MyFileTransferProtocol(MFTP)

Documentation

Florin Eugen Rotaru

UAIC, Facultatea de Informatica, Iasi
30.12.2022

Abstract. We present the design of a server-client based application, which delivers a platform for file sharing between multiple users. The application follows the TCP/IP paradigm and is implemented with **C Socket Programming**. The report briefly describes the features of the application(i.e., the main operations that are available for the user and also for the server admin). Following other more technical details such as application structure, primitives used and logical principles, we explore a few ways in which the application may be improved and optimized, such as variations the concurrent server implementation and changes of data management.

Keywords: TCP/IP, File Transfer, FTP, Server-Client, Concurrent Networks.

1 Introduction

The main goal of this application is to provide a platform for real time file exchange between users. Each user interacts with the server only after logging into their account, which had been previously created. A user may have different permissions for file management, these permissions are handeld in a **read-write/read-only** manner. The account information such as username, password and permissions are serialized into a local static data structure. Of course, the user credentials will be transmitted in a secure way using encoding.

2 Technologies Used

We used the TCP/IP model which implements the TCP protocol at the Transport Level. This is because it provides a reliable end-to-end, bidirectional, congestion-controlled byte stream and preserves the order[1], all of which are particularly important when dealing with files and file exchange.

Since we mentioned that the users are able to exchange files in real time, the TCP implementation will be **concurrent.**

The implementation was done using **BSD Sockets API.** For Serialization and Deserialization of user data we used a *.json* format file, in which data was added, checked, and modified. The interaction of the C program and *.json* format was done with an open source library, namely parson.c/parson.h.

3 Application Architecture

The structure used is the **Client-Server Model**, where the **Server** will manage the requests, the users and also the local file database.

Each of the clients will be served by a different process, as to assure the concurrency of the server. The main actions of the **server** are:

1. Showing contents of the filesystem to the client.
2. Sending a file;
3. Receiving a file;
4. Registering a new user;
5. Creates a new directory;
6. Logging a user;
7. Changing the user data[1];

[1] :We also provide an utility *script* used to modify the read/write permissions of an user. The script is only available to the server administrator.

The **Client** has access to the following operations:

1. Request to download a file;
2. Request to upload a file;
3. Request to log in;
4. Request to create a directory;
5. Request to create an account;
6. Inspect the filesystem.
7. Reading the command instructions.

The operations take place after a command is run on the command line. The command documentation will be provided by the client application and will be available to all users that are using the application.

3.1 User Manual

Once the server is running, start the client executable:

```
$ ./executable <IP_Adress> <PORT>
```

Once you have managed to secure a connection, you have access to the following commands:

1. `$ login <username>` to initiate the login process. After running this command, the user will be asked to enter the password, which will be masked as input.

2. `$ sign up` to initiate the sign up process (creating a new account). After running this command, the user will be asked to choose a login and then a password, which will be masked as input.

3. `$ seefiles` to enter ***file interaction mode.*** This command is only available once the user has signed in on his account.

4. `$ sign out` signs out of the account.

5. `$ cmds` display instructions.

6. `$ quit` shuts down the client application.

File interaction mode offers the following commands:

1. `$ <directory_name>` navigates to the following directory, only works if the user is currently inspecting the parent directory.

2. `$ mkdir <directory_name>` creates a new directory in the currently inspected directory.

3. `$ get <filename>` use this command to download a file from the filesystem.

4. `$ upload <filename>` use this command to upload a local file to the remote filesystem.

5. `$ cmds` display instructions.

