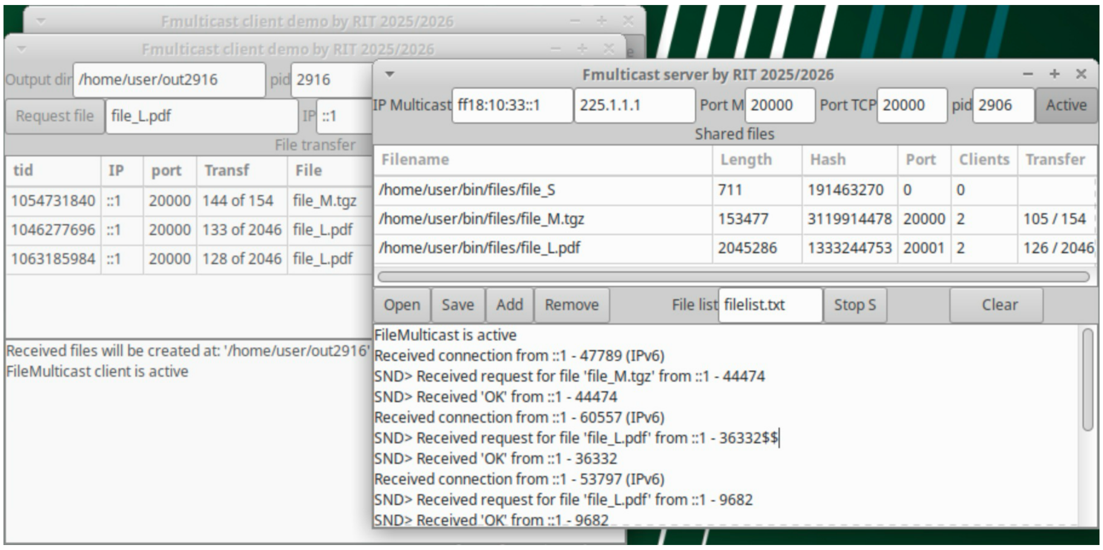
NOVA SCHOOL OF SCIENCE & TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING


Fmulticast client demo by RIT 2025/2026
Fmulticast client demo by RIT 2025/2026
Fmulticast server by RIT 2025/2026
Request file
File transfer
Shared files
Open
Save
Add
Remove
Stop S
Clear
Active

Output dir /home/user/out2916 pid 2916
Request file file_L.pdf IP ::1

tid	IP	port	Transf	File
1054731840	::1	20000	144 of 154	file_M.tgz
1046277696	::1	20000	133 of 2046	file_L.pdf
1063185984	::1	20000	128 of 2046	file_L.pdf

```
Received files will be created at: '/home/user/out2916'
FileMulticast client is active
```

IP Multicast ff18:10:33::1 225.1.1.1 Port M 20000 Port TCP 20000 pid 2906

Filename	Length	Hash	Port	Clients	Transfer
/home/user/bin/files/file_S	711	191463270	0	0	
/home/user/bin/files/file_M.tgz	153477	3119914478	20000	2	105 / 154
/home/user/bin/files/file_L.pdf	2045286	1333244753	20001	2	126 / 2046

File list filelist.txt

```
FileMulticast is active
Received connection from ::1 - 47789 (IPv6)
SND> Received request for file 'file_M.tgz' from ::1 - 44474
SND> Received 'OK' from ::1 - 44474
Received connection from ::1 - 60557 (IPv6)
SND> Received request for file 'file_L.pdf' from ::1 - 36332$$
SND> Received 'OK' from ::1 - 36332
Received connection from ::1 - 53797 (IPv6)
SND> Received request for file 'file_L.pdf' from ::1 - 9682
SND> Received 'OK' from ::1 - 9682
```

Integrated Telecommunication Networks (RIT)

