

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

REDES INTEGRADAS DE TELECOMUNICAÇÕES

2025 / 2026

Mestrado Integrado / Mestrado em Engenharia Eletrotécnica
e de Computadores

4º ano / 1º ano

**Introdução ao desenvolvimento de aplicações em Gnome 3:
Desenvolvimento de Aplicações *dual stack* com *sockets* e
endereços Multicast**

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

Luis Bernardo

Índice

1. Objetivo	3
2. Ambiente de desenvolvimento de aplicações em C/C++ no Linux	3
2.1. <i>Sockets</i>	3
2.1.1. <i>Sockets</i> datagrama (UDP)	5
2.1.2. <i>Sockets</i> TCP	6
2.1.3. Configuração dos <i>sockets</i>	7
2.1.4. IP Multicast	8
2.1.4.1. Associação a um endereço IPv4 Multicast	8
2.1.4.2. Associação a um endereço IPv6 Multicast	8
2.1.4.3. Partilha do número de porto	9
2.1.4.4. Definição de um tempo máximo de espera (timeout) para uma leitura	9
2.1.4.5. Definição do alcance de um grupo Multicast	9
2.1.4.6. Eco dos dados enviados para o <i>socket</i>	10
2.1.5. Funções auxiliares	10
2.1.5.1. Conversão entre formatos de endereços	10
2.1.5.2. Obter o endereço IPv4 ou IPv6 local	11
2.1.5.3. Obter número de porto associado a um <i>socket</i>	12
2.1.5.3. Espera em vários <i>sockets</i> em paralelo	12
2.1.5.4. Obter o tempo atual e calcular intervalos de tempo	13
2.1.5.5. Outras funções	13
2.1.6. Estruturas de Dados	14
2.1.7. Concorrência baseada em tarefas (<i>threads</i>)	15
2.1.7.1. Criação de <i>threads</i>	15
2.1.7.2. Sincronização entre <i>threads</i>	16
2.1.8. Concorrência baseada em subprocessos	17
2.1.8.1. Criação de subprocessos	17
2.1.8.2. Sincronização entre processos	17
2.1.9. Leitura e escrita de ficheiros binários	18
2.1.10. Temporizadores fora do ambiente gráfico	19
2.2. Aplicações com Interface gráfica Gtk+/Gnome	19
2.2.1. Editor de interfaces gráficas Glade-3	19
2.2.2. Funções auxiliares	23
2.2.2.1. Tipos de dados da biblioteca Gnome (glib) – lista (GList)	24
2.2.2.2. Funções para manipular strings	24
2.2.2.3. Aceder a objetos gráficos	24
2.2.2.4. Terminação da aplicação	24
2.2.2.5. Eventos externos – <i>sockets</i> e pipes	24
2.2.2.6. Eventos externos – <i>timers</i>	25
2.2.3. Utilização de <i>threads</i> em aplicações com interface gráfica	25
2.2.4. Utilização de subprocessos em aplicações com interface gráfica	27
2.3. O ambiente integrado Eclipse para C/C++	27
2.4. Configuração do Linux para correr aplicações <i>dual-stack</i> multicast	28
3. Exemplos de Aplicações	29
3.1. Cliente e Servidor UDP para IPv4 Multicast em modo texto	29
3.2. Cliente e Servidor UDP para IPv6 Multicast em modo texto	31
3.3. Cliente e Servidor TCP para IPv6 em modo texto	33
3.4. Programa com <i>threads</i> em modo texto	35
3.5. Programa com subprocessos em modo texto	36
3.6. Cliente e Servidor UDP com interface gráfico	38
3.6.1. Servidor	38
3.6.2. Cliente	43
3.6.3. Exercícios	48
3.7. Cliente e Servidor TCP com interface gráfico	49
3.7.1. Servidor	49
3.7.2. Cliente	50
3.7.3. Exercícios	51

1. OBJETIVO

Familiarização com o ambiente Linux e com o desenvolvimento de aplicações *dual stack* utilizando *sockets*, a biblioteca gráfica Gtk3/Gnome3, a ferramenta Glade-3, e o ambiente de desenvolvimento Eclipse para C/C++. Este documento inclui uma parte inicial, com a descrição da interface de programação, seguida de vários programas de exemplo. O enunciado descreve os exemplos e o método de introdução do código no ambiente de desenvolvimento. Para melhor aproveitar este enunciado, sugere-se que analise o código fornecido e que complete os exercícios propostos, de forma a aprender a utilizar o ambiente e as ferramentas.

2. AMBIENTE DE DESENVOLVIMENTO DE APLICAÇÕES EM C/C++ NO LINUX

O sistema operativo Linux inclui os compiladores '*gcc*' e '*g++*' que são usados para desenvolver aplicações, respetivamente nas linguagens de programação 'C' e 'C++'. Existem várias bibliotecas e ambientes gráficos que podem ser usados para realizar interfaces de aplicação com o utilizador. As duas mais comuns são o KDE e o Gnome, associadas também a dois ambientes gráficos distintos utilizáveis no sistema operativo Linux. No segundo trabalho da disciplina de RIT vai ser usada a biblioteca gráfica do Gnome 3, designada de Gtk3. O ambiente de desenvolvimento de aplicações Eclipse tem algumas semelhanças com o usado no NetBeans, funcionando como um ambiente integrado (uma interface única) a partir de onde se realiza o desenho de interfaces, edição do código, compilação e teste. No trabalho vai-se usar uma aplicação separada para realizar o desenho de interfaces (*glade*), que corre dentro do ambiente gráfico do Eclipse. Para consultar o manual das funções e bibliotecas pode ser usado o comando *man* na linha de comando, a aplicação *DevHelp*, ou são consultadas páginas Web com documentação das interfaces de programação. Tudo o resto pode ser realizado dentro do ambiente integrado Eclipse, embora também pudessem ser usados outros editores de código (e.g. *netbeans* para C/C++, *kate*, *gedit*, *vi*, etc.).

