LINGI2346 - Distributed application design

Teacher : Marc Lobelle

Problem 1: Sockets

Université Catholique de Louvain

Group 24

Léonard Debroux
Thibaut Knop

2013-2014

1 Discussion

In this section, we'll discuss about the different choices we've made regarding the implementation of a transfer protocol.

1.1 Protocol choice

We chose to use TCP and not UDP for several reasons:

- The transmission errors and reordering are managed by TCP
- A stream oriented connection better suits the transmission of files.

In the implementation, the choice is made upon the socket creation.

1.2 Operating process of the server

The mode that seemed to be the better is the concurrent mode.
We chose this option to allow several client to connect to the server and use the transfer protocol.
The server thus behaves as telnet and forks upon receiving a connection attempt.

1.3 Allocation of the various functions to the server and to the client

Since it is a client-server architecture, the client initiates the connexion and the server replies to the requests of the client. Actually, the architecture of the protocol is very similar to the one used in telnet protocol.

The server waits to receive a connection request from a server. When a demand is received, the server forks and deletes the sockets that is linked to the client. The son deletes the socket that awaits connections and can begin to wait for the client to send a command to execute.

1.4 How to send and execute commands to/by the server

In order to send the command and be able to tackle the problem of having to deal with unknown length, we chose to create a header that is sent each time a command is sent. The header is a small structure that contains the type of the command and the length of the argument. If the command does not need an argument, that length is set to 0.

You can find the different possible values for the type in `header.h`
When a command enters client-side, the command is parsed to identify which one it is, and then, a header is sent accordingly. If the command is supposed to be followed by an argument, the argument is retrieved and sent to the server.

To cope with the endianness of the different systems architectures, we have to serialize some of our data before sending it through the channel. As it is only in the header that we use `uint32_t` instead of `char` everywhere else, we only need to serialize those variables.
This is done by calling the function `htonl()` on the varibles. We send the result of this conversion in our header to the other part. Upon reception of a header, the variable must be deserialize to be readable on the system. The funtion call to do so `ntohl()`.

Server-side, upon the reception of a header, depending on the type of command, either it is executed and the result is sent back, or the server waits to receive an argument, and then, upon reception, executes the command. If the commands is such that the client is waiting for a response, the same mechanism as above is used : the server send a header to the client, containing a special type (either

GET_SIZE or ERRNO_RET). Then the client know how much bytes have to been read on the socket.

Note that a special type for the header is ERRNO_RET and is used by the server to return some information about a failure to the client.

1.5 How to transfer textual data of unspecified length from the client to the server or from the server to the client

As specified above, the use of a header allows the server (resp. the client) to read the exact right number of bytes specified in the header. More precisely, the server expects to receive headers, specifying the command to run. In the case of the PWD and LS commands, the server don't have to receive any other arguments from the clients. It executes directly the corresponding command and return the result to the client. In the other cases, the server read exactly the number of bytes specified in the header.

For the distant LS command, we had different way to return the entries from the current directory to the client :

- Concatenate all the entries into 1 single string of a specified length, typically a multiple of MSS (see next subsection), and precede this sending by a header containing the length of the string that contains the entries. However, if the concatenation of all entries into one string have a length superior to the specified length, the server has to cut the string into 2 packets to be sent, which can cause more complexity with the headers.
- Send back every entries on the socket to the client, until a last packet containing the char \n is read by the client (the char \n cannot be contained in files name). The last therefore knows that it doesn't have to wait for other entries anymore. Each entries is sent into a 256 bits initialized string, which allows the client to knows how much bytes it have to read at each time. The value of 256 bits actually corresponds to the maximum filename length, NAME_MAX, defined in limits.h.

We chose the second solution more because we hadn't thought to the first one! If we had a little more time, we would probably change for the first solution : it is probably best to avoid to send to much small packets when it is not required.

1.6 How to transfer a binary file of length unknown a priori

This topic refers directly to the usage of GET and PUT. As the behaviour of the two commands is very similar as they basically perform the same action except from the direction of the information, we'll describe the behavior of PUT only.
The steps that are in emphasis are the one that allow us to send unknown length data.

