define HERE_I_AM          0
#define THANK_YOU          1
#define SORRY              2
#define OK                 3
#define LATER              4

#define DONE               1
#define BUSY               0
#define IDLE               1
#define MAXLINE            80
#define MAXMESG            512
#define TIMEOUT            3
#define MASTER_HOST_ADDR   "127.0.0.1"
#define MASTER_UDP_PORT    6060

/* Definition of the struct of each node of the slaves' list */
typedef struct slave_node
{
  int id;                  /* Slave's unique ID, used for internal purposes */
  u_short port;            /* Slave's TCP port.*/
  u_long ip_address;       /* Slave's ip address in u+long format */
  char *address;           /* The slave's ip address in dotted representation*/
  char *name_address;      /* The slave's domain name */
  int  total_cmds_no;      /* Number of commands assigned to the slave.*/
  char *last_url;          /* Last URL assigned to the slave for fetching.*/
  int  status;             /* The status of the slave, 0 for busy, 1 for idle*/
  struct slave_node *next; /* Pointer to next slave enlisted.*/
}slave;

/* Definition of the struct of each node of the tasks' list */
typedef struct task_node
{
  int  id;                 /* Task's unique ID, used for internal purposes */
  char url[MAXLINE+1];     /* The url of the file to be downloaded */
```

```
  int  status;             /* The status of the task, finished or not*/
  struct task_node *next;  /* Pointer to next task enlisted.*/
}task;

/************************************************************************
 * int error(errormsg)                                                  *
 * Error function that terminates application                           *
 * Parameters : errormsg - An explanatory string of the fatal error     *
 ************************************************************************/
int error(errormsg)
char *errormsg;
{
   printf("%s\n",errormsg);
   perror(0);
   exit(1);
}

/************************************************************************
 * int set_timeout (int cur_fd, unsigned int seconds)
 * Sets a timeout using select() for the socket passed as a parameter   *
 * Parameters : cur_fd - The file descriptor of a socket                *
 *              seconds - The seconds of the timeout                    *
 ************************************************************************/
int set_timeout (int cur_fd, unsigned int seconds)
{
  fd_set set;
  struct timeval timeout;
  int res;

  /* Initialize the file descriptor set. */
  FD_ZERO (&set);
  FD_SET (cur_fd, &set);

  /* Initialize the timeout data structure. */
  timeout.tv_sec = seconds;
  timeout.tv_usec = 0;

  /* select returns 0 if timeout, 1 if input available, -1 if error. Repeat
select if a signal interrupted*/
  while (((res = select (FD_SETSIZE,&set, NULL, NULL,&timeout)) == -1) &&
(errno == EINTR));
  return res;
}
```

Αρχείο master (master.c)

```
/*************************************************************
 * master.c                                                  *
 * The program of the master process in our application.     *
 * Responsible for handling the list of slaves and tasks,    *
 * accepting, registering and commanding new slaves,         *
 * and assigning new tasks to slaves                         *
 *************************************************************/

#include "inet.h"

/* Global variables for the head and tail of each list */
slave *slave_list_head, *slave_list_tail;
task *task_list_head, *task_list_tail;

/*********************************************************************
 * void add_slave(int id, u_long ip_address, char *address, char      *
 *   *name_address, u_short port)                                     *
 * Adds a new node at the end of the list of the slaves with values   *
 * of fields passed as parameters                                     *
 *********************************************************************/
void add_slave(int id, u_long ip_address, char *address, char *name_address,
u_short port)
{
  slave *temp;

  if (slave_list_head == NULL)
  /* The list is empty, insertion of the first node */

  {
    slave_list_head              = (slave *)malloc(sizeof(slave));
    slave_list_head->id          = id;
    slave_list_head->port        = port;
    slave_list_head->ip_address  = ip_address;
    slave_list_head->address     = (char *)malloc(strlen(address)+1);
    strcpy(slave_list_head->address, address);
    slave_list_head->name_address = (char *)malloc(strlen(name_address)+1);
    strcpy(slave_list_head->name_address,name_address);
    slave_list_head->total_cmds_no = 0;
    slave_list_head->last_url    = (char *)malloc(MAXLINE+1);
    strcpy(slave_list_head->last_url,"-");
    slave_list_head->next        = NULL;
    slave_list_tail              = slave_list_head;
  }
  else
  /* The list is not empty, add the node to the end of the list */
  {
    temp                         = (slave *)malloc(sizeof(slave));
    temp->id                     = id;
    temp->port                   = port;
    temp->ip_address             = ip_address;
    temp->address                = (char *)malloc(strlen(address)+1);
    strcpy(temp->address, address);
    temp->name_address           = (char *)malloc(strlen(name_address)+1);
    strcpy(temp->name_address,name_address);
    temp->total_cmds_no          = 0;
    temp->last_url               = (char *)malloc(MAXLINE+1);
    strcpy(temp->last_url,"-");
    temp->next                   = NULL;
    slave_list_tail->next        = temp;
    slave_list_tail              = temp;
  }
}
```

