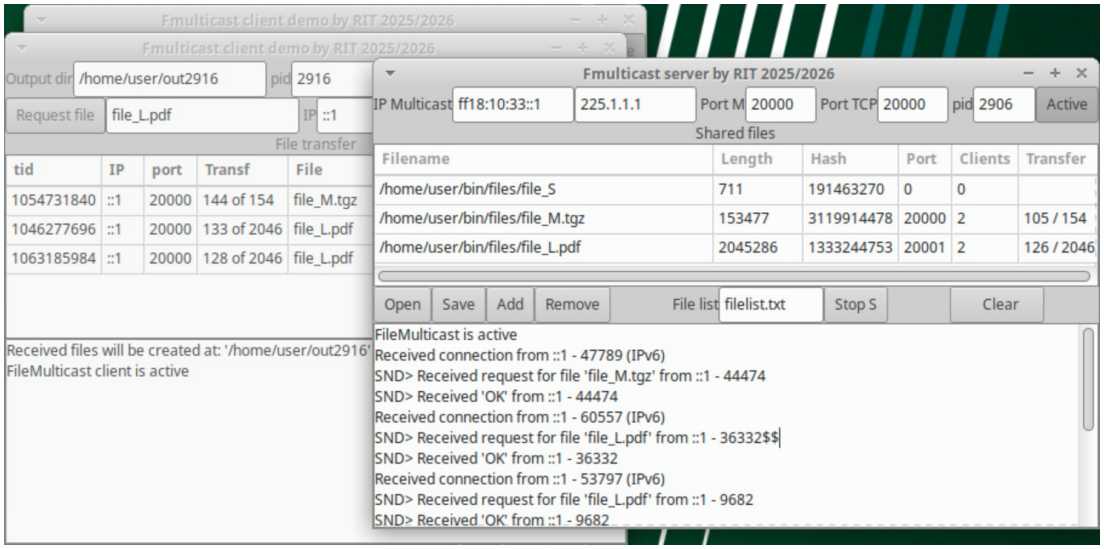
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING



Redes Integradas de Telecomunicações

2025/2026

Trabalho 2:
Aplicação *multicast* para difusão de ficheiros em IPv4 e IPv6

Aulas 7 a 10

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

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

Luis Bernardo

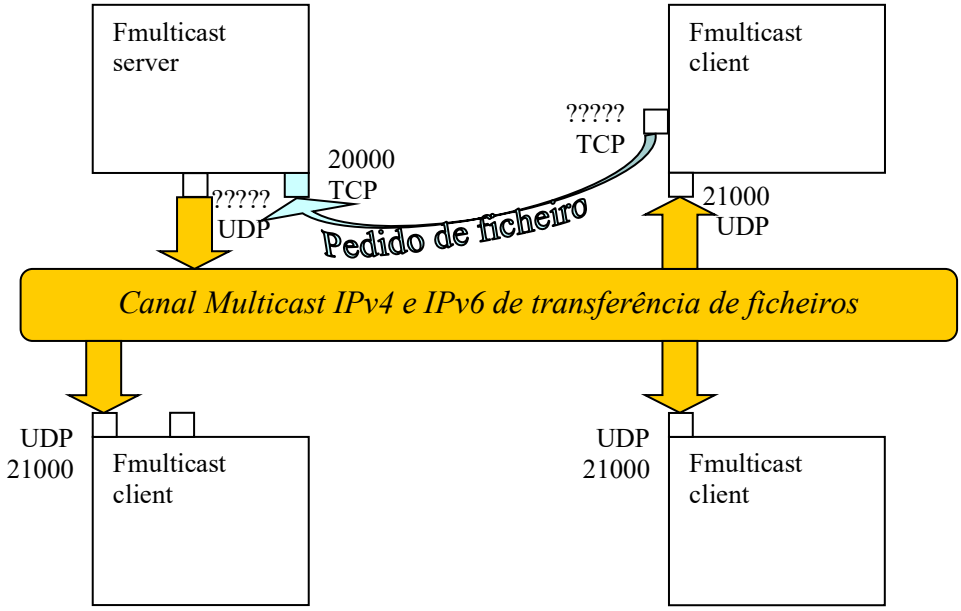
1. Objetivos

Familiarização com a programação usando endereços IPv4 e IPv6, o ambiente gráfico Gnome/Gtk+ 3, e os mecanismos de gestão de tarefas (*threads* POSIX) e de sincronização entre tarefas no sistema operativo Linux. O trabalho consiste no desenvolvimento de uma aplicação para difusão de ficheiros fiável numa rede de camada dupla com IPv4 e IPv6. A aplicação recebe pedidos em mensagens de texto através de uma ligação TCP, e envia os ficheiros em paralelo para todos os clientes utilizando endereços Multicast IPv6 e IPv4. A aplicação envia os pedidos através da ligação TCP sempre que o utilizador realiza um pedido. Os clientes podem estar a receber em PARALELO vários ficheiros e não devem NUNCA bloquear devido à falha de uma transmissão.

O trabalho consiste no desenvolvimento de um executável: ***fmulticast_client***.

2. Especificações

A aplicação *Filemulticast* é realizada através de dois programas: o *fmulticast_client* e o *fmulticast_server*. O *fmulticast_server* necessita de dois parâmetros de configuração: o endereço IP Multicast dos grupos (por omissão vão-se usar os endereços "ff18:10:33::1" e "225.1.1.1") e o número de porto TCP onde se recebem os pedidos (por omissão vai-se usar o porto 20000).



A aplicação *fmulticast_client* cria um socket TCP temporário e estabelece a ligação para um *fmulticast_server* (para o porto 20000 na figura, com quatro máquinas distintas), enviando-lhe o pedido. Assim, cada servidor (*fmulticast_server*) vai usar um socket TCP público, configurado para aceitar ligações. Cada cliente (*fmulticast_client*) vai usar um número variável de sockets TCP privados (um por cada pedido de ligação em paralelo). A transferência de ficheiros corre um protocolo de difusão fiável sobre sockets UDP. Cada nó que esteja a receber o ficheiro tem de criar um socket UDP multicast configurado para o porto associado ao ficheiro (21000 no caso da figura). O servidor usa um socket UDP para enviar os fragmentos do ficheiro para a rede e receber os pedidos de retransmissão seletiva.

A aplicação *fmulticast_server* começa por aguardar que o utilizador configure o canal (endereços multicast IPv6 e IPv4) e o número de porto TCP para onde vai transmitir o conteúdo do ficheiro. Após o utilizador premir um botão, a aplicação arranca o socket TCP de receção de pedidos e fica preparada para receber ligações TCP de clientes. Paralelamente, a aplicação permite gerir a lista de ficheiros partilhados.

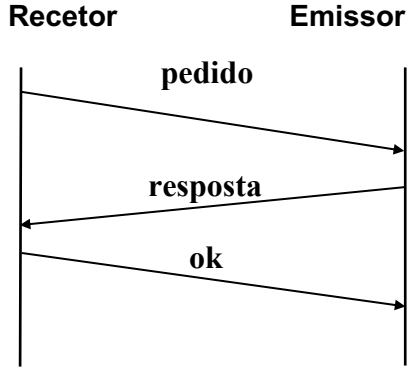
A aplicação *fmulticast_client* permite efetuar vários pedidos de ficheiro a servidores. Quando o utilizador pede um ficheiro, especifica o endereço e porto do servidor. A aplicação começa por criar um socket TCP temporário (ou reutilizar um criado anteriormente) e enviar para o servidor o pedido, ficando a aguardar pela resposta. A resposta inclui o endereço IP multicast e o porto UDP onde a aplicação deve receber o ficheiro. Após criar o socket multicast UDP (partilhado para permitir ter vários nós na mesma máquina), envia uma mensagem "OK" para o servidor, que inicia o envio periódico dos vários blocos do ficheiro, através de um socket UDP privado, para todos os clientes. O cliente mantém uma máscara de bits onde guarda a informação sobre os pacotes corretamente recebidos. O cliente envia esta máscara regularmente ao servidor em pacotes SRR (*Selective Repeat Request*). O servidor também mantém uma máscara de bits com a informação sobre todos os pacotes que os recetores ainda não receberam, controlando quais os pacotes que deve enviar. O protocolo é descrito na secção 2.2. Para permitir o paralelismo total na receção e envio de vários ficheiros, pretende-se que os alunos programem as várias transferências na aplicação *fmulticast_client* em threads distintas concorrentes. A janela principal deve listar todas as threads ativas, os ficheiros recebidos e os pacotes transferidos.