Começa-se o desenvolvimento de uma aplicação no editor da interface gráfica, que cria um ficheiro XML com a definição dos nomes dos elementos gráficos e das funções que são usadas para tratar eventos (e.g. pressão de botões de rato ou teclas). Partindo de uma estrutura base de código C comum para programas com interface Gtk3 fornecida, o programador tem então de definir as funções e símbolos gráficos definidos no ficheiro XML. Para além disso, o programador tem de acrescentar as variáveis não gráficas (*sockets*, ficheiros, comunicação entre processos, etc.) e de escrever o código para inicializar as variáveis e as rotinas de tratamento de todos os eventos.

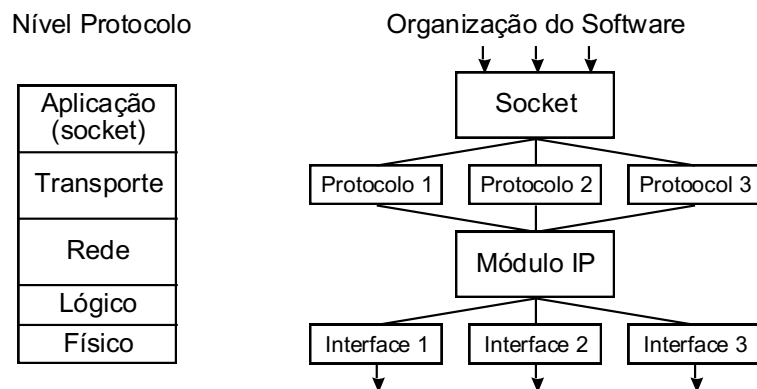
Nesta secção começa-se por introduzir a interface *socket*, utilizada para enviar mensagens UDP ou TCP. Em seguida introduzem-se as interfaces de gestão e de comunicação entre processos (*pipes*, *sockets* e sinais) e com POSIX *threads*. Na segunda parte introduz-se o desenvolvimento de aplicações usando o Glade-3 e a biblioteca gráfica Gtk3.

2.1. Sockets

Quando os *sockets* foram introduzidos no sistema Unix, na década de 70, foi definida uma interface de baixo nível para a comunicação inter-processos, que foi adotada em praticamente

todos os sistemas operativos. Nas disciplinas anteriores do curso esta interface foi usada indiretamente através de objetos complexos, que foram chamados abusivamente de *sockets*.

Um *socket* permite oferecer uma interface uniforme para qualquer protocolo de comunicação entre processos. Existem vários domínios onde se podem criar *sockets*. O domínio AF_UNIX é definido localmente a uma máquina. O domínio AF_INET suporta qualquer protocolo de nível transporte que corra sobre IPv4. O domínio AF_INET6 suporta qualquer protocolo de nível transporte que corra sobre IPv6 (ou IPv4 com pilha dupla). Um *socket* é identificado por um descritor de ficheiro, criado através da função *socket*. Ao invocar esta operação indica-se o protocolo usado através de dois campos. O primeiro seleciona o tipo de serviço (feixe fiável ou datagrama) e o segundo, o protocolo (0 especifica os protocolos por omissão: TCP e UDP). No caso dos *sockets* locais (AF_UNIX), pode-se usar a função *socketpair*, ilustrada na secção 2.1.8.



```
int socket (int domain, int type, int protocol);
```

Cria um porto para comunicação assíncrona, bidirecional e retorna um descritor (idêntico aos utilizados nos ficheiros e *pipes*).

domain - universo onde o socket é criado, que define os protocolos e o espaço de nomes.

AF_UNIX - Domínio Unix, local a uma máquina.

AF_INET - Domínio IPv4, redes Internet IPv4.

AF_INET6 - Domínio IPv6, redes Internet IPv6 ou *dual stack*.

type

SOCK_STREAM - socket TCP.

SOCK_DGRAM - socket UDP.

protocol - depende do domínio. Normalmente é colocado a zero, que indica o protocolo por omissão no domínio respetivo (TCP, UDP).