```
/*****************************************************************
 * int remove_slave(int id)                                      *
 * Removes a node from the list of slaves                        *
 * Parameters :  id - The unique id of the slave to be removed   *
 *****************************************************************/
int remove_slave(int id)
{
  slave *temp;
  slave *prev;
  int found = 0;

  if (slave_list_head == NULL)
  /* The list does not contain nodes */
  {
    printf("Slave list empty!\n");
    return 0;
  }
  else
  {
    prev = NULL;
    temp = slave_list_head;

    while ((temp != NULL) && !found)
    {
      if (temp->id != id)
      {
        prev = temp;
        temp = temp->next;
      }
      else
          found = 1;
    }
    if (!found)
      return 0;
    else
    {
      if (prev == NULL)
              /* Removal of the head node */
        slave_list_head = slave_list_head->next;
      else
              if (temp->next == NULL)
                     /* Removal of the tail of the list */
              slave_list_tail = prev;
       else
                     /* Removal of a middle node */
            prev->next = temp->next;
      return 1;
    }
  }
}

/*****************************************************************
 * slave *select_slave(int id)                                   *
 * Returns a pointer to a particular slave given its id          *
 * Parameters :  id - The unique id of the slave to be selected  *
 *****************************************************************/
slave *select_slave(int id)
{
  slave *temp;
  int found = 0;

  temp = slave_list_head;

  while ((temp != NULL) && !found)
  {
    if (temp->id == id)
    {
```

```
      found = 1;
      break;
    }
    temp = temp->next;
  }
  if (!found)
    return NULL;
  else
    return temp;
}

/*************************************************************************
 * slave *find_slave(char *address, u_short port)                        *
 * Returns a pointer to a particular slave given its address and TCP port*
 * Parameters :  address - The dotted ip address of the slave            *
 *               port - The TCP port of the slave                        *
 *************************************************************************/
slave *find_slave(char *address, u_short port)
{
  slave *temp;
  int found = 0;

  temp = slave_list_head;

  while ((temp != NULL) && !found)
  {
    if ((strcmp(temp->address, address) == 0) && (temp->port == port))
    {
      found = 1;
      break;
    }
    temp = temp->next;
  }
  if (!found)
    return NULL;
  else
    return temp;
}

/*****************************************************************
 * void update_slave(int id, char *url)                          *
 * Updates the values of url and commands assigned to a slave    *
 * Parameters :  id - The unique id of the slave to be updated    *
 *               url - The last url assigned to the slave        *
 *****************************************************************/
void update_slave(int id, char *url)
{
  slave *temp;

  if ((temp = select_slave(id)) == NULL)
    printf("Slave with id %d not found!\n", id);
  else
  {
    temp->last_url = url;
    temp->total_cmds_no = temp->total_cmds_no + 1;
  }
}

/*****************************************************************
 * void print_slave_list()                                       *
 * Traverses the whole list of slaves and prints its contents    *
 *****************************************************************/
void print_slave_list()
{
  slave *temp;

  if (slave_list_head == NULL)
  {
```

```
    printf("Slave list empty!\n");
    return;
  }

  temp = slave_list_head;

  while (temp != NULL)
  {
    printf("Slave ID:%d, port:%d, ip_address:%s - %s\n",temp->id,temp-
>port,temp->address,temp->name_address);
    printf("last_url:%s, total_cmds_no:%d\n",temp->last_url,temp-
>total_cmds_no);

    temp = temp->next;
  }
}

/*************************************************************
 * void empty_slave_list()                                   *
 * Initializes the list of slaves                            *
 *************************************************************/
void empty_slave_list()
{
  slave_list_head = slave_list_tail = NULL;
}

/*****************************************************************
 * void add_task(int id, char url[MAXLINE+1])                    *
 * Adds a new node at the end of the list of the tasks with values *
 * of fields passed as parameters                                *
 *****************************************************************/
void add_task(int id, char url[MAXLINE+1])
{
  task *temp;

  if (task_list_head == NULL)
  /* The list is empty. Insert head node */
  {
    task_list_head         = (task *)malloc(sizeof(task));
    task_list_head->id     = id;
    strcpy(task_list_head->url,url);
    task_list_head->status = !DONE;
    task_list_head->next   = NULL;
    task_list_tail         = task_list_head;
  }
  else
  /* Append node to the list */
  {
    temp = (task *)malloc(sizeof(task));
    temp->id               = id;
    strcpy(temp->url,url);
    temp->status           = !DONE;
    temp->next             = NULL;
    task_list_tail->next   = temp;
    task_list_tail         = temp;
  }
}

/*****************************************************************
 * task *select_task(int id)                                     *
 * Returns a pointer to a particular task given its id           *
 * Parameters :  id - The unique id of the task to be selected    *
 *****************************************************************/
task *select_task(int id)
{
  task *temp;
  int found = 0;
```

```
  temp = task_list_head;

  while ((temp != NULL) && !found)
  {
    if (temp->id == id)
    {
      found = 1;
      break;
    }
    temp = temp->next;
  }
  if (!found)
    return NULL;
  else
    return temp;
}