Para possibilitar o desenvolvimento do trabalho nas quatro semanas previstas, são fornecidos um ficheiro de especificação de interface "*fmulticast_client.glade*" com a janela principal do programa e um conjunto de ficheiros de código com algumas funções adicionais de acesso à interface gráfica e de interação com sockets. Estes ficheiros já incluem o arranque da aplicação e todas as funções de manipulação da lista de pedidos. O trabalho consiste apenas NA PROGRAMAÇÃO DA THREAD DE RECEÇÃO DO FICHEIRO. A aplicação servidora completa *fmulticast_server* é fornecida com o enunciado do trabalho.

2.1. Pedido de ficheiros

A partir do momento em que fica ativa, o cliente pode realizar um pedido de ficheiro, criando um socket TCP e estabelecendo uma ligação para um servidor. Esta operação é bloqueante e pode parar toda a aplicação – para garantir que a interface gráfica e as restantes tarefas não ficam bloqueadas, **deve ser realizada numa *thread***.

O pedido de ficheiro é realizado através da troca de mensagens binárias sobre uma ligação TCP que se estabelece entre o recetor e o emissor.



Mensagem pedido

```
char *nome;  { nome terminado com '\0'}
```

Mensagem resposta /* sequência de */

```
 short int CID; // ID de cliente (>= 0)
 short int SID; // ID de sessão
 unsigned long long f_len; // File length
 int block_size; // Tamanho bloco
 int n_blocks; // Número de blocos
 unsigned int f_hash; // Hash do ficheiro
   //Endereço e porto (m)ulticast
 union { // Endereço (m)ulticast
     struct in6_addr maddr6; // IPv6
     struct in_addr maddr4;  // IPv4
 } u;
 unsigned short int mport; // porto multicast
OU // Se o ficheiro não existe
 short int CID= -1;
 short int error_msg_len;
 char *error_msg; {mensagem com '\0' no fim}
```

Mensagem ok

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

A mensagem de pedido enviada para o emissor (servidor) contém apenas o nome do ficheiro pretendido. O nome de ficheiro não deve incluir o caminho (nomes de diretórios) e deve ser terminado com o carater '\0'.

O servidor, ao receber o nome valida se tem, ou não, o ficheiro pedido: retorna um CID igual a `-1` se não tem o ficheiro e uma mensagem de erro (`error_msg`); se tem o ficheiro, retorna uma mensagem com: o identificador único de cliente (`CID`), o identificador de sessão (`SID`, presente em todos os pacotes UDP), o comprimento do ficheiro (`f_len`), a dimensão máxima dos blocos de dados transmitidos (`block_size`), o número de blocos transmitidos (`n_blocks`), o valor de *hash* do ficheiro (`f_hash`, para testar se o ficheiro é bem recebido), e o endereço multicast e o porto (`mport`) para onde o ficheiro vai ser transmitido. Caso o pedido seja feito a partir de um endereço IPv6 nativo, é devolvido um endereço multicast IPv6 (`u.maddr6`). Se for feito a partir de um endereço IPv4 (mapeado num endereço IPv6 do tipo '::ffff:IPv4'), é devolvido um endereço multicast IPv4 (`u.maddr4`).