Por omissão, um *socket* não tem nenhum número de porto atribuído. A associação a um número de porto é realizada através da função *bind*. O valor do porto pode ser zero, significando que é atribuído dinamicamente pelo sistema.

```
int bind (int s, struct sockaddr *name, int namelen);
```

Associa um nome a um *socket* já criado.

s - identificador do socket.

name - o nome depende do domínio onde o socket foi criado. No domínio UNIX corresponde a um "*pathname*". Nos domínios AF_INET e AF_INET6 são respetivamente, dos tipos **struct sockaddr_in** e **struct sockaddr_in6**, que são compostos pelo endereço da máquina, protocolo e número de porto.

namelen - inteiro igual a `sizeof(*name)`

Exemplo de atribuição do número de porto com um valor dinâmico definido pelo sistema para um *socket* IPv4:

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

...
struct sockaddr_in name;

...
name.sin_family = AF_INET;           // Domínio Internet
name.sin_addr.s_addr = INADDR_ANY; // Endereço IP local (0.0.0.0)
name.sin_port = htons(0);           // Atribuição dinâmica
if (bind(sock, (struct sockaddr *)&name, sizeof(name))) {
    perror("Erro na associação a porto"); ...
}
```

Exemplo de atribuição do número de porto com um valor dinâmico definido pelo sistema para um *socket* IPv6 ou *dual stack* (IPv6+IPv4):

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

...
struct sockaddr_in6 name;
unsigned short int porto;

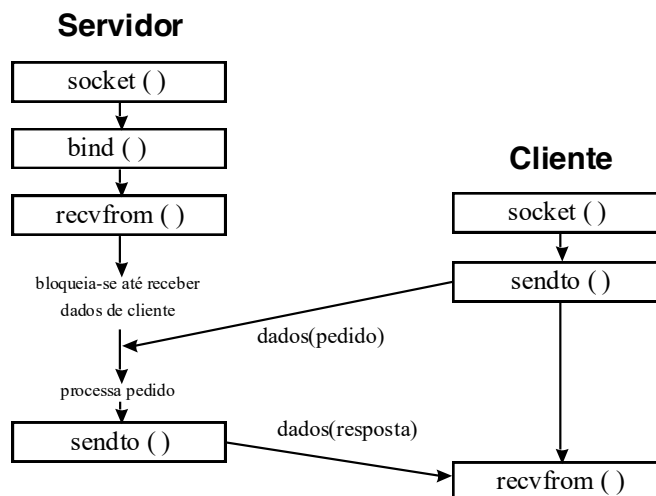
...
name.sin6_family = AF_INET6;
name.sin6_flowinfo = 0;
name.sin6_port = htons(porto);           // Porto definido pelo utilizador
name.sin6_addr = in6addr_any;           // IPv6 local por omissão
if (bind(sock, (struct sockaddr *)&name, sizeof(name)) < 0) {
    perror("Erro na associação a porto"); ...
}
```

Como qualquer outro descritor de ficheiro, um *socket* é fechado através da função `close`:

```
int close (int s);
```

2.1.1. Sockets datagrama (UDP)

Depois de criado, um *socket* UDP está preparado para receber mensagens usando a função `recvfrom`, `recv` ou `read`. Estas funções são bloqueantes, exceto se já houver um pacote à espera no *socket* ou se for seleccionada a opção (*flag*) `MSG_DONTWAIT`. O envio de mensagens é feito através da função `sendto`.



```
int recvfrom (int s, char *buf, int len, int flags, struct
             sockaddr *from, int *fromlen);
```

ou

```
int recv (int s, char *buf, int len, int flags);
```

ou

```
int read (int s, char *buf, int len);
```

Recebe uma mensagem através do *socket* *s* de um *socket* remoto. Retorna o número de bytes lidos ou -1 em caso de erro.

buf - buffer para a mensagem a receber.

len - dimensão do buffer.

flags :

MSG_OOB - Out of band;

MSG_PEEK - Ler sem retirar os dados do socket;

MSG_DONTWAIT - Não esperar por mensagem.

from - endereço do socket que enviou a mensagem (retornado pela função).

fromlen - ponteiro para inteiro inicializado a `sizeof(*from)` (função retorna bytes escritos).

```
int sendto (int s, char *msg, int len, int flags, struct
            sockaddr *to, int tolen);
```

Envia uma mensagem através do *socket* *s* para o *socket* especificado em *to*.

msg - mensagem a enviar.

len - dimensão da mensagem a enviar

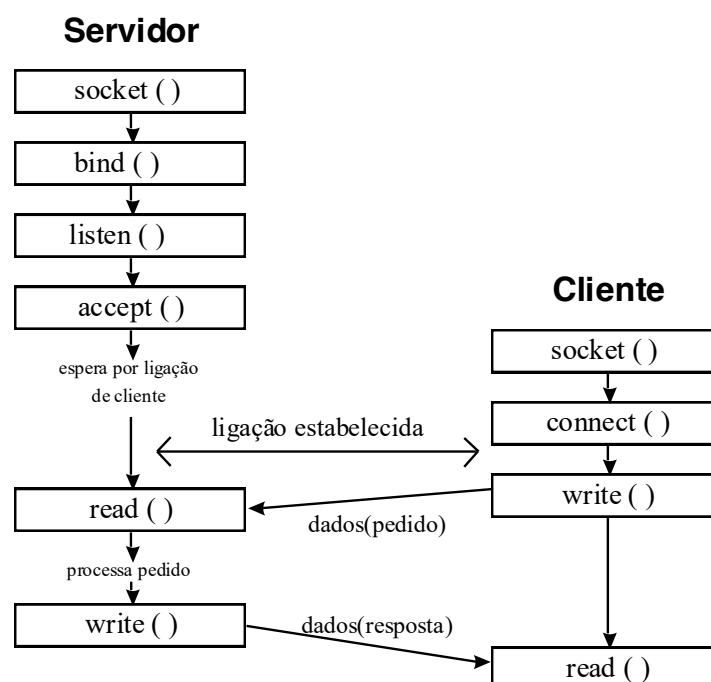
flags - 0 (sem nenhuma opção)

to - endereço do socket para onde vai ser enviada a mensagem.