2025/2026

2nd Laboratory Work:
Multicast application for file diffusion in IPv4 and IPv6

Classes 7 to 10

Mestrado integrado / Mestrado em
Engenharia Eletrotécnica e de Computadores / MERSIM

https://tele1.deec.fct.unl.pt/rit **Luis Bernardo**

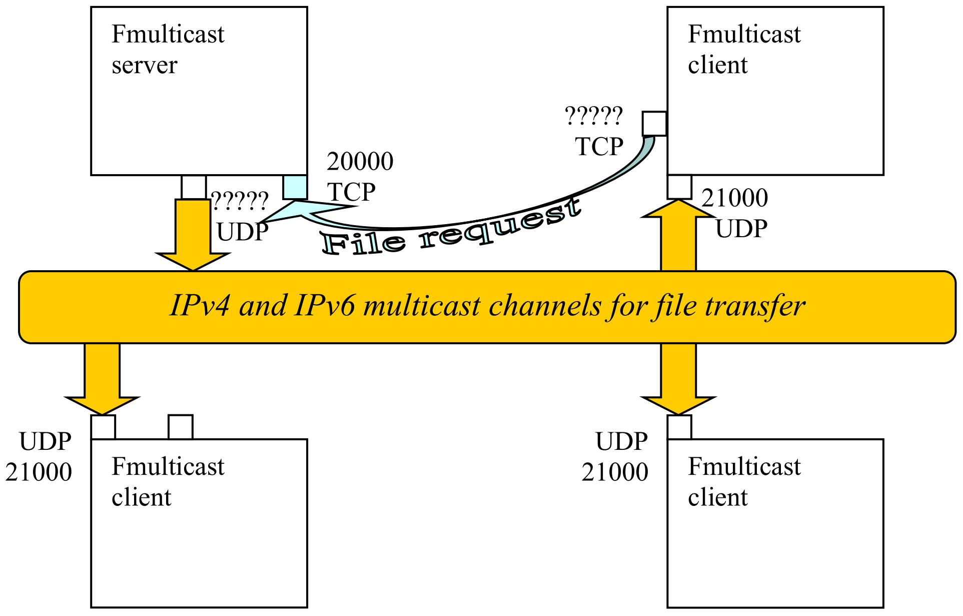
1. Objectives

Familiarisation with programming using IPv4 and IPv6 addresses, the Gnome/Gtk+3 graphical environment, and the mechanisms of thread management and inter-thread synchronisation in the Linux operating system.
The work consists of the development of an application for reliable file diffusion in a dual-stack IPv4/IPv6 network. The application receives requests in text messages over a TCP connection and sends the files in parallel to all clients using IPv6 and IPv4 Multicast addresses. The application sends requests over the TCP connection whenever the user places an order. The receivers may receive multiple files in parallel and should never block due to a transmission failure.

The work involves implementing a program: fmulticast_client.

2. Specifications

The *Filemulticast* application is implemented through two programs: *fmulticast_client* and *fmulticast_server*. The *fmulticast_server* requires two configuration parameters: the Multicast IP address of the groups (by default, the addresses "ff18:10:33::1" and "225.1.1.1" will be used) and the TCP port number where the requests are received (by default, port 20000 will be used).



The *fmulticast_client* application creates a temporary TCP socket and establishes a connection to a *fmulticast_server* (to port 20000 in the figure, with four distinct machines), sending the request to it. Thus, each server (*fmulticast_server*) will use a public TCP socket, configured to accept connections. Each client (*fmulticast_client*) will use a variable number of private TCP sockets (one per parallel connection request). A reliable multicast protocol is run for file transfer over UDP sockets. Each node that is receiving the file creates a multicast UDP socket configured for the port associated with the file (21000 in the case of the figure). The server uses a UDP socket to send the file fragments to the network and receive the selective retransmission requests.

The *fmulticast_server* application starts by waiting for the user to configure the channel (IPv6 and IPv4 multicast addresses) and the listening TCP port number. After the user presses a button, the application starts the TCP socket for receiving requests and is ready to accept TCP connections. In parallel, the application allows you to manage the list of shared files.

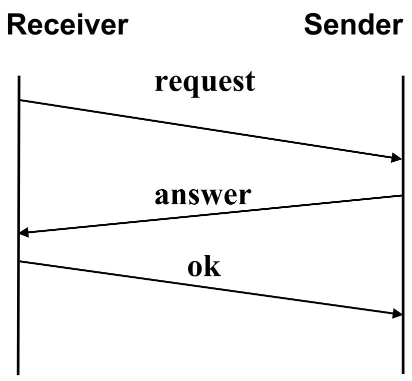
The application *fmulticast_client* allows you to make multiple file requests to servers. When the user requests a file, it specifies the server's address and port. The application begins by creating a temporary TCP socket (or reusing a previously created one) and sending the request to the server, then waiting for the response. The response includes the multicast IP address and the UDP port where the application should receive the file. After creating the UDP multicast socket (shared to allow multiple nodes on the same machine), it sends an "OK" message to the server, which starts periodically sending the various blocks of the file through a private UDP socket to all clients. The client maintains a bit mask where it stores the information about correctly received packets. The client sends this mask regularly to the server in SRR (Selective Repeat Request) packets. The server also maintains a mask with all packets that recipients have not yet received, controlling which packets to send. The protocol is described in section 2.2. To allow full parallelism in receiving and sending multiple files, students are expected to program the various tasks in different threads. The main window should list all active threads, received files, and downloaded packets.