The actions performed by the client are the following:

- *header sent to inform of the command and of the length of the filename*;
- message sent to give the filename;
- file is opened in binary read mode;
- *header sent to tell the number of full size packets to be sent*;
- loop sending full size packets;
- *header informing of the length of the last packet sent*;
- message containing the last packet.

The actions performed by the server are the following:

- recognize the command to execute;

- receive the filename;
- *receive the number of full size packets to be received*;
- create/open a file in binary write mode with the right filename;
- loop to receive the full size packets;
- *receive a header informing of the length of the last packet to receive*;
- receive the last packet of the file.

The size for the packets is set at 1072 bytes as it corresponds to two time the maximum segment size that ipv4 host are required to be able to handle.

1.7 Can one do something useful with the OOB data within the framework of this problem ?

Out of Band data could be used in this project to implement a channel to send priority messages. Those messages could be, for instances, control messages. If one wanted to interrupt a file transfert, that could be done by sending a messages saying so on that channel. However, in our case, the client send the command and then synchronize on an answer from the server. Therefore, OOB data are less interesting, since it is mainly used to send a control message independently of the other data?

2 Presentation

2.1 Protocols implementing the various functions

We'll describe here how we implemented the different commands.

- `ls, pwd, cd` - Those are implemented almost the same way. As explained above, a header is sent and depending on the presence of an argument, a packet is sent to give the argument. The reply is than sent back to the client.

- `get, put` - Those protocols are implemented the same way, except that the direction client-server changes from one to the other. The detail of the protocol is given in the section 1.6.

2.2 User guides of the client and server

Server The only command you can perform for the server is to run it by launching `./myftpd`. All the commands and responses the server will perform are sent from and to the client.

Client As specified in the functional description of the application, the client is launch by running `./myftp` in the right corresponding directory (see next Subsection). The commands available are :

- `pwd`: Display the current directory (of the environment) of the server
- `lpwd`: Display the current directory (of the environment) of the client
- `cd dir`: Change the current directory of the server into `dir`. Not that the usage of relative and absolute paths, as well as ˜ paths are accepted.
- `lcd dir`: Change the current directory of the client into `dir`. Not that the usage of relative and absolute paths, as well as ˜ paths are accepted.
- `ls`: Display the contents of the current directory of the server.
- `lls`: Display the contents of the current directory of the client.
- `get file`: The file `file` is copied from the current directory of the server towards the current directory of the client.

- `put file`: The file `file` is copied from the current directory of the client towards the current directory of the server.
- `bye`: Close the file transfer session

2.3 The source code directory and instruction for building the executable files

The directory containing the source codes is named `Mission2-Group24`.
It contains the following files :

- `header.h`: contains the typedef declaration of the structure `msgHeader`, as well as the type of possible message, encoded as a `int`.
- `utils.c`: contains a set of methods which are useful for both client and server. All the methods are designed to be used and called either by the client and the server. Note that the method `sendType`, we do not forget to call `htonl` to format the `int` variables from the *host* layer to the *network* layer.
- `myftp.c`: contains the code for the client. It works as telnet and has a similar architecture code.
- `myftpd.c`: contains the code for the server. It works as telnetd and has a similar architecture code.
- `Makefile`: Automate the compilation and the linkage of the .c sources to produce executable files, `myftp` and `myftpd` respectively.

In order to build the executable files, all is needed is to run the `Makefile` by running the `make` command in the current directory (`Mission2-Group24`).

To run the client and the server, just run `./myftp` and `./myftpd` respectively, by starting the server first.

2.4 Commented listings of the programs

2.4.1 Client specific code

`../myftp/myftp.c`

```c
/*
 * Thibaut Knop & Lenoard Debroux
 * Group 24
 * INGI2146 - Mission 1
 * myftp.c
 */

#include "header.h"
#include "utils.h"
#include <sys/types.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>

