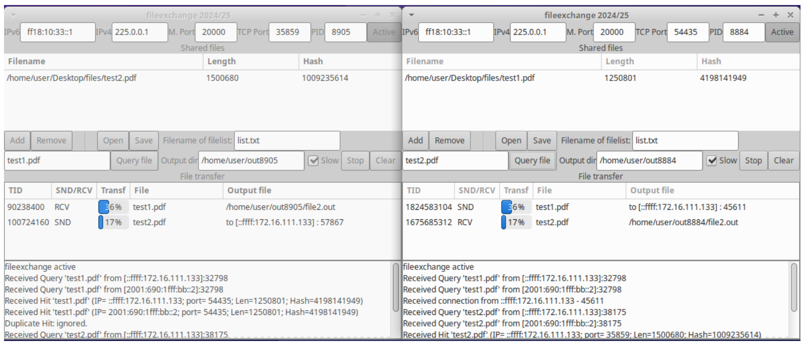
NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING



IPv6 ff18:10:33::1 IPv4 225.0.0.1 M. Port 20000 TCP Port 35859 PID 8905 Active

Filename	Length	Hash
/home/user/Desktop/files/test2.pdf	1500680	1009235614

Filename of filelist: list.txt
test1.pdf
Output dir /home/user/out8905 ✓ Slow

TID	SND/RCV	Transf	File	Output file
90238400	RCV	36%	test1.pdf	/home/user/out8905/file2.out
100724160	SND	17%	test2.pdf	to [::ffff:172.16.111.133] : 57867

```
fileexchange active
Received Query 'test1.pdf' from [::ffff:172.16.111.133]:32798
Received Query 'test1.pdf' from [2001:690:1fff:bb::2]:32798
Received Hit 'test1.pdf' (IP= ::ffff:172.16.111.133; port= 54435; Len=1250801; Hash=4198141949)
Received Hit 'test1.pdf' (IP= 2001:690:1fff:bb::2; port= 54435; Len=1250801; Hash=4198141949)
Duplicate Hit: ignored.
Received Query 'test2.pdf' from [::ffff:172.16.111.133]:38175
```

IPv6 ff18:10:33::1 IPv4 225.0.0.1 M. Port 20000 TCP Port 54435 PID 8884 Active

Filename	Length	Hash
/home/user/Desktop/files/test1.pdf	1250801	4198141949

Filename of filelist: list.txt
test2.pdf
Output dir /home/user/out8884 ✓ Slow

TID	SND/RCV	Transf	File	Output file
1824583104	SND	36%	test1.pdf	to [::ffff:172.16.111.133] : 45611
1675685312	RCV	17%	test2.pdf	/home/user/out8884/file2.out

```
fileexchange active
Received Query 'test1.pdf' from [::ffff:172.16.111.133]:32798
Received Query 'test1.pdf' from [2001:690:1fff:bb::2]:32798
Received connection from ::ffff:172.16.111.133 - 45611
Received Query 'test2.pdf' from [::ffff:172.16.111.133]:38175
Received Query 'test2.pdf' from [2001:690:1fff:bb::2]:38175
Received Hit 'test2.pdf' (IP= ::ffff:172.16.111.133; port= 35859; Len=1500680; Hash=1009235614)
```

Integrated Telecommunication Networks (RIT)