/*****************************************************************
 * task *find_task(char *url)                                    *
 * Returns a pointer to a particular task given its url          *
 * Parameters :  url - The url of the file for download          *
 *****************************************************************/
task *find_task(char *url)
{
  task *temp;
  int found = 0;

  temp = task_list_head;

  while ((temp != NULL) && !found)
  {
    if (strcmp(temp->url, url) == 0)
    {
      found = 1;
      break;
    }
    temp = temp->next;
  }
  if (!found)
    return NULL;
  else
    return temp;
}

/*****************************************************************
 * int all_tasks_done()                                          *
 * Examines if the list of tasks is exchausted, i.e. all the tasks       *
 * have been succesfully completed. Returns 0 if at least one            *
 * task is not completed, otherwise if all tasks are done returns 1.     *
 *****************************************************************/
int all_tasks_done()
{
  task *temp;
  int found = 0;

  temp = task_list_head;

  while ((temp != NULL) && !found)
  {
    if (temp->status != DONE)
    {
      found = 1;
      break;
    }
    temp = temp->next;
  }
  if (found)
    return 0;
```

```
  else
    return 1;
}

/*****************************************************************
 * void update_task(int id, int status)                          *
 * Updates the value of status of a task                         *
 * Parameters :  id - The unique id of the task to be updated    *
 *               status - The new value of status                *
 *****************************************************************/
void update_task(int id, int status)
{
  task *temp;

  if ((temp = select_task(id)) == NULL)
    printf("Task with id %d not found!\n", id);
  else
    temp->status = status;
}

/*****************************************************************
 * void print_task_list()                                        *
 * Traverses the whole list of tasks and prints its contents     *
 *****************************************************************/
void print_task_list()
{
  task *temp;

  if (task_list_head == NULL)
  {
    printf("Task list empty!\n");
    return;
  }

  temp = task_list_head;

  while (temp != NULL)
  {
    printf("Task ID:%d, url:%s",temp->id,temp->url);
    if ((temp->status) != DONE)
      printf("status:NOT DONE\n");
    else
      printf("status:DONE\n");
    temp = temp->next;
  }
}

/*****************************************************************
 * void empty_slave_list()                                       *
 * Initializes the list of tasks                                 *
 *****************************************************************/
void empty_task_list()
{
  task_list_head = task_list_tail = NULL;
}

/*****************************************************************
 * void slave_commands_menu()                                    *
 * Prints the menu of commands for a slave                       *
 *****************************************************************/
void slave_commands_menu()
{
    printf("\n") ;
    printf("*****************\n") ;
    printf("* SLAVE COMMANDS*\n") ;
    printf("*****************\n") ;
    printf("* 1. GET url    *\n") ;
    printf("* 2. STOP       *\n") ;
```

```
    printf("* 3. EXIT       *\n") ;
    printf("* 4. STATUS     *\n") ;
    printf("*****************\n") ;
}

/*****************************************************************
 * void general_commands_menu()                                  *
 * Prints the menu of general commands for the master process    *
 *****************************************************************/
void general_commands_menu()
{
    printf("\n") ;
    printf("*************************\n") ;
    printf("*    GENERAL COMMANDS   *\n") ;
    printf("*************************\n") ;
    printf("* 1. PRINT TASK LIST    *\n") ;
    printf("* 2. PRINT SLAVE LIST   *\n") ;
    printf("* 3. SELECT SLAVE       *\n") ;
    printf("* 4. QUIT PROGRAM       *\n") ;
    printf("*************************\n") ;
}

/*****************************************************************************
 * int read_slave_port(char buf[])                                           *
 * Extracts and returns the slave's port from the end of message passed      *
 * as parameter. The message has the format e.g. HERE_I_AM 4004              *
 * Parameters : buf - A string containing the registration message           *
 * received from slave from UDP channel                                      *
 *****************************************************************************/
int read_slave_port(char buf[])
{
  int i = 0, nCount = 0, a[10], nRet = 0, j, k, nTemp;

  for (i=0; i<10; i++)
    a[i]= '\0';

  while (buf[i] != '\n')
  {
    if (buf[i] > '9' && buf[i] < '0')
    {
      i++;
      continue;
    }

    a[nCount] = buf[i] - '0';
    nCount++;
    i++;
  }

  for (j=0 ; j < nCount ; j++)
  {
    nTemp = 1;
    for (k=0 ; k < nCount-j-1 ; k++)
      nTemp = nTemp * 10;

    nRet += a[j] * nTemp;
  }

  return nRet;
}
```

```
/*********************************************************
 * int main(int argc, char *argv[])                      *
 * The main procedure of the master program              *
 *********************************************************/
int main(int argc, char *argv[])
{
  char *filename;
  u_short UDP_port, slave_tcp_port;
  FILE *tasks_file;
  char url[MAXLINE+1], mesg[MAXMESG], recvmesg[MAXMESG+1];
  int task_id, slave_id, sel_sl_id;
  int cli_len, recvlength, time_expired, nbytes;
  int UDP_sockfd, TCP_sockfd;
  struct sockaddr_in UDP_serv_addr, UDP_cli_addr, TCP_slav_addr;
  u_long slave_ipaddr;
  slave *temp_slave;
  task *temp_task;
  int menu_selection, slave_selection, task_selection;
  struct hostent *hp;

  if ((argc < 2) || (argc > 3))
    error("Master Usage: master <tasks_filename> [udp_port]\n");

  filename = argv[1];

  if (argc > 2)
    UDP_port=atoi(argv[2]);
  else
    UDP_port = MASTER_UDP_PORT;

  printf("Master listens to udp_port:%d\n", UDP_port);

  if ((tasks_file = fopen(filename,"r"))==NULL)
    error("Master:Error opening file with tasks!\n");