In order to enable the development of the work in the four weeks provided, an interface specification file "*fmulticast_client.glade*" is provided, which includes the main program window and a set of additional functions to access the graphical interface and interact with sockets. These files already include application startup and all the transfer list manipulation functions. The work consists ONLY OF THE FILE RECEIVING PROGRAMMING. The full server application *fmulticast_server* is provided with the laboratory assignment.

2.1. File request

From the moment it is active, the client can perform a file request, creating a TCP socket and establishing a connection to the server node. This operation is blocking and can stop the entire application. To ensure that the graphical interface and the remaining tasks are not blocked, **it must be performed in a separate thread**.

The file request is performed by exchanging binary messages over a TCP connection that is established between the client (receiver) and the server (sender).



```
request message
char *name;  { name ended by '\0'}

answer message /* sequence of */
 short int CID; // Client ID (>= 0)
 short int SID; // Session ID
 unsigned long long f_len; // File length
 int block_size; //
 int n_blocks; // Number of blocks
 unsigned int f_hash; // File's hash
   // (m)ulticast address and port
union { // Multicast address
     struct in6_addr maddr6; // IPv6
     struct in_addr maddr4;  // IPv4
 } u;
unsigned short int mport; // multicast port
OR // If the file does not exist
 short int CID= -1;
 short int error_msg_len;
 char *error_msg; {message ended by '\0'}

ok mesage
 const char *ok= "OK"; {including '\0'}
```

The request message sent to the sender (server) contains only the name of the desired file. The filename should not include the path (directory names) and should be terminated with the character '\ 0'.

The sender, when given the valid name: it returns a CID equal to -1 if it does not have the requested file and an error message (`error_msg`); if it has the file, it returns a message with: unique client identifier (`CID`), session identifier (`SID`, present in all UDP packets), file length (`f_len`), maximum size of transmitted data blocks (`block_size`), number of transmitted blocks (`n_blocks`), hash value of the file (`f_hash`, to allow testing if the file is well received), and the multicast address and port (`mport`) to where the file will be transmitted. If the request is made from a native IPv6 address, an IPv6 multicast address (`u.maddr6`) is returned. If it is made from an IPv4 address (mapped to an IPv6 address of type '::ffff:IPv4'), an IPv4 multicast address (`u.maddr4`) is returned.

After receiving the reply message, the receiver (client) must create a multicast socket, associate it with the port and the multicast address indicated by the sender, and send an "OK" message (literally). After receiving the *"OK"*, the server starts sending the file. If an existing transmission already exists, it modifies the transmission mask to retransmit the entire file.

The TCP connection can be maintained during transmission and used by the receiver to indicate that it will not receive the file (by sending the string "END" with the character '\ 0' at the end). Alternatively, both the receiver and the sender can terminate this connection, without affecting the file transmission that is occurring, reducing problems resulting from too many open file descriptors in the process.

2.2. File transfer

The file transfer starts after the server receives "OK" from the first client through the TCP connection. The server launches a task that periodically (with a period of `sender_timer_period` milliseconds) sends a new DATA package, containing one more segment of the file. |Clients must send SRR packets to confirm data reception and request retransmissions, ensuring that the sender receives an SRR for at least every `SRR_timeout` milliseconds. If the SRR is not received, the sender excludes the clients from the list of valid receivers. When this list is empty, or when all pending segments are confirmed, the server stops the transmission task.