2024/2025

2nd Laboratory work: Locating and exchanging files on dual-stack networks

Classes 7 to 10

Mestrado integrado / Mestrado em Engenharia Eletrotécnica e de Computadores

https://tele1.deec.fct.unl.pt/rit

Luis Bernardo

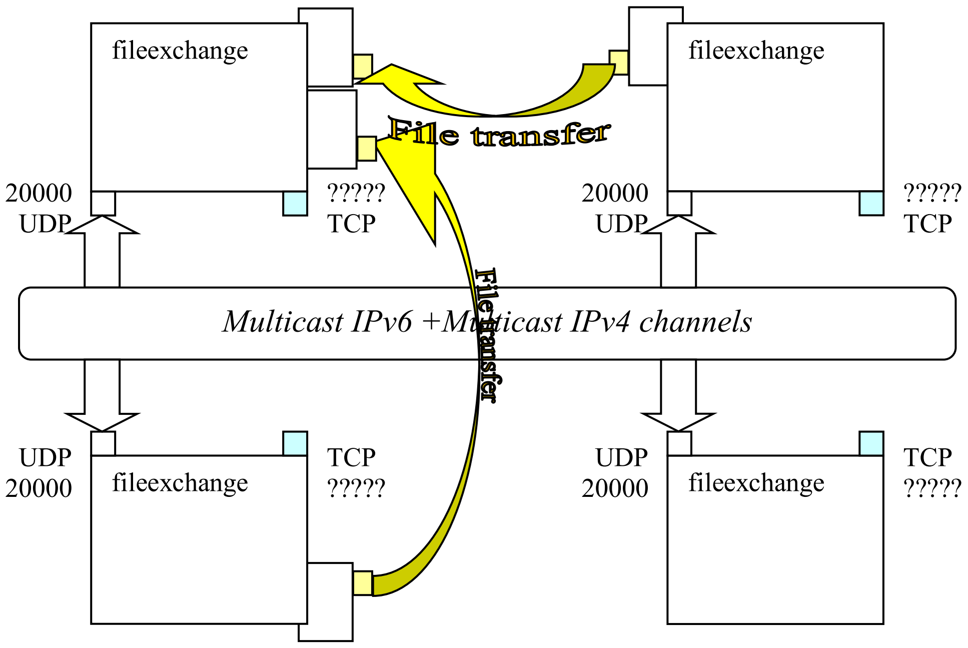
1. Objectives

Familiarization with programming using IPv4 and IPv6 addresses, the Gnome/Gtk+3.0 graphical environment, thread management mechanisms (*POSIX threads*), and synchronization between threads in the Linux operating system. The work involves developing an application for exchanging files in a dual-stack network. The application registers with IPv6 and IPv4 multicast addresses and sends the query packets to those addresses whenever the user searches for a file. Each participant receives the request and responds if they have the file. The application then downloads the file from one of the participants using a dedicated thread (through a TCP socket).

The work consists of the development of an application, ***fileexchange***.

2. Specifications

The *fileexchange* application requires two configuration parameters: the multicast IP addresses of the groups where the search is made (by default, the addresses "ff18:10:33::1" and "225.0.0.1" will be used) and the UDP port number (by default, the port 20000 will be used).



The *fileexchange* application performs the search for files using an IPv6 datagram socket. To receive requests to the groups, it uses two UDP *sockets* associated with the IPv4 and IPv6 groups, configured for the port number. It uses a private IPv6 UDP socket (with a single port) to send requests and receive responses. To exchange files, a TCP socket receives connections (with a single port), and an arbitrary number of additional data TCP sockets are used to request and receive the file contents.

The *fileexchange application* starts by waiting for the user to configure the IPv6 and IPv4 multicast channels (IPv6/IPv4 addresses + port number) where they want to listen. After the user presses the 'Active' button, the application boots up with all four sockets and is ready to send or receive UDP messages or TCP connections. At the same time, the application allows you to modify the list of shared files.

When the user chooses to search for a file, the application starts by sending the request to the groups and waits for a response until a maximum waiting time. If it has received a response, it launches a thread that creates a temporary TCP socket to receive the file. Ultimately, it must create a file in an output directory defined in the graphical interface. Upon receipt of a new TCP connection, a thread must also be created to send the requested file. The main window should list all active threads, indicating the file names and the percentage of the file that has already been received/downloaded.

An incomplete implementation is provided that includes all graphical user interface features (including the shared file list management) and some of the socket management and messaging features. **To reduce the workload, work has been reduced to the processing of responses to file requests, thread management, and TCP communication components.**

2.1. File search

Once active, the application can send a file search message to the groups. The name to search should not include the path (i.e., directory names) and should be terminated with the character '\0'. This message also includes a lookup sequence number.