```

```
//#define M2_ADDR "130.104.172.88"
//#define M2_ADDR "127.0.0.1"


int getStringLength(char*, char);
int fillString(char*, char*, int);
int sendMsg(char*, int);
int getString(char*, char**, char);
int cmdcmp(char*, char*);

main(argc, argv) int    argc; char   *argv[ ];
{
  char* addr;
  if(argc > 1){
    addr = argv[1];
  }
  else {
    addr = "127.0.0.1";
  }
  int sd1;
  struct sockaddr_in     m2;

  bzero((char *) &m2       , sizeof(m2));
  m2.sin_family     = AF_INET;
  m2.sin_addr.s_addr = inet_addr(addr);
  m2.sin_port       = htons(TELNETD_PORT);

  if ( (sd1 = socket(PF_INET, SOCK_STREAM, 0)) < 0){
    perror("socket error in telnet");
    exit(-1);
  }

  if (connect(sd1 , (struct sockaddr *) &m2 , sizeof ( m2 )) < 0){
    perror("connect error in telnet");
    exit(-1);
  }

  /*
   * Normally the buffer should be 4096 long, since the MAX_PATH is 4096.
   * However, for the purpose of this assignment, we think 512 is a good tradeoff
   * between user possibility and performance.
   */
  char buffer[512];
  msgHeader in_header;

  /*
   * Compares the content of the buffer (filled from stdin)
   * and performs the different operations
   */

  while(strcmp(buffer, "bye")) {
    printf(">> ");
    fgets(buffer,512,stdin);

    /*
     * Local command pwd : retrieve the current directory and print in on sdtout
     */
    if(cmdcmp("lpwd", buffer)){
      char *curr_dir;
      int i = getPwd(&curr_dir);
      if(!i){
        printf("%s\n", curr_dir);
      } else {
        fprintf(stderr, "Error : %s\n",strerror(i));
      }
```

```c
    }

    /*
     * Local command cd : change the current directory
     * Save first the current directory,
     * then retrieve the arg from the user input
     * and pass thoses references to the cd method (from utils.c)
     */
    else if(cmdcmp("lcd", buffer)){
      char * current;
      int i = getPwd(&current);
      char* arg;
      getArg("lcd", buffer, &arg);
      int j = cd(arg, &current);
      free(arg);
      if (j!=0){
        fprintf(stderr, "Error : %s\n",strerror(j));
      }
    }

    /*
     * Local command ls : retrieve entries from current path
     * Save first the current directory,
     * and pass that reference to the ls method (from utils.c)
     * with s = -1 (local command)
     */
    else if(cmdcmp("lls", buffer)){
      char *current;
      int i = getPwd(&current);
      if (i==0){
        int j = getLs(current, -1);
        if(j!=0){
          fprintf(stderr, "Error : %s\n",strerror(j));
        }
      } else {
        fprintf(stderr, "Error : %s\n",strerror(i));
      }
    }

    /*
     * Distant command pwd : return the current path from the environment of the server
     * Send a header containing the type of the message : PWD.
     * The length field of the header contains 0, since there is no need
     * for the server to read something. The type PWD is enough.
     * The client waits for a header from the server to know how much bytes to read
     */
    else if(cmdcmp("pwd", buffer)){
      sendType(sd1, PWD, 0);
      if(read(sd1, &in_header, sizeof(msgHeader))){
        int len = ntohl(in_header.length);
        if(ntohl(in_header.type) == GET_SIZE){
          read(sd1, buffer, len);
          printf("%s\n", buffer);
        }else{
          fprintf(stderr, "Error : %s\n",strerror(len));
        }
      }
    }

    /*
     * Distant command cd : change the current directory of the environment of the server
     * Retrieve the path from the user input,
     * Send a header containing the type of the command CD, and the length of the
```

```
         path to be read
     * The server will read the header, and then knows that he has to read a certain ↩
         amount of bytes
     * (indicated by the length field in the header)
     * Then send the arg to the server
     * The client waits for the response from the server, receiving first a header ↩
         from the server
     * who announces the length the client has to read.
     */
    else if(cmdcmp("cd", buffer)){
      char* arg;
      getArg("cd", buffer, &arg);

      sendType(sd1, CD, strlen(arg)+1);
      sendMsg(arg, sd1);
      if(read(sd1, buffer, 6)){
        if(strcmp(buffer, "ok!")){
          printf("Error: cd failed\n");
        }
      }
      free(arg);
    }

    /*
     * Distant command ls : retrieve entries from current path for the environment of↩
          the server
     * Send a header containg the type of the command, LS.
     * The length field of the header contains 0, since there is no need
     * for the server to read something. The type LS is enough.
     * The client waits for the response from the server : since the server writes ↩
         every entries he found,
     * the client read every entries until it reads end.
     */
    else if(cmdcmp("ls", buffer)){
      sendType(sd1, LS, 0);
      while(read(sd1, buffer, 256)){
        if(buffer[0] == 10){
          read(sd1, &in_header, sizeof(msgHeader));
          if(ntohl(in_header.type) == ERRNO_RET && ntohl(in_header.length) != 0){