Clients wait on a UDP multicast socket for receiving DATA packets, **sending SRR every 2 DATA packets received, or after `receiver_SRR_timeout` milliseconds without receiving any DATA packets**. The client can transmit up to three consecutive SRRs without receiving DATA packets, and then give up file reception after the fourth DATA packet wait period.

```
DATA packet: { sequence of }
 unsigned char type;     // message type -  DATA=1
 short int SID;          // Session ID
 int seq;                // Block sequence number
 int len;                // Block data length
 char *block;            // File contents
```

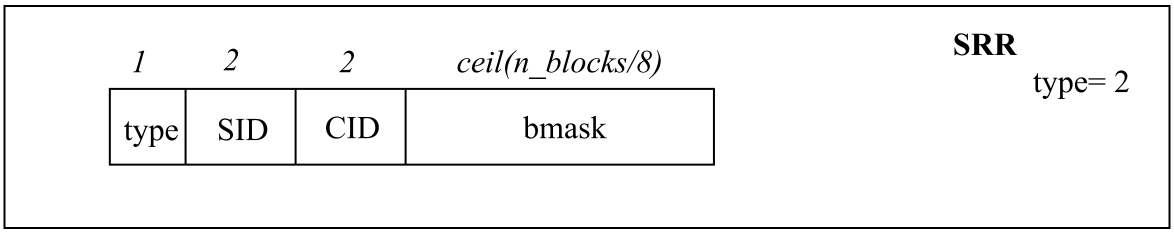
1	*2*	*4*	*4*	*len*
type	SID	seq	len	File block

DATA
type= 1

DATA packets must have a file block with `sender_block_size` bytes, except the last block of the file, which may have fewer. In addition, they contain the session identifier and the sequence number, which indicates the position of the block in the file.

SRR packets include SID and CID to identify the flow and receiver, and an array of bytes that contains a bit mask with a number of bits equal to the total number of packets that make up the file. Each bit represents a received block. Students are given the *bitmask.c* and *bitmask.h* files, which define the `BITMASK` type and functions to manipulate it, in order to facilitate the programming of bit masking.

```
SRR packet: { sequence of }
 unsigned char type;    // message type  -  SRR=2
 short int SID;         // Session ID
 short int CID;         // Client ID
 char *bmask;           // Bit mask
```

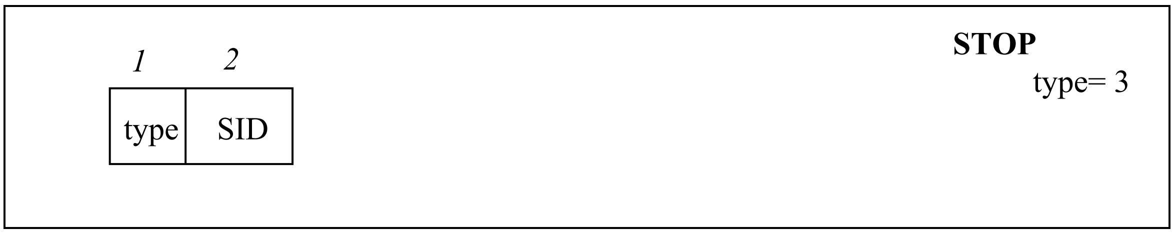


The server can stop the transmission of a file at any time by sending a STOP packet. In parallel, a client may abandon the reception of a file and notify the server using two complementary mechanisms:

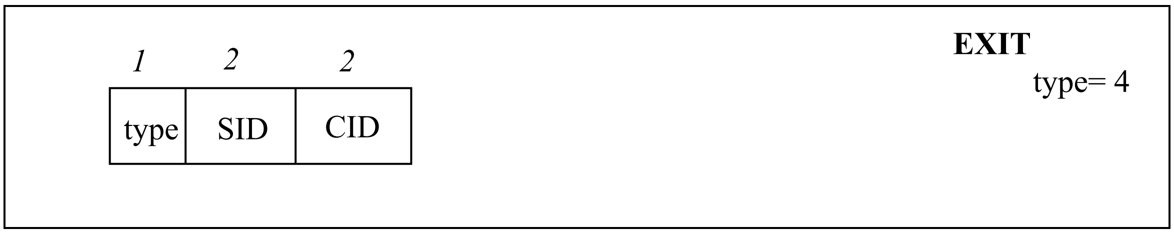
- Sending an EXIT packet through the UDP socket to the server if it has already received DATA packets;
- Sending an "END" message over the TCP connection.