Após receber a mensagem de resposta, o cliente deve criar um socket multicast, associar-se ao porto e ao endereço multicast indicado pelo servidor, e enviar uma mensagem com a string *"OK"* (literalmente). Após a receção do *"OK"*, o servidor inicia o envio do ficheiro, ou caso já exista uma transmissão em curso, modifica a máscara de transmissão de maneira a voltar a transmitir a totalidade do ficheiro.

A ligação TCP pode ser mantida durante a transmissão, podendo ser usada pelo cliente para indicar que vai desistir da receção do ficheiro (enviando a string "END" com o carater '\0' no fim). Alternativamente, tanto o recetor como o emissor podem terminar esta ligação, sem afetar a transmissão de ficheiros que está a ocorrer, reduzindo problemas resultantes de existirem demasiados descritores de ficheiros abertos no processo.

2.2. Transferência de ficheiros

A transferência de ficheiros inicia-se após o emissor receber "OK" do primeiro cliente através da ligação TCP. O servidor lança uma thread que periodicamente (com período `sender_timer_period` milisegundos) envia um novo pacote DATA, com mais um segmento do ficheiro. Os clientes devem enviar pacotes SRR a confirmar a receção dos dados e a pedir retransmissões, garantindo que o servidor recebe um SRR pelo menos em cada `SRR_timeout` milissegundos. Caso não receba, o servidor exclui o cliente da lista de clientes válidos. Quando esta lista fica vazia, ou quando todos os segmentos pendentes são confirmados, o servidor para a tarefa de transmissão.

Os clientes esperam num socket UDP multicast pela receção de pacotes DATA, **enviando SRR por cada 2 pacotes DATA, ou após estarem `receiver_SRR_timeout` milisegundos sem receberem nenhum pacote DATA**. O cliente pode transmitir até o máximo de três SRR seguidos sem receber pacotes DATA, desistindo da receção após o quarto período de espera por pacotes DATA.

```
Pacote DATA: { sequência contígua de }
 unsigned char tipo;     // tipo de mensagem  -  DATA=1
 short int SID;          // ID de sessão
 int seq;                // número de sequência do bloco
 int len;                // comprimento dos dados no bloco
 char *bloco;            // array de carateres do ficheiro
```

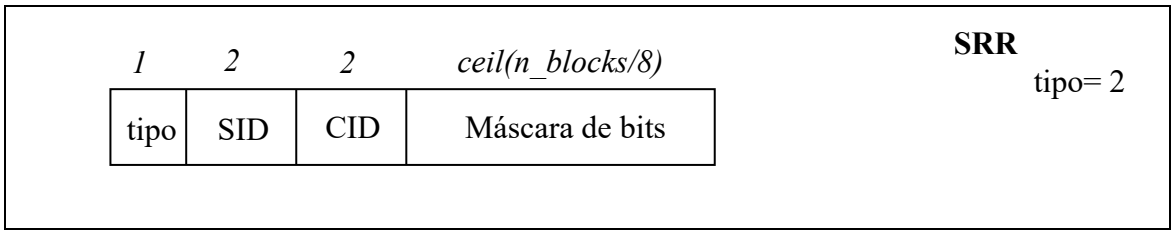
1	*2*	*4*	*4*	*len*
tipo	SID	seq	len	Bloco do ficheiro

DATA
tipo= 1

Os pacotes DATA devem ter um bloco do ficheiro com `sender_block_size` bytes, exceto o último bloco do ficheiro que pode ter menos. Para além disso, contêm o identificador de sessão e o número de sequência, que indica a posição do bloco no ficheiro.