  /* Initialize the list of tasks */
  empty_task_list();

  task_id = 0;

  while ( fgets(url, MAXLINE+1, tasks_file) != NULL )
  {
    printf("URL:%s",url);
    task_id = task_id + 1;
    add_task(task_id, url);
  }

  fclose(tasks_file);

  print_task_list();

  /*  Open a UDP socket  */

  if ((UDP_sockfd = socket(AF_INET, SOCK_DGRAM, 0))<0)
    error("Master:Can't open datagram socket\n");

  printf("Master:Opened UDP socket with descriptor:%d\n",UDP_sockfd);

  /*  Bind local address for UDP socket */

  bzero((char *) &UDP_serv_addr, sizeof(UDP_serv_addr));
  UDP_serv_addr.sin_family      = AF_INET;
  UDP_serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  UDP_serv_addr.sin_port        = htons(UDP_port);

  if (bind(UDP_sockfd, (struct sockaddr *) &UDP_serv_addr,
sizeof(UDP_serv_addr))<0)
    error("Master:Can't bind local address\n");
```

```
printf("Master:Succesfully bound address for UDP socket\n");

empty_slave_list();

slave_id = 0;

while (!all_tasks_done())
{
  time_expired = set_timeout(UDP_sockfd, TIMEOUT);
  if (time_expired == -1)
    error("Master:Select error\n");
  else
    if (time_expired == 0)
    {
      printf("Master:Time waiting new slave registrations has expired!\n");
      printf("Master:Continuing...\n");

      do
      {
        general_commands_menu();
        printf("Select a number of your choice:") ;
        scanf("%d", &menu_selection);
      }while (menu_selection < 1 || menu_selection > 4) ;

      switch (menu_selection)
     {
     case 1:{
            if (task_list_head == NULL)
              printf("Task list empty!\n");
            else
              print_task_list();
            break;
            }
     case 2:{
             if (slave_list_head == NULL)
                printf("Slave list empty!\n");
             else
                print_slave_list();
             break;
             }
     case 3:{
             if (slave_list_head == NULL) {
                 printf("Slave list empty!\n");
                  break;
             } else
                print_slave_list();
             do
             {
               temp_slave = NULL ;
               printf("Give slave id:");
               scanf("%d", &sel_sl_id);
              } while ((temp_slave = select_slave(sel_sl_id)) == NULL);

            do
            {
              slave_commands_menu();
              printf("Select a number of your choice:") ;
              scanf("%d", &slave_selection);
             } while (slave_selection < 1 || slave_selection > 4) ;

           /* Connect to the slave to send him a command.  Fill in the
           structure TCP_slav_addr with the address of the slave */

            bzero((char *) &TCP_slav_addr, sizeof(TCP_slav_addr));
            TCP_slav_addr.sin_family         = AF_INET;
            TCP_slav_addr.sin_addr.s_addr    = htonl(temp_slave->ip_address);
            TCP_slav_addr.sin_port           = htons(temp_slave->port);
```

```
              /* Open a TCP socket */
              if ((TCP_sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
                    error("Master:Can't open TCP stream socket\n");

            printf("Master:Opened TCP socket with descriptor:%d\n",TCP_sockfd);

              /* Now connect to the slave */

              if (connect(TCP_sockfd, (struct sockaddr*) &TCP_slav_addr,
sizeof(TCP_slav_addr)) < 0)
              error("Master:Can't connect to slave with id:%d",temp_slave->id);

             printf("Master:Connected to TCP socket\n");

              switch (slave_selection)
              {
              case GET:{
                    printf("GET\n");
                    print_task_list();
                    do
                    {
                          temp_task = NULL;
                          printf("Select task id:");
                          scanf("%d", &task_selection);
                    } while ((temp_task = select_task(task_selection)) ==
NULL);

                    bzero(mesg,sizeof(mesg));
                    strcpy (mesg,"GET ");
                    strcat(mesg,temp_task->url);
                    printf("Message to be sent to TCP socket: %s\n",mesg);

                    update_slave(temp_slave->id, temp_task->url);

                    break;
                    }
              case STOP:{
                         printf("STOP\n");
                         bzero(mesg,sizeof(mesg));
                        strcpy (mesg,"STOP\n");
                        printf("Message to be sent to TCP socket: %s\n",mesg);
                           break;
                    }
              case EXIT:{
                         printf("EXIT\n");
                         bzero(mesg,sizeof(mesg));
                         strcpy (mesg,"EXIT\n");
                         printf("Message to be sent to TCP socket: %s\n",mesg);

                        if ( remove_slave(temp_slave->id) != 1 )
                         printf("Slave with id:%d not found in list!\n",
temp_slave->id);