The request message sent to the group consists of an octet with the message type (QUERY), followed by the unique sequence number and the filenames' length and string.

```
File search message: { sequence of }
 unsigned char type;          // message type  -  QUERY= 20
 uint32_t seq;                // request sequence number
 short int len;               // length of 'fname' string = strlen(fname)+1
 char[] fname;                // 'fname' string
```

1	4	2	len
QUERY	seq	len	frame

Upon receiving the request message, the *fileexchange* application should compare the file name with the shared file names (excluding the path) and return a response message (HIT) to the sender, where it indicates the characteristics of the found file (the length and *hash* value) and the TCP port number where the sender should connect to download the file. It also includes the data sent by the sender to allow the association with the request (when several concurrent requests were made).

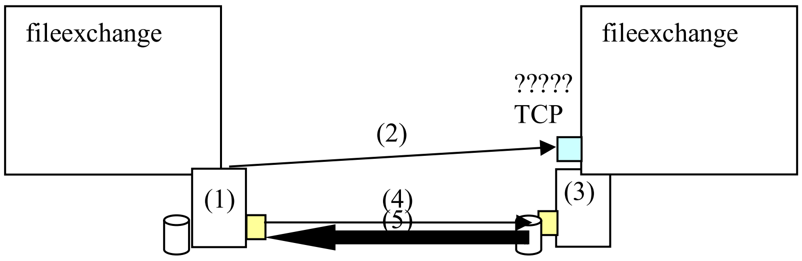
The response message sent to the request sender consists of an octet with the type of message (HIT) followed by the sequence number sent by the sender, the length and name of the requested file, the length of the file, the hash value of the file contents, and the port of the TCP socket where it receives connections.

```
File hit message: { sequence of }
 unsigned char type;          // message type  -  HIT= 10
 uint32_t seq;                // request sequence number
 short int len;               // length of 'fname' string = strlen(fname)+1
 char[] fname;                // 'fname' string
 unsigned long long flen;     // File length
 uint32_t fhash;              // File content's hash value
 unsigned short sTCPport;     // server's TCP port number
 struct in6_addr srvIP;       // server's IPv6 address
```

1	4	2	len	8	4	4	16
HIT	seq	len	frame	flen	fhash	TCPport	srvIP

2.3. File transfer

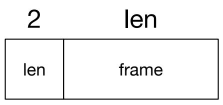
After receiving the first response (i.e. HIT message), the application should download the file, ignoring the following responses. Reading and writing of file blocks should be performed in a thread created on the sender and receiver of the file. After selecting a server, the file receiver (who sent the QUERY) must create a file download thread (1), which creates a temporary TCP socket to receive the file. After establishing the connection (2), the connection accept callback is triggered at the sender (which sent the HIT). This callback should also launch a thread (3), which sends the desired file.

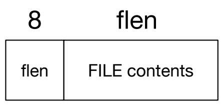


In the communication channel between the receiver and the sender, the communication must start with the sending of a file request (4). The receiver tells the sender what the file name is. The sender must first send (5) the length of the file (or zero, if it does not exist) and then must send the contents of the file requested by the receiver:

```
File request: { sequence of }
 short int len_filename; // Filename length(max. 256 bytes)
 char[] filename;        // Filename string ended by '\0'

File reply: { sequence of }
                       // File length (0 if non-existing)
 unsigned long long file_length;
 char[] file_contents;  // File contents
```





3. Application development

To facilitate the application's development, a fully functional test program (*demo_fileexchange* + *demo_fileexchange.glade*), the Glade-3 *fileexchange.glade* file with the definition of the test program's graphical interface, and a set of files described below are provided.