            fprintf(stderr, "Error : %s\n",strerror(ntohl(in_header.length)));
          }
          break;
        }
        printf("%s\n", buffer);
      }
    }

    /*
     * Distant command bye:
     * the client wishes to terminate
     * it informs the server and terminates
     */
    else if(cmdcmp("bye", buffer)){
      sendType(sd1, BYE, 0);
      close(sd1);
      break;
    }

    /*
     * Distant command get:
     * the client wants to retrieve a file from the server
     * the length of the filename is send in the header
     * the filename is send to the server
     * a new file is created client-side
     * the number of packets of length PACKET_SIZE is received in a header
     * then, those packets are received.
     * a header is received to tell the size of the last packet
     * then, the last packet is received.
     */
    else if(cmdcmp("get", buffer)){
      char* arg;
      getArg("get", buffer, &arg);
      if(strlen(arg) > 0){
        sendType(sd1, GET, strlen(arg)+1);
        sendMsg(arg, sd1);

        char *curr_dir;
        int i = getPwd(&curr_dir);
        if(!i){
          char str[strlen(curr_dir) + strlen(arg) + 1];
          strcpy(str, curr_dir);
          strcat(str, "/");
          strcat(str, arg);

          read(sd1, &in_header, sizeof(msgHeader));

          if(ntohl(in_header.type) != ERRNO_RET){
            FILE* f = NULL;
            f = fopen(str, "wb");

            int len = ntohl(in_header.length);

            int j;

            char received[PACKET_SIZE];
            for(j = 0; j<len; j++){
              read(sd1, received, PACKET_SIZE);
              fwrite(received, sizeof(received[0]), sizeof(received)/sizeof(received↩
                  [0]), f);
            }

            msgHeader end_header;
            read(sd1, &end_header, sizeof(end_header));

            if(ntohl(end_header.type) == GET_LAST){
              int elen = ntohl(end_header.length);
              if(elen != 0){
                char last[elen];
                read(sd1, last, elen);
                fwrite(last, sizeof(last[0]), sizeof(last)/sizeof(last[0]), f);
              } else {

              }
              printf("File received: %s\n", arg);
            }
            fclose(f);
          } else {
            fprintf(stderr, "Error : %s\n",strerror(ntohl(in_header.length)));
          }

        }
        free(arg);
      } else {
        printf("get requires an argument\n");
      }
    }

    /*
     * get
     * the length of the filename is sent in the header
```

```c
     * the filename is sent in the next packet
     * the client then sends the number of packet of PACKET_SIZE size
     * to be sent.
     * then, the file is splitted and sent
     * before sending the last packet, a header is sent to inform
     * the server of the length of the last packet
     * and the last packet is sent.
     */
    else if(cmdcmp("put", buffer)){
      char* arg;
      getArg("put", buffer, &arg);

      if(strlen(arg) > 0){
        sendType(sd1, PUT, strlen(arg)+1);
        sendMsg(arg, sd1);

        char *curr_dir;
        int i = getPwd(&curr_dir);
        if(!i){

          char str[strlen(curr_dir) + strlen(arg) + 1];
          strcpy(str, curr_dir);
          strcat(str, "/");
          strcat(str, arg);
          FILE* f = NULL;
          errno = 0;
          f = fopen(str, "rb");
          if(f != NULL){

            fseek(f, 0, SEEK_END);
            int size = ftell(f);
            rewind(f);

            int nb_packets = size/PACKET_SIZE;

            sendType(sd1, GET_SIZE, nb_packets);
            int j;
            for(j = 0; j<nb_packets; j++){
              unsigned char part[PACKET_SIZE];
              int n = fread(part, sizeof(part[0]), sizeof(part)/sizeof(part[0]), f);
              write(sd1, part, PACKET_SIZE);
            }

            int last_size = size-nb_packets*PACKET_SIZE;
            sendType(sd1, GET_LAST, last_size);
            if(last_size != 0){
              unsigned char part[last_size];
              int n = fread(part, sizeof(part[0]), sizeof(part)/sizeof(part[0]), f);

              write(sd1, part, last_size);
            }
            fclose(f);
            printf("File sent: %s\n", arg);
          }
          else {
            printf("File not found\n");
            sendType(sd1, ERRNO_RET, errno);
          }
        }

        free(arg);
      } else {
        printf("put requires an argument\n");
      }

    }
    else{
      printf("Unknown command\n");
    }

  }
}

int getString(char* data, char** result, char sep){
  int i = getStringLength(data, sep);
  char str[i+1];
  int j;
  for(j=0; j<i+1; j++){
    if(j == i){
      str[j] = 0;
    }
    else {
      str[j] = data[j];
    }
  }
  *result = str;
  return 0;
}

int getStringLength(char* str, char sep){
  int i=0;
  while(str[i] != sep){
    i++;
  }
  return i;
}

int cmdcmp(char* cmd, char* str){
  int i;
  for(i=0; i<strlen(cmd); i++){
    if(cmd[i]!=str[i]){
      return 0;
    }
  }
  if(cmd[i]!=0 && cmd[i]!=32 && cmd[i]!=10){
    return 0;
  }
  return 1;
}
```