                        break;
                    }
              case STATUS:{
                         printf("STATUS\n");
                         bzero(mesg,sizeof(mesg));
                         strcpy (mesg,"STATUS\n");
                         printf("Message to be sent to TCP socket: %s\n",mesg);

                         break;
                             }
              }

              if ((nbytes = write(TCP_sockfd, mesg, (strlen(mesg)))) <0)
                error("Master: write error in stream socket");
```

```
            printf("Message sent over TCP socket: %s of bytes
%d\n",mesg,nbytes);

            /* Read a line from the socket - the answer of the slave */

            bzero(recvmesg, MAXMESG);

            nbytes = read(TCP_sockfd, recvmesg, MAXMESG-1);

            if (nbytes < 0)
              error("Master: read error in stream socket");

            printf("Master:Message received:%s\n", recvmesg);
            close(TCP_sockfd);

               break;
          }
      case 4:{
              if (slave_list_head != NULL)
              {
                printf("Exiting all slaves!!!\n");

                temp_slave = slave_list_head ;

                while (temp_slave != NULL)
                {
               /* Connect to the slave to send him a command. Fill in the
               structure TCP_slav_addr with the address of the slave */

                 bzero((char *) &TCP_slav_addr, sizeof(TCP_slav_addr));
                 TCP_slav_addr.sin_family      = AF_INET;
                 TCP_slav_addr.sin_addr.s_addr= htonl(temp_slave->ip_address);
                 TCP_slav_addr.sin_port        = htons(temp_slave->port);

                 /* Open a TCP socket */
                 if ((TCP_sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
                          error("Master:Can't open TCP stream socket\n");

                  printf("Master:Opened TCP socket with descriptor:
%d\n",TCP_sockfd);

                  /* Now connect to the slave */

                 if (connect(TCP_sockfd, (struct sockaddr*) &TCP_slav_addr,
sizeof(TCP_slav_addr)) < 0)
                           error("Master:Can't connect to slave with
id:%d",temp_slave->id);

            printf("Master:Connected to TCP socket\n");
            bzero(mesg,sizeof(mesg));
            strcpy (mesg,"EXIT");

             if ((nbytes = write(TCP_sockfd, mesg, (strlen(mesg)))) <0)
                error("Master: write error in stream socket");

             printf("Message sent over TCP socket: %s of bytes
%d\n",mesg,nbytes);

              /* Read a line from the socket - the answer of the slave */

                 bzero(recvmesg, MAXMESG);
                 nbytes = read(TCP_sockfd, recvmesg, MAXMESG-1);

                 if (nbytes < 0)
                    error("Master: read error in stream socket");
                 printf("Master:Message received:%s\n", recvmesg);
```

```
            close(TCP_sockfd);

             temp_slave = temp_slave->next;
                }
              }
              else
                printf("Slave list empty!\n");

              printf("Terminating!!!\n");

              close(UDP_sockfd);

              exit(0);
               }
       }

      }
      else
      {
       /*  Receive from the UDP_socket the registration and other messages sent
       from slaves  */

        cli_len = sizeof(UDP_cli_addr);
       bzero(mesg, MAXMESG);
        if ((recvlength = recvfrom(UDP_sockfd, mesg, MAXMESG, 0, (struct
sockaddr *) &UDP_cli_addr, &cli_len)) < 0)
         error("Master:Receive from UDP socket error\n");

        printf("Master:Message received from UDP socket: %s\n",mesg);

        slave_tcp_port =  (u_short)read_slave_port(mesg) ;

       /* Convert from network to host representation */
        slave_ipaddr = ntohl(UDP_cli_addr.sin_addr.s_addr);

        if (!strncmp (mesg, "HERE_I_AM",9))
        {
       /* It is a slave registration message. Get the TCP port of the slave and
       the IP address from sockadrr_in struct*/

          printf("Master:New slave registration with TCP port:%d ip_address:%s
\n",slave_tcp_port, inet_ntoa(UDP_cli_addr.sin_addr));

          slave_id = slave_id + 1;

         hp = gethostbyaddr((char *) &UDP_cli_addr.sin_addr,
sizeof(UDP_cli_addr.sin_addr), AF_INET);

        add_slave(slave_id, slave_ipaddr, inet_ntoa(UDP_cli_addr.sin_addr),
(char *) hp->h_name, slave_tcp_port);
         print_slave_list();
       }
        else
          if (!strncmp (mesg, "THANK_YOU",9))
          {
            if ((temp_slave =
find_slave(inet_ntoa(UDP_cli_addr.sin_addr),slave_tcp_port)) == NULL)
              printf("Master:Slave with TCP port:%d ip_address:%s not
found\n",slave_tcp_port, inet_ntoa(UDP_cli_addr.sin_addr));
            else
            {
              printf("Master:Slave with TCP port:%d ip_address:%s
found\n",slave_tcp_port, inet_ntoa(UDP_cli_addr.sin_addr));

              printf("Master:Slave last url:%s\n", temp_slave->last_url);
              if ((temp_task = find_task(temp_slave->last_url)) == NULL)
                printf("Master:URL:%s not found in task list\n",temp_slave-
>last_url);
```