tolen - inteiro igual a `sizeof(*to)`

2.1.2. Sockets TCP

Com *sockets* TCP é necessário estabelecer uma ligação antes de se poder trocar dados. Os participantes desempenham dois papéis diferentes. Um *socket* TCP servidor necessita de se



preparar para receber pedidos de estabelecimento de ligação (`listen`) antes de poder receber uma ligação (`accept`). Um *socket* TCP cliente necessita de criar a ligação utilizando a função `connect`. Após estabelecer ligação é possível receber dados com as funções `recv` ou `read`, e enviar dados com as funções `send` ou `write`.

```
int connect (int s, struct sockaddr *name, int namelen);
```

Estabelece uma ligação entre o *socket* `s` e o outro *socket* indicado em `name`.

```
int listen (int s, int backlog);  
backlog - comprimento da fila de espera de novos pedidos de ligação.
```

Indica que o *socket* `s` pode receber ligações.

```
int accept (int s, struct sockaddr *addr, int *addrlen);
```

Bloqueia o processo até um processo remoto estabelecer uma ligação. Retorna o identificador de um novo *socket* para transferência de dados.

```
int send (int s, char *msg, int len, int flags); ou  
int write(int s, char *msg, int len);
```

Envia uma mensagem através do *socket* `s` para o *socket* remoto associado. Retorna o número de bytes efetivamente enviados, ou `-1` em caso de erro. Na função `send`, o parâmetro `flags` pode ter o valor `MSG_OOB`, significando que os dados são enviados fora de banda.

```
int recv (int s, char *buf, int len, int flags); ou  
int read (int s, char *buf, int len);
```

Recebe uma mensagem do *socket* remoto através do *socket* `s`. Retorna o número de bytes lidos, ou `0` se a ligação foi cortada, ou `-1` se a operação foi interrompida. Na função `recv`, o parâmetro `flags` pode ter os valores `MSG_OOB` ou `MSG_PEEK` significando respetivamente que se quer ler dados fora de banda, ou se pretende espreitar os dados sem os retirar do *buffer*.

```
int shutdown (int s, int how);
```

Permite fechar uma das direções para transmissão de dados, dependendo do valor de `how`: `0` – só permite escritas; `1` – só permite leituras; `2` – fecha os dois sentidos.

Os *sockets* TCP podem ser usados no modo bloqueante (por omissão), onde as operações de estabelecimento de ligação, leitura ou escrita se bloqueiam até que os dados estejam disponíveis, ou no modo não bloqueante, onde retornam um erro (`EWOULDBLOCK`) quando ainda não podem ser executadas. A modificação do modo de funcionamento é feita utilizando a função `fcntl`:

```
fcntl(my_socket, F_SETFL, O_NONBLOCK); // modo não bloqueante  
fcntl(my_socket, F_SETFL, 0); // modo bloqueante (por omissão)
```

2.1.3. Configuração dos *sockets*

A interface *socket* suporta a configuração de um conjunto alargado de parâmetros dos protocolos nas várias camadas. Os parâmetros podem ser lidos e modificados respetivamente através das funções `getsockopt` e `setsockopt`.

```
#include <sys/types.h>  
#include <sys/socket.h>  
int setsockopt(int s, int level, int optname, const void *opt-val, socklen_t, *optlen);
```

A função `setsockopt` recebe como argumentos o descritor de *socket* (`s`), a camada de protocolo que vai ser configurada (`level` – `SOL_SOCKET` para o nível *socket* e `IPPROTO_TCP` para o protocolo TCP) e a identidade do parâmetro que se quer configurar (`optname`). A lista de opções suportadas para o nível IP está definida em `<bits/in.h>`. O tipo do parâmetro (`opt-`

val) passado para a função depende da opção, sendo do tipo inteiro para a maior parte dos parâmetros. A função retorna 0 em caso de sucesso.

```
int getsockopt(int s, int level, int optname, void *opt-val, socklen_t *optlen);
```

A função `getsockopt` recebe o mesmo tipo de parâmetros e permite ler os valores associados às várias opções.

Exemplos de parâmetros para *sockets* TCP são:

- `SO_RCVBUF` e `SO_SNDBUF` da camada `SOL_SOCKET` – especifica respetivamente o tamanho dos *buffers* de receção e envio de pacotes para *sockets* TCP e UDP;
- `SO_RCVTIMEO` e `SO_SNDTIMEO` da camada `SOL_SOCKET` – especifica respetivamente o tempo máximo (*timeout*) para realizar operações de receção e envio através de um *socket* TCP ou UDP;
- `SO_REUSEADDR` da camada `SOL_SOCKET` – permite que vários *sockets* partilhem o mesmo porto no mesmo endereço IP;
- `TCP_NODELAY` da camada `IPPROTO_TCP` – controla a utilização do algoritmo de Nagle;
- `SO_LINGER` da camada `IPPROTO_TCP` – controla a terminação da ligação, evitando que o *socket* entre no estado `TIME_WAIT`.