2.4.2 Server specific code

../myftp/myftpd.c

```c
/*
 * Thibaut Knop & Lenoard Debroux
 * Group 24
 * INGI2146 - Mission 1
 * myftpd.c
 */

#include "header.h"
#include "utils.h"
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <arpa/inet.h>
#include <errno.h>


int sigflag;

void resquiescat(){
  int status;
  wait(&status);
  sigflag = 1;
} /*called by SIGCHLD event handler*/

main (argc, argv) int argc; char *argv[ ];
{
  int sdw, sd2,clilen,childpid;
  struct sockaddr_in m1,m2;

  sigset(SIGCHLD, resquiescat);

  if ( ( sdw = socket (PF_INET, SOCK_STREAM, 0)) < 0){
    perror("socket error in telnetd");
    exit(-1);
  }

  bzero((char *) &m2 , sizeof( m2 ));

  m2.sin_family      = AF_INET; /* address family : Internet */
  m2.sin_addr.s_addr = htonl(INADDR_ANY);
  m2.sin_port        = htons(TELNETD_PORT);

  if (bind(sdw, (struct sockaddr *) &m2 , sizeof( m2 )) < 0){
    perror("bind error in telnetd");
    exit(-1);
  }

  if(listen(sdw,5)<0){
    perror("listen error in telnetd");
    exit(-1);
  }

      clilen = sizeof(m1);
  for ( ; ; ) {
    sigflag = 0;

    if ((sd2 = accept(sdw, (struct sockaddr *) &m1 , &clilen))<0)
    {

      if(sigflag == 1)continue;
      perror("accept error in telnetd");
      exit(-1);
    }

    if((childpid = fork()) < 0){
       perror("fork error in telnetd");
       exit(-1);
    }

    else if (childpid == 0){
      close(sdw);

      msgHeader in_header;
```

```c

    /*
     * The server waits for a header to be sent.
     * Upon arrival, the exectution depends on the type of the header.
     */
    printf("Waiting for command\n");
    while(read(sd2, &in_header, sizeof(msgHeader))){
      int type = ntohl(in_header.type);
      int len = ntohl(in_header.length);
      /*
       * pwd: print working directory
       * Computed by the method getPwd
       * The result is sent back to the client
       */
      if (type == PWD){
        char *curr_dir;
        int i = getPwd(&curr_dir);
        printf("%s\n", curr_dir);
        if(!i){
          int j = sendType(sd2, GET_SIZE, strlen(curr_dir)+1);
          if(j==0){
            write(sd2, curr_dir, strlen(curr_dir)+1);
          }
        }else{
          int z = sendType(sd2, ERRNO_RET, i);
          if (z!=0){
            fprintf(stderr, "Error : %s\n",strerror(z));
          }
        }
      }

      /*
       * ls
       * the result of the ls command is computed by getLs
       * the result is sent to the client in the function call
       */
      else if (type == LS){
        printf("ls\n");
        char *curr_dir;
        int i = getPwd(&curr_dir);
        if(!i){
          getLs(curr_dir, sd2);
        }
      }

      /*
       * cd
       * the length of the path in argument is given in the header
       * the path is given in the packet that is read below
       * the current directoryis then changed
       */
      else if (type == CD){
        printf("cd\n");
        char buffer[len];

        read(sd2, buffer, len);

        char * current;
        int i = getPwd(&current);

        int j = cd(buffer, &current);
        if(j == 0){
          write(sd2, "ok!", strlen("ok!")+1);
        } else{
          write(sd2, "fail!", strlen("fail!")+1);
        }
      }

      /*
       * get
       * the length of the filename is given in the header
       * the next read packet states the file to get.
       * the server then sends the number of packet of PACKET_SIZE size
       * it will send to the client.
       * then, the file is splitted and sent
       * before sending the last packet, a header is sent to inform
       * the client of the length of the last packet
       * and the last packet is sent.
       */
      else if (type == GET){