```
              else
              {
                    update_task(temp_task->id,DONE);
                    printf("Master:Task with id:%d status changed to
DONE\n",temp_task->id);
              }
            }
          }
          else
            if (!strncmp (mesg, "SORRY",5))
            {

            }
      }
  }

  printf("Task list done!!!\n");

  if (slave_list_head != NULL)
  {
   printf("Exiting all slaves!!!\n");

   temp_slave = slave_list_head ;

   while (temp_slave != NULL)
   {
       /* Connect to the slave to send him a command. Fill in the structure
       TCP_slav_addr with the address of the slave */

     bzero((char *) &TCP_slav_addr, sizeof(TCP_slav_addr));
     TCP_slav_addr.sin_family    = AF_INET;
     TCP_slav_addr.sin_addr.s_addr      = htonl(temp_slave->ip_address);
     TCP_slav_addr.sin_port             = htons(temp_slave->port);

     /* Open a TCP socket */

     if ((TCP_sockfd = socket(AF_INET, SOCK_STREAM, 0))<0)
       error("Master:Can't open TCP stream socket\n");

     printf("Master:Opened TCP socket with descriptor:%d\n",TCP_sockfd);

     /* Now connect to the slave */

     if (connect(TCP_sockfd, (struct sockaddr*) &TCP_slav_addr,
sizeof(TCP_slav_addr)) < 0)
       error("Master:Can't connect to slave with id:%d",temp_slave->id);

     printf("Master:Connected to TCP socket\n");

     bzero(mesg,sizeof(mesg));

     strcpy (mesg,"EXIT");

     if ((nbytes = write(TCP_sockfd, mesg, (strlen(mesg)))) <0)
       error("Master: write error in stream socket");

     printf("Message sent over TCP socket: %s of bytes %d\n",mesg,nbytes);


     /* Read a line from the socket - the answer of the slave */

     bzero(recvmesg, MAXMESG);

     nbytes = read(TCP_sockfd, recvmesg, MAXMESG-1);

     if (nbytes < 0)
       error("Master: read error in stream socket");
```

```
     printf("Master:Message received:%s\n", recvmesg);

     close(TCP_sockfd);

     temp_slave = temp_slave->next;
    }
  }
  else
    printf("Slave list empty!\n");

  printf("Terminating!!!\n");

  close(UDP_sockfd);

  exit(0);
}
```

Αρχείο slave (slave.c)

```
/************************************************************
 * slave.c                                                  *
 * The program of the slave process in our application.     *
 * Responsible for registering to the master process,       *
 * and executing the tasks assigned from master, that is    *
 * downloading the files from the urls sent from master     *
 ************************************************************/

#include "inet.h"

/* Global variables for the slave program */

int slv_status=IDLE,  /* The slave status (defined in inet.h) */
    childpid,         /* The process id of the child process */
    UDP_sockfd, TCP_sockfd, connTCP_sockfd;  /* socket descriptors */

struct sockaddr_in  master_UDP_addr,
                    master_TCP_addr,
                    slav_UDP_addr,
                    slav_TCP_addr;
char TCP_port[8] , url[MAXLINE];

/*************************************************************************
 * void writeUDP (int respond, int sockfd, struct sockaddr *pmast_addr, int  *
 *     addr_len, char TCP_port[8])                                          *
 * Sends a message to the master via UDP channel. The message to be sent is  *
 * specified by the value of var respond                                     *
 *************************************************************************/
void writeUDP (int respond, int sockfd, struct sockaddr *pmast_addr, int
addr_len, char TCP_port[8])
{
       char message[MAXMESG];
       int n;

       switch (respond) {
              case HERE_I_AM :{   strcpy(message, "HERE_I_AM ");
                                  break;
                              }
              case THANK_YOU :{   strcpy(message, "THANK_YOU ");
                                  break;
                              }
              case SORRY     :{   strcpy(message, "SORRY ");
                                  break;
                              }
                      }
       strcat(message, TCP_port);
       strcat(message, "\n");
       n = strlen(message);

       if (sendto(sockfd, message, n, 0, pmast_addr, addr_len) != n)
              error("Slave: Sendto error on UDP socket\n");
       printf("Message sent to UDP socket:%s\n",message);
}

/************************************************************************
 * void writeTCP (int respond, int sockfd)                               *
 * Sends a message to the master via TCP channel.                        *
 * Parameters : respond - A symbolic value determining the type of message *
                      sockfd - The TCP socket descriptor                 *
 ************************************************************************/
void writeTCP (int respond, int sockfd)
{
       char message[MAXMESG];
       int n;
```

```
	switch (respond) {
		case OK		:{	strcpy(message, "OK");
					break;
				}
		case LATER	:{	strcpy(message, "LATER");
					break;
				}
		case IDLE	:{	strcpy(message, "IDLE");
					break;
				}
		case BUSY	:{	strcpy(message, "BUSY");
					break;
				}
			}
	strcat(message, "\n");
	n = strlen(message);
	if (write(sockfd, message, n) != n)
		error("Slave: Write error in TCP socket\n");
	printf("Succesfully wrote to the TCP socket:%s\n",message);