In either case, the server deletes the client from the list of active receivers, excluding it from the transmission stop count.

```
STOP packet: { sequence of }
 unsigned char type;    // message type -  STOP=3
 short int SID;         // Session ID
```



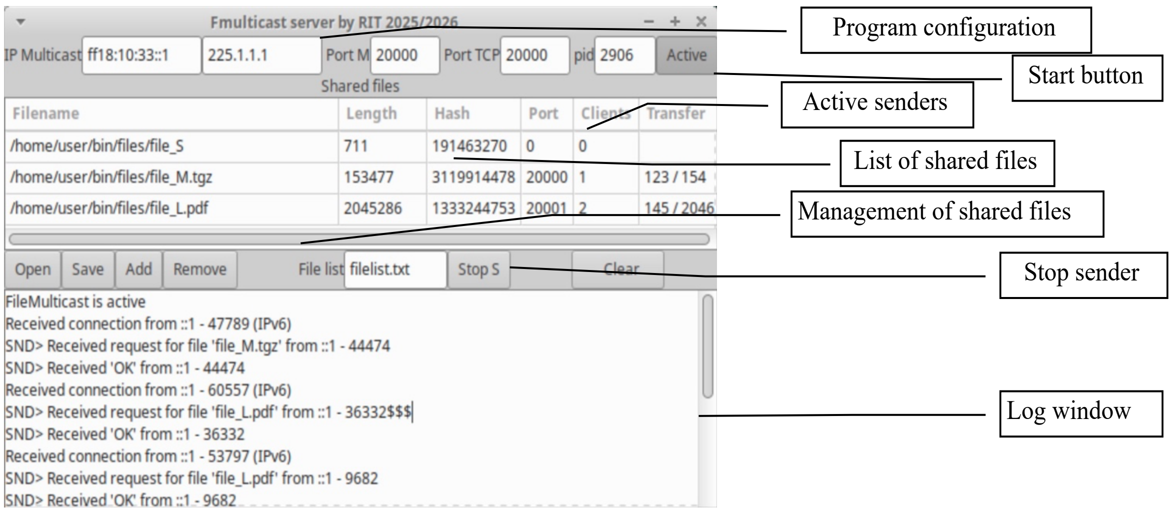
```
EXIT packet: { sequence of }
 unsigned char type;    // message type  -  EXIT=4
 short int SID;         // Session ID
 short int CID;         // Client ID
```



3. Program development

To facilitate the development and testing of the application the definition of the graphical interface (*fmusticast_client.glade*), the program start code (*main.c*), a set of functions to manage the GUI and threads table (*gui_g3.c* and *gui.h*), which handles graphical events (*callback.c* and *callback.h*), to manipulate files (*file.c* and *file.h*), to manage sockets (*sock.c* and *sock.h*), to manage bitmaps (*bitmap.c* and *bitmap.h*) are provided complete. You only have to program the module that handles file receiving (*receiver_th.c* and *receiver_th.h*) and some functions related to IPv4 multicast communication. A fully functional demo program (*demo_fmulticast_client* + *demo_fmulticast_client.glade*) and server program (*fmulticast_server* + *fmulticast_server.glade*) are also provided, with the graphical interface shown below.

3.1. *fmulticast_server* (complete)



IP Multicast ff18:10:33::1 225.1.1.1 Port M 20000 Port TCP 20000 pid 2906 Active

Filename	Length	Hash	Port	Clients	Transfer
/home/user/bin/files/file_S	711	191463270	0	0	
/home/user/bin/files/file_M.tgz	153477	3119914478	20000	1	123 / 154
/home/user/bin/files/file_L.pdf	2045286	1333244753	20001	2	145 / 2046