        printf("get\n");
        char buffer[len];
        read(sd2, buffer, len);

        char *curr_dir;
        int i = getPwd(&curr_dir);
        if(!i){
          char str[strlen(curr_dir) + strlen(buffer) + 1];
          strcpy(str, curr_dir);
          strcat(str, "/");
          strcat(str, buffer);
          FILE* f = NULL;
          errno = 0;
          f = fopen(str, "rb");
          if(f != NULL){
            fseek(f, 0, SEEK_END);
            int size = ftell(f);
            rewind(f);
            int nb_packets = size/PACKET_SIZE;
            sendType(sd2, GET_SIZE, nb_packets);
            int j;
            for(j = 0; j<nb_packets; j++){
              unsigned char part[PACKET_SIZE];
              int n = fread(part, sizeof(part[0]), sizeof(part)/sizeof(part[0]), f)↩
                  ;
              write(sd2, part, PACKET_SIZE);
            }
            int last_size = size-nb_packets*PACKET_SIZE;
            sendType(sd2, GET_LAST, last_size);
            if(last_size != 0){
              unsigned char part[last_size];
              int n = fread(part, sizeof(part[0]), sizeof(part)/sizeof(part[0]), f)↩
                  ;

              write(sd2, part, last_size);
            }
            fclose(f);
            printf("File sent: %s\n", buffer);
          } else {
            sendType(sd2, ERRNO_RET, errno);
          }
        }
      }

      /*
       * put
       * the length of the filename is given in the header
       * a sent packet gives the name of the file
       * a new file is created.
       * the number of packets of length PACKET_SIZE is received in a header
```

```
         * then, those packets are received.
         * a header is received to tell the size of the last packet
         * then, the last packet is received.
         */
        else if (type == PUT){
          printf("put\n");
          char buffer[len];
          read(sd2, buffer, len);

          char *curr_dir;
          int i = getPwd(&curr_dir);
          if(!i){
            char str[strlen(curr_dir) + strlen(buffer) + 1];
            strcpy(str, curr_dir);
            strcat(str, "/");
            strcat(str, buffer);

            msgHeader in_header;
            read(sd2, &in_header, sizeof(msgHeader));

            if(ntohl(in_header.type) != ERRNO_RET){
              FILE* f = NULL;
              f = fopen(str, "wb");

              len = ntohl(in_header.length);

              int j;

              char received[PACKET_SIZE];
              for(j = 0; j<len; j++){
                read(sd2, received, PACKET_SIZE);
                fwrite(received, sizeof(received[0]), sizeof(received)/sizeof(↩
                    received[0]), f);
              }

              msgHeader end_header;
              read(sd2, &end_header, sizeof(end_header));

              if(ntohl(end_header.type) == GET_LAST){
                int elen = ntohl(end_header.length);
                if(elen != 0){
                  char last[elen];
                  read(sd2, last, elen);
                  fwrite(last, sizeof(last[0]), sizeof(last)/sizeof(last[0]), f);
                }
                printf("File received: %s\n", buffer);
              }
              fclose(f);
            } else {
              printf("Error: shouldn't be reached");
            }
          }
        }

        /*
         * bye
         * the server acknolodges that the client is out and terminates.
         */
        else if (type == BYE){
          printf("Client disconnected !\n");
          close(sd2);
          break;
        }
        else {
          printf("Error: shouldn't be reached");
        }
        printf("Waiting for command\n");
      }
      exit(0);
    }

    close(sd2); /* parent process */
  }
}
```