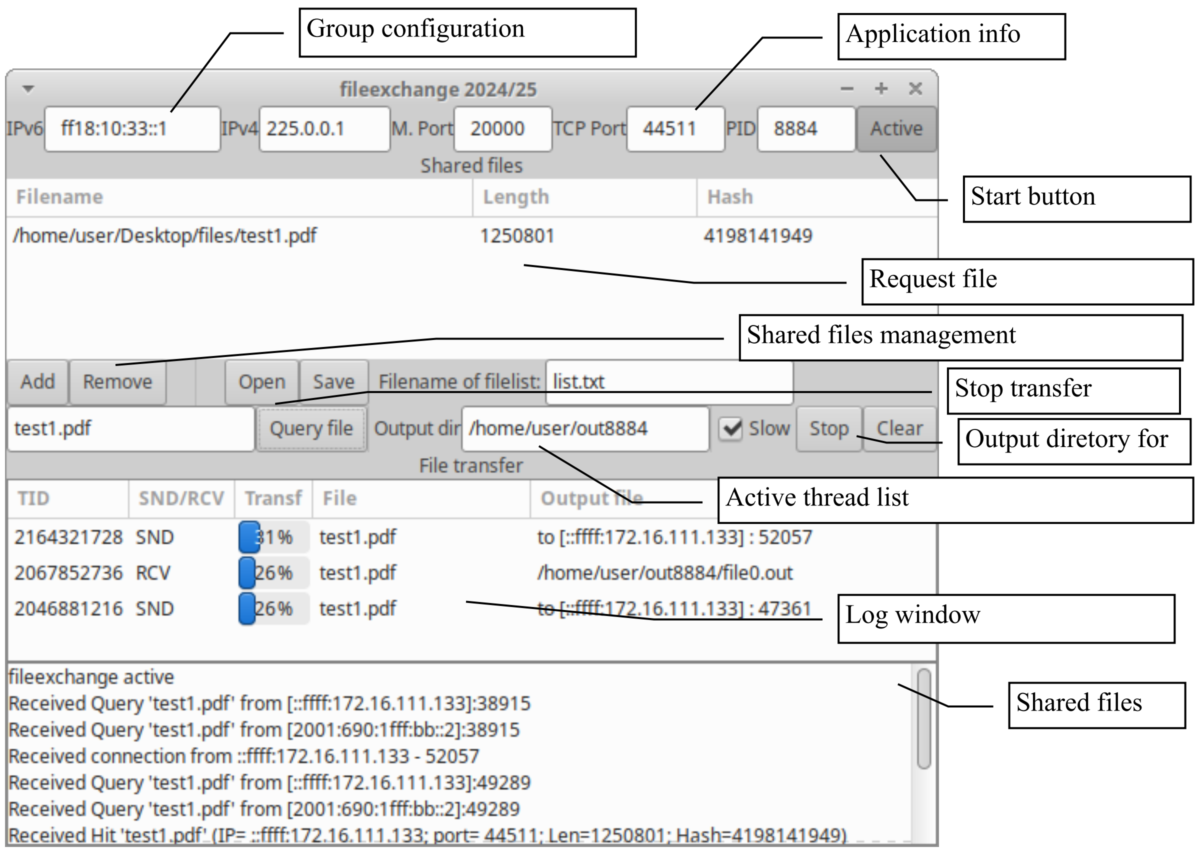
The interface contains a first line where the IPv6 and IPv4 multicast addresses and port numbers are configured and where the application writes the TCP port number, where it receives connections and the process number. The "*Active*" button controls the server startup.

The second and third row contains a table (of type *GtkTreeView*) with the list of shared files (represents the file name, length, and hash value of the file's contents). The next line contains the file list commands: they allow you to add or remove a file, and save or read a file with a list of files, with the name in the box on the right. The next line controls the transfer of files: it allows you to request files, stop a transfer, define whether to optimize the transfer, and shows the directory where the files are written. The directory name results from concatenating "*out*" with the *pid* of the process. This line also includes a button to clear the text window from the last line.

The fifth line contains the list of active threads at a given time, with the number of *tid* (thread id), whether it is sender (SND) or receiver (RCV), the percentage of bytes transferred, the name of the requested file, and the file being created.

The sixth (last) line contains a box for writing messages.

Each group can make all the modifications they want to the base program. However, it is recommended that you invest the time in correctly implementing the proposed application.