File list filelist.txt

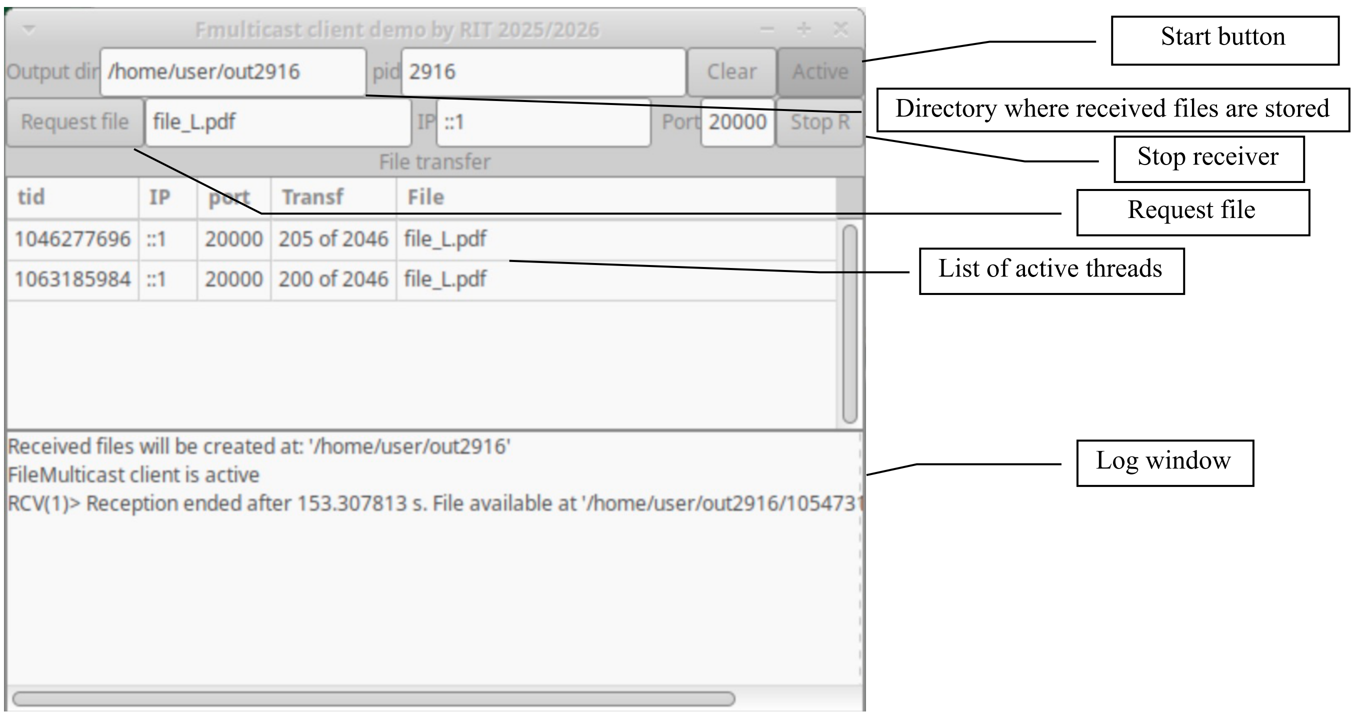
```
FileMulticast is active
Received connection from ::1 - 47789 (IPv6)
SND> Received request for file 'file_M.tgz' from ::1 - 44474
SND> Received 'OK' from ::1 - 44474
Received connection from ::1 - 60557 (IPv6)
SND> Received request for file 'file_L.pdf' from ::1 - 36332$$$
SND> Received 'OK' from ::1 - 36332
Received connection from ::1 - 53797 (IPv6)
SND> Received request for file 'file_L.pdf' from ::1 - 9682
SND> Received 'OK' from ::1 - 9682
```

The graphical interface contains an initial line where you configure the IPv6 and IPv4 multicast addresses, the TCP port number, and the initial UDP port number, which will be used by the sender to choose new multicast channels (only the port changes; the IP address remains the same). The "Active" button controls the application's startup. The application auto-configures the ports when multiple instances are running on the same machine.

On the second and third lines the interface contains a table (of type *GtkTreeView*) with the list of shared files (represents the file name, length, and hash value of the file contents in the first three columns) and the active senders (represents for each active file sender, the port used, the number of connected recipients, and the total number of successfully sent blocks). The following line contains the file list commands: (*Add*) to add or (*Remove*) to remove a file, and (*Save*) to save or (*Open*) to read a file from a file list, with the name represented in the box to the right. It also includes the commands to stop a sender (after selecting a line). These two lines are completely programmed.

The fourth, and last line, contains a log box for writing messages.

3.2. *demo_fmulticast_client* (complete)



Output dir: /home/user/out2916 pid: 2916
Request file: file_L.pdf IP: ::1 Port: 20000

tid	IP	port	Transf	File
1046277696	::1	20000	205 of 2046	file_L.pdf
1063185984	::1	20000	200 of 2046	file_L.pdf

```
Received files will be created at: '/home/user/out2916'
FileMulticast client is active
RCV(1)> Reception ended after 153.307813 s. File available at '/home/user/out2916/105473
```

The client application does not require any initial configuration. The graphical interface contains a first line that displays information about the directory where the received files are stored, the local process identifier (pid), and includes two buttons: a button to clear the log text window of the last line and the "Active" button, which controls the application startup.

The second line controls file reception, allowing you to request files by specifying the file name, IPv6 or IPv4 address, and port number of the server program. It also allows you to stop a receiver (by selecting a receiver in the table).