2.4.3 ../Shared code

../myftp/utils.c

```
/*
 * Thibaut Knop & Lenoard Debroux
 * Group 24
 * INGI2146 - Mission 1
 * utils.c
 */


#include "header.h"
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <arpa/inet.h>



//////////////////////////////////////////////////////////////////////////////
////////////////////////// AUX FUNCTIONS ///////////////////////////////
//////////////////////////////////////////////////////////////////////////////
/*
 * Replace a substring orig from str by a substring rep, if orig is in str.
 * Return a pointer to the new str.
 */
char *replace_str(char *str, char *orig, char *rep)
{
  static char buffer[4096];
  char *p;
  if(!(p = strstr(str, orig)))  // Is 'orig' even in 'str'?
    return str;
  strncpy(buffer, str, p-str); // Copy characters from 'str' start to 'orig' st$
  buffer[p-str] = '\0';
  sprintf(buffer+(p-str), "%s%s", rep, p+strlen(orig));
  return buffer;
}

/*
 * Return 0 if str starts with pre, 1 else.
 */
int startsWith(const char *str, const char *pre){
    size_t lenpre = strlen(pre),
           lenstr = strlen(str);
    return lenstr < lenpre ? -1 : strncmp(pre, str, lenpre) == 0;
}


/*
 * cmd represents the command type (cd, ls, lls, ...)
 * str is the input from the stdin
 * arg_result will contain the argument of the command.
 * Ex: cd path/test   cmd = cd    str = cd path/test   arg_result = path/test
 */
int getArg(char* cmd, char* str, char** arg_result){
  int len = strlen(cmd);
  int i=len;
  while(str[i]==32){
    i++;
  }
  char temp[strlen(str)];
  int j;
  for(j=0; j<strlen(str); j++){
    if(str[j+i]!=0 && str[j+i]!=32 && str[j+i]!=10){
      temp[j] = str[j+i];
    } else {
      break;
    }
  }
  temp[j]=0;
  errno = 0;
  *arg_result = malloc(strlen(temp)+1);
  if(errno == 0){
    strcpy(*arg_result, temp);
    return 0;
  }
  return errno;
}

/*
 * Sugar for the sending of a message msg on the socket designated by its
 * socket descriptor s
 */
 int sendMsg(char* msg, int s){
  errno = 0;
  write(s, msg, strlen(msg)+1);
  return errno;
}

/*
 * Sugar for the sending of a header h  on the socket designated by its
 * socket descriptor s .
 $ h is a of a msgHeader type, see in header.h for definition
 */
int sendHeader(msgHeader* h, int s){
  errno = 0;
  write(s, h, sizeof(h));
  return errno;
}

/*
 * Sugar for defining a header and sending it on the socket designated
 * by its socket descriptor s.
 * type and length are the fields the structure msgHeader h.
 *
 * To avoid any errors due to the endianness used by the operating system
 * we convert the int in the structure msgHeader from the host layer to the
 * network layer thanks to the function htonl()
 * Upon receiving a header, the inverse conversion must be done, thanks
 * to ntohl().
```

```c
 * It is done in myftp.c and myftpd.c
 */
int sendType(int s, int type, int length) {
  msgHeader h;
  h.length = htonl(length);
  h.type = htonl(type);
  int result = sendHeader(&h, s);
  return result;
}






////////////////////////////////////////////////////////////////////////////
////////////////////////// CORE FUNCTIONS //////////////////////////////////
////////////////////////////////////////////////////////////////////////////

/*
 * LS
 *
 * s = -1 if local command, for client
 * s = the number of the socket descriptor if distant command, for server
 * if distant : write each entry of the directory specified by path on the socket,
 * and send end when there is no more entries
 * if local: just print the entry
 */
int getLs(char* path, int s){
  DIR *dir;
  struct dirent *dent;
  dir = opendir(path);
  if(dir!=NULL)
  {
    int temp = 0;
    while((dent=readdir(dir))!=NULL) {
      if(s < 0) {
        printf ("%s\n", dent->d_name);
      } else {
        printf ("send[%s]\n", dent->d_name);
        errno = 0;
        write(s, dent->d_name, sizeof(dent->d_name));
        if(errno != 0){
          temp = errno;
          break;
        }
      }
    }
    if(s >= 0){
      char end[256];
      end[0] = 10;
      end[1] = 0;
      errno = 0;
      write(s, end, 256);
      errno = temp;
      sendType(s, ERRNO_RET, errno);
    }
  } else {
    printf("Wrong Path");
  }
  errno = 0;
  closedir(dir);
  return errno;
}