}

/*************************************************************************
 * void extract_url_from_mesg (char message[MAXMESG])                    *
 * Accepts as input a message of the form "GET <url>" and extracts the url   *
 * part storing it in the global variable url                            *
 * Parameters : message -A string containing the command received from master*
 *************************************************************************/
void extract_url_from_mesg (char message[MAXMESG])
{
	int i=0, j=0;
	char c;

	bzero(url, MAXLINE);
	while (!(isspace(message[i++])));
	while ((c = message[i]) != '\n') {
		url[j++] = c;
		i++;
	}

}

/*************************************************************************
 * void notify_termination (int pid, int status)                         *
 * Used for asynchronous notification of the termination of the child process*
 * Parameters : pid - the process id of the child process finished       *
 *                    status - The status of termination (succesful or not)  *
 *************************************************************************/
void notify_termination (int pid, int status)
{
	int respond;

	status = status >> 8;
	printf("Child with process ID %d finished with status %d\n", pid,
status);
	slv_status = IDLE;		/* Update slave status to idle */
	if (status == 0)
		respond = THANK_YOU;
	else if (status == 1)
		respond = SORRY;
	writeUDP(respond, UDP_sockfd, (struct sockaddr *) &master_UDP_addr,
sizeof(master_UDP_addr), TCP_port);

}
```

```
/*************************************************************************
 * void sigchld_handler (int signum)                                     *
 * Signal handler for the signal SIGCHLD issued by child process to slave*
 * when finished                                                         *
 *************************************************************************/
void sigchld_handler (int signum)
{
  int pid;
  int status;
  /* Wait for the child process without hanging the program */
  pid = waitpid (childpid, &status, WNOHANG);
  if (pid <= 0)     /* No child to be noticed*/
        return;
  notify_termination(pid,status);
}

/*************************************************************************
 * void sigterm_handler (int signum)                                     *
 * Signal handler for the signal SIGTERM issued by slave to child        *
*************************************************************************/
void sigterm_handler (int signum)
{
      exit(1);
}

/*****************************************************************
 * int main(int argc, char *argv[])                               *
 * The main procedure of the slave program                        *
 *****************************************************************/
main(int argc, char *argv[])
{
      int n, status, time_expired,
            mast_length,  /* The length of the struct sockaddr of master */
            respond;      /* The type of message to be sent */
      u_short master_UDP_port;   /* port read from command line */
      u_long master_ipaddr;
      char mast_request[MAXMESG];
      struct hostent  *hptr;        /*  result of host name lookup  */

      if ((argc < 2) ||(argc>4))
            error("Usage: slave <slave_port> [master_addr [master_port]\n");

      /* Get from command line the TCP port for us and if given the IP address
      and UDP port for master */

      bzero(TCP_port, 8);
      strcpy(TCP_port, argv[1]);
      if (argv[2]) {

      /*  Check whether the name or the IP address of the server is given */
            if ((hptr=gethostbyname(argv[2]))!=NULL)
                  bcopy((char *)hptr->h_addr, (char *) &master_ipaddr, hptr-
>h_length);
            else
                  master_ipaddr = inet_addr(argv[2]);

            if (argv[3])
                  master_UDP_port = atoi(argv[3]);
            else master_UDP_port = MASTER_UDP_PORT;

      } else

            master_ipaddr =  inet_addr(MASTER_HOST_ADDR);

      /*  Open a UDP socket  */
      if ((UDP_sockfd=socket(AF_INET, SOCK_DGRAM, 0))<0)
            error("Slave: Cannot open datagram socket\n");
```

```
	printf("Slave opened UDP socket with descriptor: %d\n",UDP_sockfd);

	/* Fill in the master_UDP_addr structure with the address of master*/

	bzero((char*) &master_UDP_addr, sizeof(master_UDP_addr));
	master_UDP_addr.sin_family = AF_INET;
	master_UDP_addr.sin_addr.s_addr  = master_ipaddr;
	master_UDP_addr.sin_port   = htons(master_UDP_port);

	/*  Bind any local address for us in UDP socket*/

	bzero((char *) &slav_UDP_addr, sizeof(slav_UDP_addr));
	slav_UDP_addr.sin_family   = AF_INET;
	slav_UDP_addr.sin_addr.s_addr     = htonl(INADDR_ANY);
	slav_UDP_addr.sin_port            = htons(0);

	if (bind(UDP_sockfd, (struct sockaddr *) &slav_UDP_addr,
sizeof(slav_UDP_addr))<0)
		error("Slave: Cannot bind local address for UDP socket\n");
	printf("Successfully bound address for UDP socket\n");


	/*  Send the registration message to the UDP socket  */

	respond = HERE_I_AM;
	writeUDP(respond, UDP_sockfd, (struct sockaddr *) &master_UDP_addr,
sizeof(master_UDP_addr), TCP_port);

	/* Open a TCP socket */

	if ((TCP_sockfd=socket(AF_INET, SOCK_STREAM, 0))<0)
		error("Slave: Cannot open stream socket\n");
	printf("Successfully opened TCP socket with descriptor:
%d\n",TCP_sockfd);