Por exemplo, para modificar a dimensão do *buffer* de envio usar-se-ia:

```
int v=64000; // bytes
if (setsockopt(s, SOL_SOCKET, SO_SNDBUF, &v, sizeof(v)) < 0) { ... erro ... }
```

2.1.4. IP Multicast

Qualquer *socket* datagrama pode ser associado a um endereço IP *Multicast*, passando a receber os pacotes difundidos nesse endereço. O envio de pacotes é realizado da mesma maneira que para um endereço *unicast*. Todas as configurações para suportar IP *Multicast* são realizadas ativando-se várias opções com a função `setsockopt`.

2.1.4.1. Associação a um endereço IPv4 Multicast

A associação a um endereço IP multicast é realizada utilizando a opção `IP_ADD_MEMBERSHIP` do nível `IPPROTO_IP`. O valor do endereço IPv4 deve ser classe D (224.0.0.0 a 239.255.255.255). O endereço "224.0.0.1" é reservado, agrupando todos os *sockets* IP Multicast.

```
struct ip_mreq imr;
if (!inet_aton("225.1.1.1", &imr.imr_multiaddr)) { /* falhou conversão */; ... }
imr.imr_interface.s_addr = htonl(INADDR_ANY); /* Placa de rede por omissão */
if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *) &imr,
    sizeof(struct ip_mreq)) == -1) {
    perror("Falhou associação a grupo IPv4 multicast"); ... }
}
```

A operação inversa é realizada com a opção `IP_DROP_MEMBERSHIP`, com os mesmos parâmetros.

2.1.4.2. Associação a um endereço IPv6 Multicast

A associação a um endereço IPv6 multicast é realizada utilizando a opção `IPV6_JOIN_GROUP` do nível `IPPROTO_IPV6`. O valor do endereço deve ser da classe multicast (`ff00::0/8`).

```
struct ipv6_mreq imr;
if (!inet_pton(AF_INET6, "ff18:10:33::1", &imr.ipv6mr_multiaddr)) { /*falhou conversão*/ }
imr.ipv6mr_interface = 0; /* Interface 0 */
if (setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, (char *) &imr,
    sizeof(imr)) == -1) {
    perror("Falhou associação a grupo IPv6 multicast"); ... }
}
```


A operação inversa é realizada com a opção `IP_LEAVE_GROUP`, com os mesmos parâmetros.

2.1.4.3. Partilha do número de porto

Por omissão apenas pode haver um *socket* associado a um número de porto. Usando a opção `SO_REUSEADDR` do nível `SOL_SOCKET` é possível partilhar um porto entre vários *sockets*, recebendo todos os sockets as mensagens enviadas para esse porto.

```
int reuse= 1;
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *) &reuse, sizeof(reuse)) < 0) {
    perror("Falhou setsockopt SO_REUSEADDR"); ...
}
```

2.1.4.4. Definição de um tempo máximo de espera (timeout) para uma leitura

Define um tempo máximo para operações de leitura usando uma variável do tipo *struct timeval*. As operações de leitura (e.g. *read*, *recv*, *recvfrom*, etc.) param ao fim do tempo especificado se não for recebido um pacote, ou retornam com os dados recebidos até essa altura, se forem insuficientes. Se não recebem nada a operação de leitura retorna -1, ficando a variável *errno* com o valor `EWOULDBLOCK`. Cancela-se definindo um tempo nulo.

```
struct timeval timeout;
timeout.tv_sec = 10; // 10 segundos
timeout.tv_usec = 0; // 0 microsegundos
if (setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout, sizeof(timeout)) < 0) {
    error("setsockopt failed\n");
}
```

2.1.4.5. Definição do alcance de um grupo Multicast

Para IPv4, o alcance de um grupo é definido apenas no envio de pacotes. O tempo de vida (TTL) de um pacote enviado para um endereço IPv4 Multicast pode ser controlado usando a opção `IP_MULTICAST_TTL`. O valor de 1 restringe o pacote à rede local. Os pacotes só são redifundidos em routers multicast para valores superiores a 1. Na rede MBone pode-se controlar o alcance pelo valor de TTL (<32 é restrito à rede da organização; <128 é restrito ao continente).

```
u_char ttl= 1; /* rede local */
if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, (char *) &ttl,
    sizeof(ttl)) < 0) {
    perror("Falhou setsockopt IP_MULTICAST_TTL");
}
```

A opção equivalente para IPv6 é `IPV6_MULTICAST_HOPS`, mas é menos usada porque, neste caso, o alcance de um grupo é definido pelo valor do endereço. Os endereços multicast IPv6 têm a seguinte estrutura:

8	4	4	112 bits
11111111	flags	scope	ID grupo

As flags contêm um conjunto de 4 bits `|0|0|0|T|`, onde apenas T está definido.

- T = 0 define um endereço multicast permanente (ver RFC 2373 e 2375);
- T = 1 define um endereço não permanente (transiente), vulgarmente usado nas aplicações de utilizador.

O *scope* define o limite de alcance do grupo multicast. Os valores são:

0 reservado	1 local ao nó	2 local à ligação	3 não atribuído
4 não atribuído	5 local ao lugar	6 não atribuído	7 não atribuído
8 local à organização	9 não atribuído	A não atribuído	B não atribuído
C não atribuído	D não atribuído	E Global	F reservado

2.1.4.6. Eco dos dados enviados para o *socket*

Com a opção `IP_MULTICAST_LOOP` é possível controlar se os dados enviados para o grupo são recebidos, ou não, no *socket* IPv4 local.

```
char loop = 1;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

Para IPv6 existe a opção equivalente: `IPV6_MULTICAST_LOOP`.