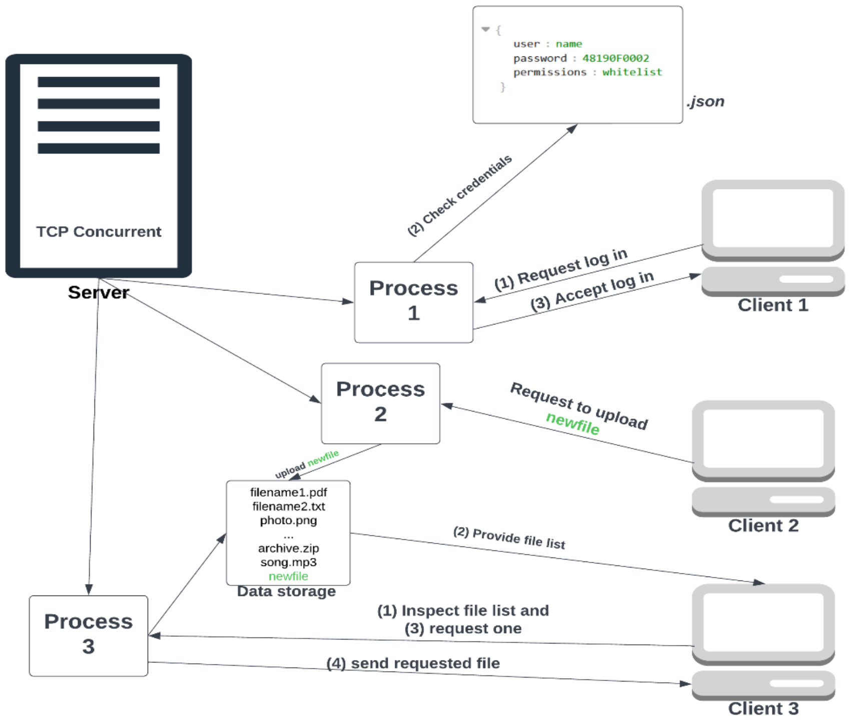


Figure 1 - Application Diagram

4 Implementation Details

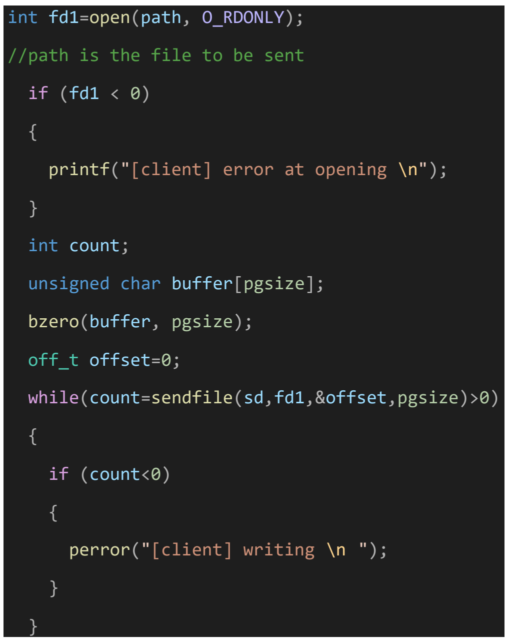
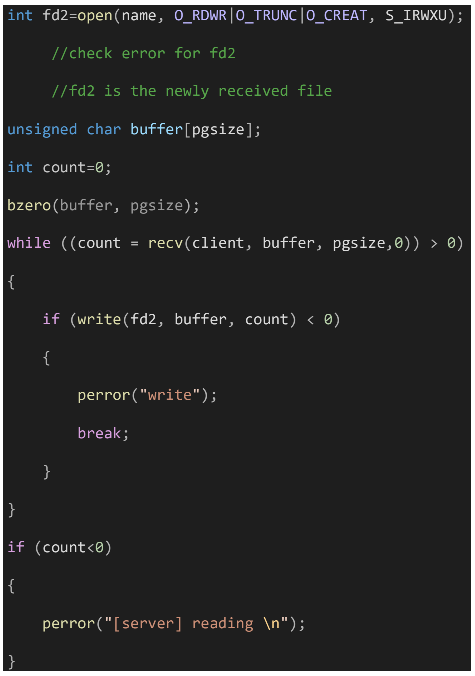
4.1 Exceptions

When the client interacts with the server, some exceptions may occur. They happen when the client makes an erroneous request, such as asking to download a file that the server does not have, or requesting something that is not permitted, such as setting an username already present in the users database. Here are **some** examples of possible exceptions:

- When creating a new account, the username is taken;
- The server already has a file with the same name
  -the name of the received file will be slightly modified;
- The client already has a file with the same name
  -the user will be asked if he wishes to overwrite the file;
- The client does not have permissions for the operation requested.
- Etc.