The third line contains the list of active receiver threads at a given time, including the tid (*thread id*), IPv6 address and port number of the sender, the number of received / total blocks of the file, and the file name being received.

The fourth, and last line, contains a log box for writing messages.

3.3. *fmulticast_client* (to implement)

The program provided is composed of seven modules:

- *bitmask.c* and *bitmask.h* – Bit mask, **complete**;
- *gui_g3.c* and *gui.h* – Management of the graphical interface, **complete**;
- *callbacks.c* and *callbacks.h* – Request handling of the graphical interface, **complete**;
- *file.c* and *file.h* – File manipulation, **complete**;
- *sock.c* and *sock.h* – Functions to handle sockets, **complete**;
- *main.c* – Application startup, **complete**;
- *receiver_th.c* and *receiver_th.h* – **Incomplete** receiver code, **to be completed**;
- *fmulticast_client.glade* – Definition of the graphical interface.

All the code to be developed will be placed, or invoked from the "`receiver_thread_function`" function, which contains the code run on the file-receiving thread, with the receiving algorithm. The list of functions to be modified is as follows:

```
void* receiver_thread_function(void *ptr);
ReceiverTh *new_receiverTh_desc(const gchar *name, const gchar *ip, int port);
gboolean send_SRR(ReceiverTh *t, short int sid, short int cid);
gboolean send_EXIT(ReceiverTh *t, short int sid, short int cid);
```

The data associated with each *thread* is stored in a ReceiverTh structure, which includes data on connection status, TCP connections (*addr*, *is_ipv4*, *st*), and UDP reception (*sm*, *bmask*, *saddr_def*, *u*, *cid*, *sid*). You can add more fields to the structure.

The sender (server) address is memorized when the first UDP packet is received in the multicast socket (*sm*). After receiving the first packet, the *saddr_def* variable is set to TRUE and the sender's address is stored in *u.saddr6* or *u.saddr4*, respectively for IPv6 and IPv4.

```
typedef struct ReceiverTh {
      gboolean active;              // Receiver state
      struct ReceiverTh *self;      // equal to the pointer if valid
      struct sockaddr_in6 addr;     // TCP Destination address
      gboolean is_ipv4;             // TRUE if IPv4, FALSE if IPv6
      pthread_t tid;                // Thread ID
      char fname[81];               // Requested filename
      int st;                       // TCP socket descriptor
      int sm;                       // Multicast UDP socket descriptor
      FILE *sf;                     // File descriptor
      char name_str[80];
      char name_f[256];             // name of created file
      BITMASK bmask;                // BITMASK with received blocks
      gboolean saddr_def;           // If sender's IP address is known
      union {
              struct sockaddr_in6 saddr6;   // UDP sender's IPv6 address
              struct sockaddr_in saddr4;    // UDP sender's IPv4 address
      } u;
      short int cid;                // Client ID
      short int sid;                // Session ID

      // Additional fields are needed to implement the receiver logic
      // ...
} ReceiverTh;
```

3.4 Goals

A sequence for the implementation of the program could be:

1. Read and understand the code that is provided with the work assignment (BEFORE THE FIRST CLASS);
2. Start programming the *thread* supporting only IPv6 addresses. In the receiver_thread_function function, you should start with creating the TCP socket and establishing the TCP connection to the server;
3. Program the TCP communication in IPv6: create the TCP socket, establish the connection and perform the exchange of information with the server until sending the "OK";
4. Program the data reception cycle of the UDP socket in parallel with data from the TCP socket, using the select function, executing the protocol described in this document;
5. Complete the *send_SRR*, *send_EXIT* and *receiver_thread_function* functions to support IPv4;
6. Complete the code developed in the previous points by making it robust to response failures or unexpected errors.

The use of *threads* makes the task of *debugging* the code and managing the parallelism in accessing the GUI more complicated.

In order to get to the end of the four weeks of work with everything ready, it is necessary to use ALL PRACTICAL CLASSES, which must: begin phase 3 during class 2; start phase 4 during class 3; conclude phase 5 and make last-minute adjustments on the last lesson. Do not leave for the last week what you can do over the first four weeks, because **YOU WILL NOT 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 in accordance with the general principles of good coding (using indentation, comments, using variables with names conform to their functions, etc.), and;
- Proceed so that the work is evenly distributed between the two members of the group.