```
char loop = 1;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop, sizeof(loop));
```

2.1.5. Funções auxiliares

Para auxiliar o desenvolvimento de aplicações é usado um conjunto de funções para realizar a conversão de endereços entre o formato binário (IPv4: `struct in_addr` e IPv6: `struct in6_addr`) e string (`char *`), obter o número de porto associado a um *socket*, etc. Pode encontrar uma lista exaustiva das funções para IPv6 no RFC 3493.

2.1.5.1. Conversão entre formatos de endereços

Existem duas formas para identificar uma máquina na rede:

- pelo endereço IP (formato string ou binário) (e.g. "172.16.33.1" para IPv4, equivalente a "::ffff:172.16.33.1" para IPv6; ou "2001:690:2005:10:33::1" para IPv6 nativo);
- pelo nome da máquina (e.g. "tele33-pc1").

Para realizar a conversão entre o formato binário IPv4 (`struct in_addr`) e o formato *string* foram definidas duas funções:

```
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
```

A função `inet_aton` converte do formato string ("a"scii) para binário ("n"umber), retornando **0** caso não seja um endereço válido. A função `inet_ntoa` cria uma string temporária com a representação do endereço passado no argumento.

Posteriormente, foram acrescentadas duas novas funções que suportam endereços IPv6 e IPv4, e permitem realizar a conversão entre o formato binário e o formato *string*:

```
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

A função `inet_pton` converte do formato string ("p"ath) para binário, retornando **0** caso não seja um endereço válido. O parâmetro `af` define o tipo de endereço (`AF_INET` ou `AF_INET6`). O parâmetro `dst` deve apontar para uma variável do tipo `struct in_addr` ou `struct in6_addr`. A função `inet_ntop` cria uma *string* com o conteúdo de `src` num *array* de caracteres passado no argumento `dst`, de comprimento `size`. O *array* deve ter uma dimensão igual ou superior a `INET_ADDRSTRLEN` ou `INET6_ADDRSTRLEN` (respetivamente para IPv4

e IPv6), duas constantes declaradas em `<netinet/in.h>`. Retorna **NULL** em caso de erro, ou `dst` se conseguir realizar a conversão.

A tradução do nome de uma máquina, ou de um endereço IP, para o formato binário também pode ser realizada através das funções `gethostbyname` ou `gethostbyname2`:

```
struct hostent *gethostbyname(char *hostname);           // só para IPv4
struct hostent *gethostbyname2(char *hostname, int af);  // IPv4 ou IPv6
```

No programa seguinte apresenta-se um excerto de um programa com o preenchimento de uma estrutura `sockaddr_in` (IPv4), dado o nome ou endereço de uma máquina e o número de porto.

```
#include <netdb.h>

struct sockaddr_in addr;
struct hostent *hp;

...
hp= gethostbyname2(host_name, AF_INET);
if (hp == NULL) {
    fprintf (stderr, "%s : unknown host\n", host_name); ...
}
bzero((char *)&addr, sizeof addr);
bcopy (hp->h_addr, (char *) &addr.sin_addr, hp->h_length);
addr.sin_family= AF_INET;
addr.sin_port= htons(port number/*número de porto*/);
```

2.1.5.2. Obter o endereço IPv4 ou IPv6 local

É possível obter o endereço IPv4 ou IPv6 da máquina local recorrendo às funções anteriores e à função `gethostname`, que lê o nome da máquina local. Esta função preenche o nome no buffer recebido como argumento, retornando **0** em caso de sucesso. Este método falha quando não existe uma entrada no serviço DNS ou no ficheiro `/etc/hosts` com o nome no domínio pedido (IPv4 ou IPv6).

```
int gethostname(char *name, size_t len);
```

É possível obter o endereço IPv4 a partir do nome do dispositivo de rede (geralmente é `“eth0”`) utilizando a função `ioctl`.

```
static gboolean get_local_ipv4name_using_ioctl(const char *dev, struct in_addr *addr) {
    struct ifreq req;
    int fd;
    assert((dev!=NULL) && (addr!=NULL));
    fd = socket(AF_INET, SOCK_DGRAM, 0);
    strcpy(req.ifr_name, dev);
    req.ifr_addr.sa_family = AF_INET;
    if (ioctl(fd, SIOCGIFADDR, &req) < 0) {
        perror("getting local IP address");
        close(fd);
        return FALSE;
    }
    close(fd);
    struct sockaddr_in *pt= (struct sockaddr_in *)&req.ifr_ifru.ifru_addr;
    memcpy(addr, &(pt->sin_addr), 4);
    return TRUE;
}
```