Os pacotes SRR incluem o SID e o CID, para identificar o feixe e o recetor, e um array de caracteres que contém uma máscara de bits com um número de bits igual ao número total de pacotes que compõem o ficheiro. Cada bit representa um bloco recebido. São fornecidos aos alunos os ficheiros *bitmask.c* e *bitmask.h*, que definem o tipo `BITMASK` e funções para o manipular, de forma a facilitar a programação da manipulação de máscaras de bits.

```
Pacote SRR: { sequência contígua de }
 unsigned char tipo;     // tipo de mensagem  -  SRR=2
 short int SID;          // ID de sessão
 short int CID;          // ID de cliente
 char *bmask;            // Máscara de bits
```

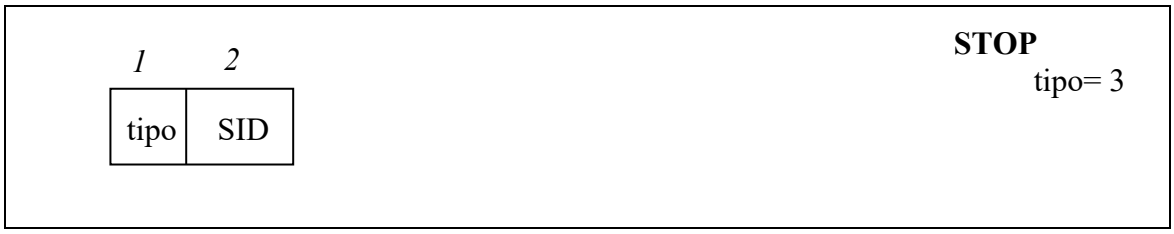


O servidor pode interromper a transmissão de um ficheiro em qualquer altura, enviando um pacote STOP. Paralelamente, um cliente pode abandonar a receção de um ficheiro, podendo avisar o servidor por dois mecanismos complementares:

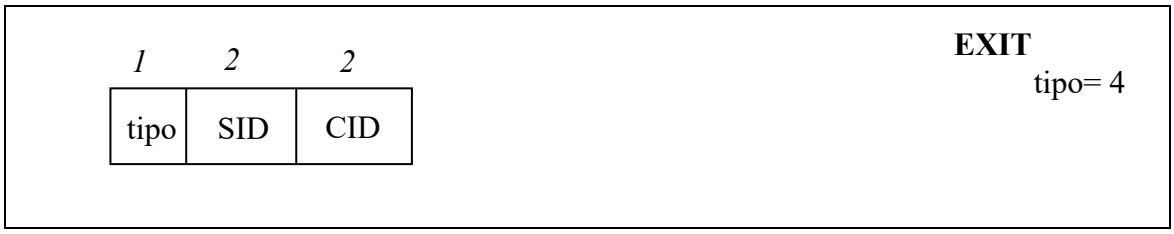
- Enviando um pacote EXIT através do socket UDP para o servidor, se já recebeu pacotes DATA;
- Enviando uma mensagem com a string "END" através da ligação TCP.

Em qualquer dos casos, o servidor exclui o cliente da lista de recetores ativos, não o contabilizando para efeitos da paragem da transmissão.

```
Pacote STOP: { sequência contígua de }
 unsigned char tipo;     // tipo de mensagem  -  STOP=3
 short int SID;          // ID de sessão
```