	/* Bind our local address for TCP socket so that the master can send
commands to us */

	bzero((char *) &slav_TCP_addr, sizeof(slav_TCP_addr));
	slav_TCP_addr.sin_family   = AF_INET;
	slav_TCP_addr.sin_addr.s_addr     = htonl(INADDR_ANY);
	slav_TCP_addr.sin_port            = htons(atoi(TCP_port));

	if (bind(TCP_sockfd, (struct sockaddr *) &slav_TCP_addr,
sizeof(slav_TCP_addr))<0)
		error("Slave: Cannot bind local address for TCP socket\n");
	printf("Successfully bound address for TCP socket\n");

	/* Listen to the TCP socket creating a backlog of 5 */
	if (listen (TCP_sockfd, 5) <0 )
		error("Slave: Error listening on TCP port\n");

	while(1) {

	/* Set a timeout for listening to the TCP socket of 3 seconds*/
	time_expired = set_timeout(TCP_sockfd, 3);
	if (time_expired == -1)
		error("Slave: Select error\n");
	else if (time_expired == 1) {
			printf("Slave listening on TCP socket...\n");

			/* A connection is pending */
			mast_length = sizeof(master_TCP_addr);

			if ((connTCP_sockfd = accept (TCP_sockfd, (struct sockaddr
*) &master_TCP_addr, &mast_length)) < 0)
				error("Slave: TCP Accept error");
```

```
                    printf("Connected Socket TCP descriptor:
%d\n",connTCP_sockfd);
                    printf("Connection in TCP socket from host with addr: %s
and port: %d\n", inet_ntoa(master_TCP_addr.sin_addr),
ntohs(master_TCP_addr.sin_port));

                    /* Read the master's command from the TCP socket */

                    bzero(mast_request, MAXMESG);
                    n = read(connTCP_sockfd, mast_request, MAXMESG);

                    if (n < 0)
                          error("Slave: Read error from TCP socket");
                    printf("Master request:%s\n", mast_request);

                    /* Do the appropriate actions according to the request */

                    if (!strncmp (mast_request, "GET", 3)) {
                    /* The master has commanded to get a file */
                          if (slv_status == IDLE) {
                          /* Slave is idle. Write OK to the TCP socket */
                                respond = OK;
                                writeTCP(respond, connTCP_sockfd);

                                /* Update slave status to busy */
                                slv_status = BUSY;

                          /* Extract url from message received from master */
                                extract_url_from_mesg(mast_request);
                                printf("Asked url is: %s \n",url);

                    /* Create a child process with fork to execute wget */
                                if ((childpid = fork()) < 0)
                                      error("Slave: Fork error");
                                else if (childpid == 0) {
                                      /* child process */
                                      /* Close the sockets */
                                      close(TCP_sockfd);
                                      close(connTCP_sockfd);

                                /* Define the signal handler for SIGTERM */
                                      signal(SIGTERM, sigterm_handler);
                                      status = 0;

                    /* Exec returns only in failure with a value of -1 */
                                if (execlp("wget", "", url, NULL) == -1)
                                            exit(1);

                                }

                          /* Father process. Define handler for signal SIGCHLD
                          issued by child when finished */
                                signal (SIGCHLD, sigchld_handler);

                          } else {

                                /* The slave is busy. Write LATER */
                                respond = LATER;
                                writeTCP(respond, connTCP_sockfd);
                          }

                    } else if (!strncmp (mast_request, "STATUS", 6)) {

                    /* The master has requested the status of the slave*/
                          writeTCP(slv_status, connTCP_sockfd);

                    } else if (!strncmp (mast_request, "STOP", 4)) {
```

```
                        /* Master ordered to stop the current transfer */
                        respond = OK;
                        writeTCP(respond, connTCP_sockfd);

                        if (slv_status == BUSY) {

                    /* Kill the child process by sending a SIGTERM signal */
                            kill(childpid, SIGTERM);

                    /* Ignore signal SIGCHLD issued by child process */
                            signal (SIGCHLD, SIG_IGN);

                            /* Update slave status to idle */
                            slv_status = IDLE;

                            /* Write SORRY to the UDP channel */
                            writeUDP(SORRY, UDP_sockfd, (struct sockaddr
*) &master_UDP_addr, sizeof(master_UDP_addr), TCP_port);

                        }

                    } else if (!strncmp (mast_request, "EXIT", 4)) {
                        /* Master ordered to exit the program */
                        respond = OK;
                        writeTCP(respond, connTCP_sockfd);

                        if (slv_status == BUSY) {

                    /* Kill the child process by sending a SIGTERM signal */
                            kill(childpid, SIGTERM);

                        /* Ignore signal SIGCHLD issued by child process */
                            signal (SIGCHLD, SIG_IGN);
                        }
                        break;
                    }

                    /* Close the connecting socket */
                    printf("Closing TCP connection...\n");
                    close(connTCP_sockfd);
                }

        }  /* end infinite loop */

        /* Close the listening socket for TCP and the UDP socket*/
        printf("Exiting program...\n");
        close(TCP_sockfd);
        close(UDP_sockfd);
}
```