```

```c

/*
 * Change of Directory
 *
 * path is a pointer to a string representing the current directory
 * dir is the argument from the cd command
 * if dir starts with ~, carry out the addition of the HOME path before the dir
 * if dir starts with /, dir is an absolute path
 */

int cd(char* dir, char** path){
  if(startsWith(dir,"~") != 0){
    *path = getenv("HOME");
    if(strlen(dir) > 1){
      char temp[strlen(dir)];
      int i = 0;
      for (i;i<strlen(dir)-2;i++){
        temp[i] = dir[i+2];
      }
      temp[i] = 0;
      char str[strlen(temp) + strlen(*path) + 1];
      strcpy(str, *path);
      strcat(str, "/");
      strcat(str, temp);
      *path = malloc(strlen(str)+1);
      strcpy(*path,str);
    }
    else{
      *path = malloc(strlen(getenv("HOME")));
      strcpy(*path,getenv("HOME"));
    }
  } else if(startsWith(dir,"/") == 0){
    char str[strlen(dir) + strlen(*path) + 1];
    strcpy(str, *path);
    strcat(str, "/");
    strcat(str, dir);
    *path = malloc(strlen(str)+1);
    strcpy(*path,str);
  } else {
    *path = malloc(strlen(dir)+1);
    strcpy(*path,dir);
  }
  errno = 0;
  int i = chdir(*path);
  free(*path);
  return errno;
}

/*
 * Put the current directory in pwd
 */
int getPwd(char** pwd){
    char temp[4096]; // 4096 is the MAX_PATH length, so it is logical to take that value
  errno = 0;
    if (getcwd(temp, sizeof(temp)) != NULL){
         int size = 0;
         int i;
         for(i=0;temp[i]!= '\0';i++){
            size++;
         }
     char* temp2;
     temp2 = malloc(size*sizeof(char));
        for(i=0;i<=size;i++){
         temp2[i] = temp[i];
        }
    *pwd = temp2;
    }
    return errno;
}
```

../myftp/header.h

```
/*
 * Thibaut Knop & Lenoard Debroux
 * Group 24
 * INGI2146 - Mission 1
 * header.h
 */

#include <stdint.h>

/*
Each message sent by the client will be preceded by the sending of
a header. That header contains the length and the type of the message to receive,
Types are defined as follows
*/

#define PWD 1
#define LPWD 2
#define CD 3
#define LCD 4
#define LS 5
#define LLS 6
#define GET 7
#define PUT 8
#define BYE 9
#define GET_SIZE 70
#define GET_LAST 71
#define ERRNO_RET 10
#define TELNETD_PORT 7000
#define PACKET_SIZE 1072

struct msgHeader{
    uint32_t length;
    uint32_t type;
};
typedef struct msgHeader msgHeader;
```

../myftp/utils.h

```
/*
 * Thibaut Knop & Lenoard Debroux
 * Group 24
 * INGI2146 - Mission 1
 * utils.c
 */

int getLs(char* path, int s);

/*
 * getPwd place in pwd the current path
 */
int getPwd(char** pwd);
```