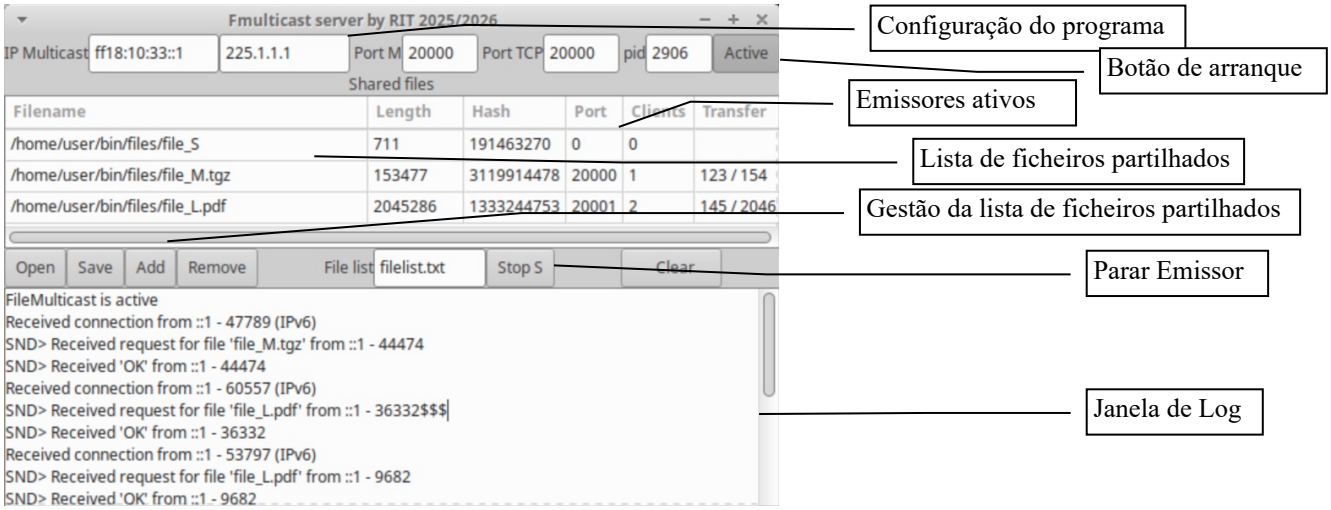

```
Pacote EXIT: { sequência contígua de }
 unsigned char tipo;     // tipo de mensagem  -  EXIT=4
 short int SID;          // ID de sessão
 short int CID;          // ID de cliente
```



3. Desenvolvimento da aplicação

Para facilitar o desenvolvimento e teste da aplicação é fornecida a definição da interface gráfica (*fmusticast_client.glade*), o código do arranque do programa (*main.c*), um conjunto de funções para gerir a interface gráfica e a tabela de threads (*gui_g3.c* e *gui.h*), que trata os eventos gráficos (*callback.c* e *callback.h*), para manipular ficheiros (*file.c* e *file.h*), para gerir sockets (*sock.c* e *sock.h*), para gerir bitmaps (*bitmap.c* e *bitmap.h*), faltando apenas programar o módulo que implementa a thread e trata a receção de ficheiro (*receiver_th.c* e *receiver_th.h*) mais algumas funções relacionadas com a comunicação multicast em IPv4. Também são fornecidos um programa servidor (*fmulticast_server* + *fmulticast_server.glade*) e um programa cliente de demonstração totalmente funcional (*demo_fmulticast_client* + *demo_fmulticast_client.glade*), com a interface gráfica descrita nas secções seguintes.

3.1. *fmulticast_server* (completo)