IPv6 ff18:10:33::1 IPv4 225.0.0.1 M. Port 20000 TCP Port 44511 PID 8884

Filename	Length	Hash
/home/user/Desktop/files/test1.pdf	1250801	4198141949

Filename of filelist: list.txt

test1.pdf

Output dir /home/user/out8884

TID	SND/RCV	Transf	File	Output file
2164321728	SND	31%	test1.pdf	to [::ffff:172.16.111.133] : 52057
2067852736	RCV	26%	test1.pdf	/home/user/out8884/file0.out
2046881216	SND	26%	test1.pdf	to [::ffff:172.16.111.133] : 47361

```
fileexchange active
Received Query 'test1.pdf' from [::ffff:172.16.111.133]:38915
Received Query 'test1.pdf' from [2001:690:1fff:bb::2]:38915
Received connection from ::ffff:172.16.111.133 - 52057
Received Query 'test1.pdf' from [::ffff:172.16.111.133]:49289
Received Query 'test1.pdf' from [2001:690:1fff:bb::2]:49289
Received Hit 'test1.pdf' (IP= ::ffff:172.16.111.133; port= 44511; Len=1250801; Hash=4198141949)
```

The C code provided is organized into seven modules:

- *sock.{c,h}* – Function library for handling *sockets* and IP addresses (including getting the local address);
- *file.{c,h}* – Function library for handling files;
- *gui.h* and *gui_g3.c* – Library of functions to handle the graphical elements of the interface, including the file and thread lists;
- *thread.{c,h}* – Functions to handle file transfer threads. **This module is incomplete** – Most of the functions that manage the file transfer still need to be completed;
- *callbacks_socket.{c,h}* – Socket start and stop functions, socket interaction callbacks, and UDP message encoding. **This module is incomplete** – the function that initializes the IPv4 UDP socket remains to be completed;
- *callbacks.{c,h}* – Application start and stop functions, timers, and program logic control. **This module is incomplete** – Part of the functions remain to be completed;
- *main.c* – App main function. Creates a directory to store the files received.
- *Makefile* – Make control script. Can be used to define compilation symbols (e.g. DEBUG) used to control the program.

The supplied program includes all complete files except for the files *thread.{c,h}*, *callbacks_socket.{c,h}* and *callbacks.{c,h}*, which the students must complete. In multicast communication, the initialization of the IPv4 sockets, the configuration of the request timeout clock, and the processing of the HIT message still need to be completed. In file communication, communication via the TCP socket is yet to be done.

The initial part of the work corresponds to reading and understanding the given code, making UDP communication feasible in a short time. STUDENTS ARE RECOMMENDED TO LOOK AT THE CODE PROVIDED IN DETAIL, so that they can take full advantage of it. You should read and understand the code that is provided with the assignment before the first class.

The work must be developed in several distinct tasks:

1. finish programming the function that handles the button sends the QUERY packet to set up a timer (to give up after QUERY_TIMEOUT ms) and not allow competing requests until the time expires or receives a HIT;
2. finish programming the HIT message receiving function, handling multiple responses to a request, and launching the file-receiving threads;
3. IPv4 multicast socket initialization programming.
4. Programming the file upload thread. You must complete the function with the code to send the file and update the graphical interface;
5. Programming the file download thread. The file must be saved in the directory created for this purpose, updating the graphical interface during the transference. You must run the protocol exchange before receiving the file;
6. If you have time, **optimize the throughput in file transfer** by modifying the size of the send and receive data buffers at the TCP level, and the parameters of the TCP sockets[1]. In the end, there will be a **prize of two points out of twenty in the grade of the work (maximum of 20) for the TWO works that have the shortest transmission time of a test file**, measured during the final discussion.

Planned goals for the four weeks of work:

1. Read the tutorial, setup the virtual machine, and start task (2);
2. Start working on task (4) during week 2;
3. Start working on task (5) during week 3;
4. Complete the work during week 4.

   The work has been optimized so that it can be carried out in parallel in groups of two elements; Tasks (1)-(2) and (3); and (4) and (5) can be performed concurrently.
   Don't forget that on November 23 it will be necessary to deliver the work. Don't leave for the last week what you can do over the first few weeks, because **you WON'T BE ABLE TO DO ALL THE WORK IN THE LAST WEEK.**

Student Posture

Each group should consider the following:

- Do not waste time with the aesthetics of input and output data;
- Program following the general principles of good coding (using indentation for comments, using variables with names conform to its functions ...) and;
- Proceed so that the work is equally distributed to the two members of the group.

[1] Note that **the test program intentionally slows down when it has the slow option selected** because it includes a '*usleep*' in the file send/receive cycles, which makes it "sleep" for 50 ms between file block sends, in addition to not configuring the *buffers* and parameters of the TCP sockets.