No entanto, a função não suporta endereços IPv6. Neste caso, uma solução possível é a invocação do comando `ifconfig` (que devolve todas as interfaces do sistema) e a aplicação de filtros à cadeia de caracteres resultante de forma a isolar o primeiro endereço global da lista, com criação de um ficheiro temporário. A função seguinte devolve uma *string* com um endereço IPv6 global no *buffer* `buf` a partir do nome do dispositivo `dev` (geralmente `“eth0”`).

```
static gboolean get_local_ipv6name_using_ifconfig(const char *dev, char *buf, int buf_len) {
    system("/sbin/ifconfig | grep inet6 | grep 'Scope:Global' | head -1 | awk '{ print $3 }' >
    /tmp/lixo0123456789.txt");
```

```
FILE *fd= fopen("/tmp/lixo0123456789.txt", "r");
int n= fread(buf, 1, buf_len, fd);
fclose(fd);
unlink("/tmp/lixo0123456789.txt"); // Apaga ficheiro

if (n <= 0) return FALSE;
if (n >= 256) return FALSE;
char *p= strchr(buf, '/');
if (p == NULL) return FALSE;
*p= '\0';
return TRUE; // Devolve o endereço Ipv6 em 'buf'
}
```

2.1.5.3. Obter número de porto associado a um *socket*

A função `getsockname` permite obter uma estrutura que inclui todas as informações sobre o *socket*, incluindo o número de porto, o endereço IP e o tipo de *socket*.

```
int getsockname ( int s, struct sockaddr *addr, int *addrlen );
```

Em seguida apresenta-se um excerto de um programa, onde se obtém o número de porto associado a um *socket* IPv4 *s*.

```
struct sockaddr_in addr;
int len= sizeof(addr);
...
if (getsockname(s, (struct sockaddr *)&addr, &len)) {
    perror("Erro a obter nome do socket"); ... }
if (addr.sin_family != AF_INET) { /* Não é socket IPv4 */ ... }
printf("O socket tem o porto %d\n", ntohs(addr.sin_port));
```

O código equivalente para um *socket* IPv6 seria.

```
struct sockaddr_in6 addr;
int len= sizeof(addr);
...
if (getsockname(s, (struct sockaddr *)&addr, &len)) {
    perror("Erro a obter nome do socket"); ... }
if (addr.sin6_family != AF_INET6) { /* Não é socket IPv6 */ ... }
printf("O socket tem o porto %d\n", ntohs(addr.sin6_port));
```

2.1.5.3. Espera em vários *sockets* em paralelo

A maior parte das primitivas apresentadas anteriormente para aceitar novas ligações e para receber dados num *socket* TCP ou UDP são bloqueantes. Para realizar aplicações que recebem dados de vários *sockets*, do teclado e de eventos de rato foi criada a função `select` que permite esperar em paralelo dados de vários descritores de ficheiro. Como quase todos os tipos de interação podem ser descritos por um descritor de ficheiro, a função é usada por quase todas as aplicações. As exceções são as aplicações multi-tarefa, onde pode haver várias tarefas ativas em paralelo, cada uma a tratar um *socket* diferente.

```
int select ( int width, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timeval *timeout );
```

Esta função recebe como argumento três *arrays* de bits, onde se indica quais os descritores de ficheiros (associados a protocolos de Entrada/Saída) onde se está à espera de receber dados (`readfds` - máscara de entrada), onde se está à espera de ter espaço para continuar a escrever (`writefds` - máscara de escrita) e onde se quer receber sinalização de erros (`exceptfds` - máscara de exceções). O campo `width` deve ser preenchido com o maior valor de descritor a considerar na máscara adicionado de um. Esta função bloqueia-se até que seja recebido um dos eventos pedidos, ou até que expire o tempo máximo de espera (definido em `timeout`). Retorna o número de eventos ativados, ou **0** caso tenha expirado o temporizador, ou **-1** em caso de erro. Os eventos ativos são identificados por bits a um nas máscaras passadas nos argumentos. Em `timeout` a função devolve o tempo que faltava para expirar o tempo de espera quando o evento foi recebido. Para lidar com máscaras de bits, do tipo `fd_set`, são fornecidas as seguintes quatro funções:

```

FD_ZERO (fd_set *fdset)           // Coloca todos os bits da máscara a 0.
FD_SET (int fd, fd_set *fdset)    // Liga o bit correspondente ao descritor de ficheiro fd.
FD_CLR (int fd, fd_set *fdset)    // Desliga o bit correspondente ao descritor fd.
FD_ISSET (int fd, fd_set *fdset)  // Testa se o bit correspondente ao descritor de ficheiro fd
    está ativo.

```

O código seguinte ilustra a utilização da função `select` para esperar durante dois segundos sobre dois descritores de ficheiros de *sockets* em paralelo:

```

struct timeval tv;
fd_set rmask;           // mascara de leitura
int sd1, sd2,           // Descritores de sockets
    n, max_d;
FD_ZERO(&rmask);
FD_SET(sd1, &rmask);    // Regista socket sd1
FD_SET(sd2, &rmask);    // Regista socket sd2
max_d= max(sd1, sd2)+1; // teoricamente pode ser getdtablesize();
tv.tv_sec= 2;           // segundos
tv.tv_usec= 0;          // microsegundos
n= select (max_d, &rmask, NULL, NULL, &tv);
if (n < 0) {
    perror ("Interruption of select"); // errno = EINTR foi interrompido
} else if (n == 0) {
    fprintf(stderr, "Timeout\n"); ...
} else {
    if (FD_ISSET(sd1, &rmask)) { // Há dados disponíveis para leitura em sd1
        ...
    }
    if (FD_ISSET(sd2, &rmask)) { // Há dados disponíveis para leitura em sd2
        ...
    }
}
}

```

A função `select` está na base dos sistemas que suportam pseudo-paralelismos baseados em eventos, estando no núcleo do ciclo principal da biblioteca gráfica Gnome/Gtk+ e de outros ambientes de programação (e.g. Delphi). Nestes ambientes a função é usada indiretamente, pois o Gnome permite registar funções para tratar eventos de leitura, escrita ou tratamento de exceções no ciclo principal da biblioteca gráfica.