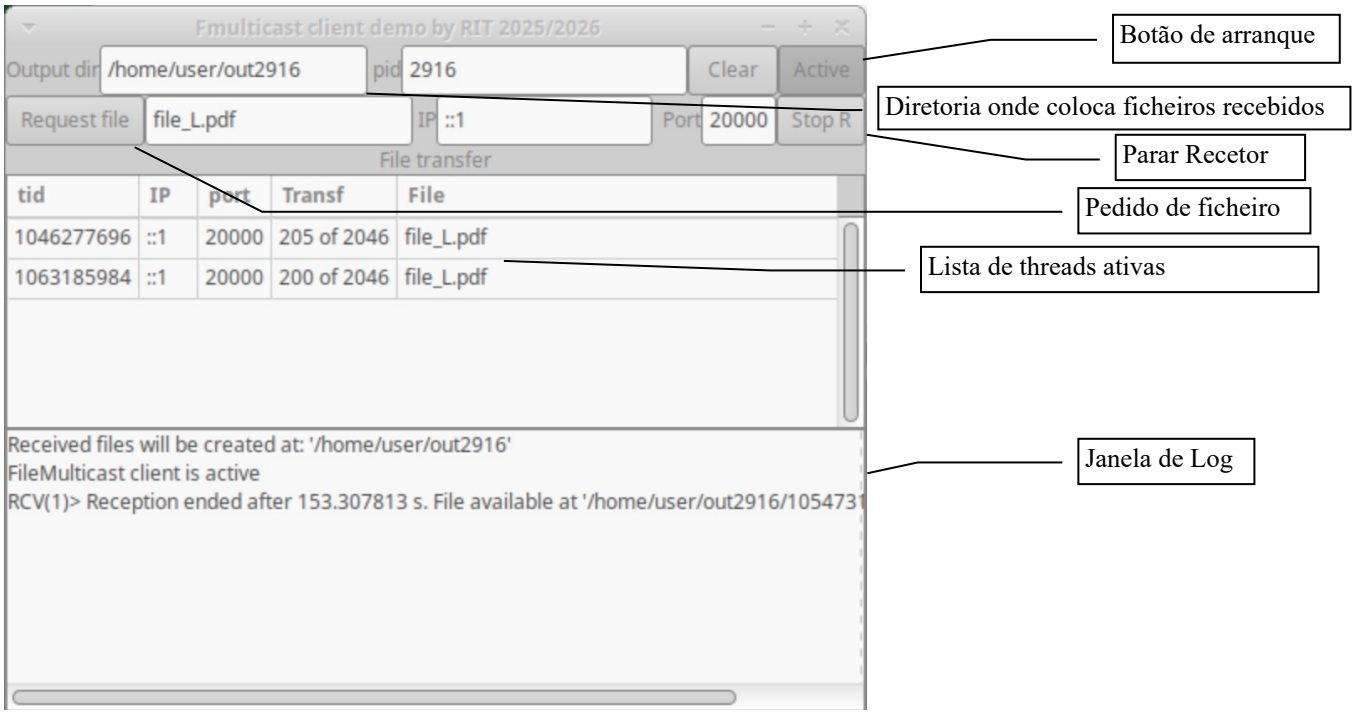


A interface contém uma linha inicial onde se configura os endereços IPv6 e IPv4 multicast, o número de porto TCP e o número de porto UDP inicial, que vai ser usado para o emissor escolher novos canais multicast (apenas muda o porto, o endereço IP é sempre o mesmo). O botão "Active" controla o arranque da aplicação. A aplicação autoconfigura os portos quando várias estão a correr na mesma máquina.

Na segunda e terceira linhas a interface contém uma tabela (do tipo *GtkTreeView*) com a lista de ficheiros partilhados (representa o nome de ficheiro, o comprimento, e o valor de *hash* do conteúdo do ficheiro nas três primeiras colunas) e os emissores ativos (representa para cada emissor de ficheiro ativo, o porto usado, o número de clientes recetores ligados, e o número total de blocos enviados com sucesso). A linha seguinte contém os comandos da lista de ficheiros: permite acrescentar (*Add*) ou remover (*Remove*) um ficheiro, e gravar (*Save*) ou ler (*Open*) um ficheiro com uma lista de ficheiros, com o nome representado na caixa à direita. Inclui também os comandos para parar um emissor (depois de selecionar uma linha), e o botão para limpar a caixa para escrever mensagens. Estas duas linhas estão completamente programadas.

A quarta, e última linha, contém uma caixa para escrever mensagens.

3.2. *demo_fmulticast_client* (completo)



A aplicação cliente não requer nenhuma configuração inicial. A interface contém uma linha inicial onde se apresentam informações sobre a diretoria onde se escrevem os ficheiros recebidos, o identificador de processo local (pid), o botão para limpar a janela de texto da última linha e o botão "Active" que controla o arranque da aplicação.

A segunda linha controla o pedido e a receção de ficheiros: permite pedir ficheiros definindo o nome do ficheiro, o endereço IPv6 ou IPv4 e o número do porto do programa servidor. Permite ainda parar um recetor (selecionando um recetor na tabela).