4.2 Sending Mechanism

The file transfer from **client to server** will be performed using method resembled by the following source code:

Client	Server
	

```
int fd1=open(path, O_RDONLY);
//path is the file to be sent
  if (fd1 < 0)
  {
    printf("[client] error at opening \n");
  }
  int count;
  unsigned char buffer[pgsize];
  bzero(buffer, pgsize);
  off_t offset=0;
  while(count=sendfile(sd,fd1,&offset,pgsize)>0)
  {
    if (count<0)
    {
      perror("[client] writing \n ");
    }
  }
```

```
int fd2=open(name, O_RDWR|O_TRUNC|O_CREAT, S_IRWXU);
     //check error for fd2
     //fd2 is the newly received file
unsigned char buffer[pgsize];
int count=0;
bzero(buffer, pgsize);
while ((count = recv(client, buffer, pgsize,0)) > 0)
{
    if (write(fd2, buffer, count) < 0)
    {
        perror("write");
        break;
    }
}
if (count<0)
{
    perror("[server] reading \n");
}
```

1. For sending the file, we used the **`sendfile()`** primitive:

   `ssize_t sendfile(int` *`out_fd`*`,` **`int`** *`in_fd`*`,` **`off_t *`***`offset`*`,` **`size_t`** *`count`***`),`**

   which copies data between one file descriptor and another. We used this primitive because it is more efficient than the combination of *read()* and *write()*,which would require transferring data to and from user space [2]. The primitive cannot be used for receiving the file, The *in_fd* argument must correspond to a file which supports *mmap(2)*-like operations (i.e., it cannot be a socket).

2. A very important aspect is ensuring that **concurrent** access to files is supported. That is, making sure that multiple processes can operate on the same critical section without resulting in undefined behaviour. To do this, we im-

plemented ***file-locks*** for reading and for writing, on the files that are accessed. For example:

sender	Receiver
`int sentfile=open(path, O_RDWR);`	`int newfile=open(filename,`
`unsigned char buffer[pgsize];`	`O_RDWR|O_TRUNC|O_CREAT, S_IRWXU);`
`bzero(buffer, pgsize);`	`unsigned char buffer[pgsize];`
`off_t offset=0;`	`bzero(buffer, pgsize);`
`struct flock lock;`	`struct flock lock;`
`lock.l_type=F_RDLCK;`	`lock.l_type=F_WRLCK;`
`lock.l_whence=SEEK_CUR;`	`lock.l_whence=SEEK_CUR;`
`lock.l_start=0;`	`lock.l_start=0;`
`lock.l_len=pgsize;`	`lock.l_len=pgsize;`
`fcntl(sentfile, F_SETLKW, &lock);`	`fcntl(getfile, F_SETLKW, &lock);`
`while(sendfile(client_connection,sentfile,&offset,`	`while ((count = read(client_connection,`
`4096)>0) {`	`buffer, sizeof buffer)) > 0)`
`lock.l_type=F_UNLCK;`	`{`
`fcntl(sentfile, F_SETLKW, &lock);`	`write(getfile, buffer,   count);`
`lock.l_type=F_RDLCK;`	`lock.l_type=F_UNLCK;`
`fcntl(sentfile, F_SETLKW, &lock`	`fcntl(getfile, F_SETLKW, &lock);`
`}`	`lock.l_type=F_WRLCK;`
`close (sentfile);`	`fcntl(getfile, F_SETLKW, &lock);`
`return 0;`	`}`

3. The working principle is the following: A process reading from or writing to a file will only block the file for a portion of the length of the modification, then unlock the portion. As these operations are iterative, the process will - lock, perform operations, and the unlock - repeatedly. If a process encounters a locked resource, it will wait until it is unlocked, due to the F_SETLKW flag. The same logic is applied for .json files.