A terceira linha contém a lista de threads de receção ativas num dado instante, com o número de tid (*thread id*), endereço IPv6 do emissor, o número de blocos recebidos / total do ficheiro, e o nome do ficheiro que está a receber.

A quarta, e última linha, contém uma caixa para escrever mensagens.

3.3. *fmulticast_client* (a desenvolver)

O programa fornecido é composto por sete módulos:

- *bitmask.c* e *bitmask.h* – Máscara de bits, **completo**;
- *gui_g3.c* e *gui.h* – Gestão da interface gráfica, **completo**;
- *callbacks.c* e *callbacks.h* – Tratamento de pedidos da interface gráfica, **completo**;
- *file.c* e *file.h* – Manipulação de ficheiros, **completo**;
- *sock.c* e *sock.h* – Funções para manipular sockets, **completo**;
- *main.c* – Arranque da aplicação, **completo**;
- *receiver_th.c* e *receiver_th.h* – Código **incompleto** da funcionalidade de receção de ficheiro, a completar no trabalho;
- *fmulticast_client.glade* – Definição da interface gráfica.

Todo o código a desenvolver vai ser colocado, ou invocado a partir da função "`receiver_thread_function`", que contém o código corrido na *thread* de receção de ficheiros, com o algoritmo de receção. A lista de funções a modificar é a seguinte:

```
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);
```

Os dados associados a cada *thread* são guardados numa estrutura do tipo `ReceiverTh`, que inclui dados sobre o estado da ligação, as ligações TCP (*addr*, *is_ipv4*, *st*) e receção por UDP (*sm*, *bmask*, *saddr_def*, *u*, *cid*, *sid*). Pode acrescentar mais campos à estrutura.

O endereço do emissor (servidor) é memorizado quando é recebido o primeiro pacote UDP no socket multicast (*sm*). Após receber o primeiro pacote, a variável *saddr_def* é colocada a TRUE e o endereço do emissor é guardado em *u.saddr6* ou *u.saddr4*, respetivamente para IPv6 e 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 Metas

Uma sequência para o desenvolvimento do programa poderá ser:

0. Ler e compreender o código que é fornecido com o enunciado do trabalho (ANTES DA PRIMEIRA AULA);
1. Começar a programação da *thread* suportando apenas endereços IPv6. Na função `receiver_thread_function`, deve começar com a criação do socket TCP e o estabelecimento da ligação TCP para o servidor;
2. Programar a comunicação TCP em IPv6: realizar a troca de informação com o emissor até ao envio da mensagem "OK";
3. Programar o ciclo de receção de dados do socket UDP em paralelo com dados do socket TCP, usando a função `select`, realizando o protocolo descrito no enunciado;
4. Completar as funções *send_SRR*, *send_EXIT* e *receiver_thread_function* de forma a suportarem IPv4.
5. Completar o código desenvolvido nos pontos anteriores tornando-o robusto a faltas de resposta pelos emissores ou a erros inesperados.

A utilização de *threads* torna mais complicadas a tarefa de *debugging* do código e a gestão do paralelismo no acesso à GUI.

Para se conseguir chegar ao fim das quatro semanas do trabalho com tudo pronto, é necessário utilizar TODAS AS AULAS PRÁTICAS, devendo-se: começar a fase 3 durante a aula 2; começar a fase 4 durante a aula 3, concluindo a fase 5 e os ajustes de última hora durante a aula 4. Não deixe para a última semana o que pode fazer ao longo das três primeiras semanas, porque **NÃO VAI CONSEGUIR FAZER TODO O TRABALHO NA ÚLTIMA SEMANA**!

Postura dos Alunos

Cada grupo deve ter em consideração o seguinte:

- Não perca tempo com a estética de entrada e saída de dados
- Programe de acordo com os princípios gerais de uma boa codificação (utilização de indentação, apresentação de comentários, uso de variáveis com nomes conformes às suas funções...) e
- Proceda de modo a que o trabalho a fazer fique equitativamente distribuído pelos dois membros do grupo.