4. We don't send the file all at once, because the file may be particularly large. We send byte-chunks. The number of bytes (*pgsize*) sent at each phase is equal to the memory page size of the system:

```
printf("The page size for this system is %ld bytes.\n",
          sysconf(_SC_PAGESIZE)); //4096, usually
```

4.3 Concurrency

Each client will be served by a fork-generated child-process, which runs concurrently with the process that makes the fork() call (parent process). Therefore, the server limit is the limit of creating new child-processes by the system.

A general structure of the server is similar to: [3]

```
int HandleSignup(int client_connection, struct user_record* new_user);
int HandleLogin(int client_connection, struct user_record* check_user);
int InspectDir(int client_connection, const char* dirname, char* dirpath);
int HandleNavigation(int client_connection, char* path, char* name);
int HandleMkdir(int client_connection, char* path, char current_name[DIRLEN]);
int SendFile(int client_connection, char* path);
int GetFile(int client_connection, char* path);
int Handle_ls(int client_connection, struct user_record *current_user);
```

```
int main()
{
    socket;//...
    sockaddr_in();//...
    bind();//...
    listen();
    while(1)
    {
        accept();//...
        int pid;
        if( (pid = fork()) == 0)
        {
            /* child process */
            close(socket);
            serve_client;
            disconnect_client;
            exit(0);
        }
        /* parent or error at fork*/
        close(socket);

    }
    return 0;
}
```

5 Conclusions & Further Optimization

5.1 Server Layout

The problem with the concurrent server model from above is the amount of CPU time it takes to `fork` a child for each client. Several ways to optimize it are:

1. **TCP Preforked Server:** Instead of generating one `fork` per client, the server preforks some number of children when it starts, and then the children are ready to service the clients as each client connection arrives:[3]

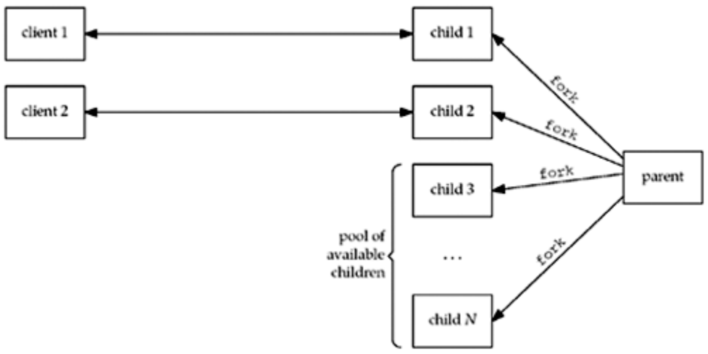
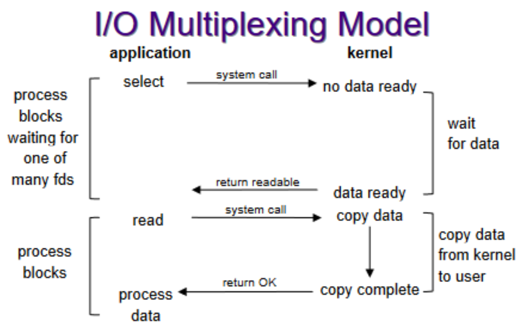


Figure 2 - Preforked Server [3]

2. **TCP Prethreaded Server:** The basic design of this server is to create a pool of threads and then let each thread call `accept`.
3. **I/O Multiplexing:** using primitives that allows checking from multiple in-puts: select() and poll(). [3] [4]



Compared to blocking model, the difference here is that the application can wait for more than one descriptors to be ready.

5.2 Data Storage

Data storage may also be optimized, employing Relational Data Bases, both for the user data, and for storing the files. The files could also be encrypted upon before being uploaded to the data base.

References

1. ANDREW S. TANENBAUM, DAVID J. WETHERALL, Computer Networks Fifth Edition.
2. Linux Manual Page, sendfile(2);
3. W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, Network Programming Volume 1, Third Edition: The Sockets Networking API
4. https://profs.info.uaic.ro/~computernetworks/cursullaboratorul.php