2.1.5.4. Obter o tempo atual e calcular intervalos de tempo

Existem várias funções para obter o tempo (`time`, `ftime`, `gettimeofday`, etc.). Utilizando a função `gettimeofday` obtém-se o tempo com uma precisão de milisegundos. Para calcular a diferença de tempos, basta calcular a diferença entre os campos (`tv_sec` – segundos) e (`tv_usec` – microsegundos) dos dois tempos combinando-os.

```

struct timezone tz;
struct timeval tv;
if (gettimeofday(&tv, &tz))
    perror("error getting time of day ");

```

2.1.5.5. Outras funções

A maior parte das funções apresentadas anteriormente modifica o valor da variável **errno** após retornarem um erro. Para escrever o conteúdo do erro na linha de comando é possível usar a função `perror` que recebe como argumento uma *string*, que concatena antes da descrição do último erro detetado.

Outro conjunto de funções lida com sequências de bytes arbitrarias e com conversão de formatos de inteiros binários:

Chamada	Descrição
bcmp (void*s1,void*s2, int n) bcopy (void*s1,void*s2, int n) memmove (void*s2,void*s1, int n) bzero (void *base, int n) long htonl (long val) short htons (short val) long ntohl (long val) short ntohs (short val)	Compara sequências de bytes; retornando 0 se iguais Copia n bytes de s1 para s2 (s1 e s2 separados) Copia n bytes de s1 para s2 (s1 e s2 com sobreposição) Enche com zeros n bytes começando em base Converte ordem de bytes de inteiros 32-bit de host para rede Converte ordem de bytes de inteiros 16-bit de host para rede Converte ordem de bytes de inteiros 32-bit de rede para host Converte ordem de bytes de inteiros 16-bit de rede para host

As quatro últimas funções (definidas em `<netinet/in.h>`) visam permitir a portabilidade do código para máquinas que utilizem uma representação de inteiros com uma ordenação dos bytes diferente da ordem especificada para os pacotes e argumentos das rotinas da biblioteca de *sockets*. Sempre que se passa um inteiro (*s*)hort (16 bits) ou (*l*)ong (32 bits) como argumento para uma função de biblioteca de *sockets* este deve ser convertido do formato (*h*)ost (máquina) para o formato (*n*)etwork (rede). Sempre que um parâmetro é recebido deve ser feita a conversão inversa: (*n*)to(*h*).

O comando 'man' pode ser usado para obter mais informações sobre o conjunto de comandos apresentado.

2.1.6. Estruturas de Dados

Os nomes dos *sockets* são definidos como especializações da estrutura:

```
<sys/socket.h>:
struct sockaddr {
    u_short sa_family;           // Address family : AF_XXX
    char     sa_data[14];       // protocol specific address
};
```

No caso dos *sockets* do domínio AF_INET, usado na Internet (IPv4), é usado o tipo struct sockaddr_in, com o mesmo número de bytes do tipo genérico. Como a linguagem C não suporta a definição de relações de herança entre estruturas, é necessário recorrer a mudanças de tipo explícitas (struct sockaddr *) para evitar avisos durante a compilação.

```
<netinet/in.h>:
struct in_addr {
    u_long          s_addr;           /* 32-bit netid/hostid network byte ordered */
};
struct sockaddr_in {
    short           sin_family;       /* AF_INET */
    u_short         sin_port;         /* 16-bit port number network byte ordered */
    struct in_addr  sin_addr;         /* 32-bit netid/hostid network byte ordered */
    char            sin_zero[8];      /* unused */
};
```

No caso dos *sockets* do domínio AF_INET6 (IPv6) é usado o tipo struct sockaddr_in6.

```
<netinet/in.h>:
struct in6_addr {
    union {
        uint8_t      u6_addr8[16];
        uint16_t     u6_addr16[8];
        uint32_t     u6_addr32[4];
    } in6_u;
#define s6_addr          in6_u.u6_addr8
#define s6_addr16       in6_u.u6_addr16
#define s6_addr32       in6_u.u6_addr32
};
struct sockaddr_in6 {
```

```

short          sin6_family;          /* AF_INET6 */
in_port_t     sin6_port;             /* Transport layer port # */
uint32_t      sin6_flowinfo; /* IPv6 flow information */
struct in6_addr sin6_addr;          /* IPv6 address */
uint32_t      sin6_scope_id; /* IPv6 scope-id */
};

```

A estrutura `struct hostent` é retornada pela função `gethostbyname` com uma lista de endereços associados ao nome.

```

<netdb.h>:
struct hostent {
    char *h_name;          /* official name of host */
    char **h_aliases;      /* alias list */
    int h_addrtype;        /* host address type */
    int h_length;          /* length of address */
    char **h_addr_list;    /* list of addresses, null terminated */
};
#define h_addr h_addr_list[0] // first address, network byte order

```

A estrutura `struct ip_mreq` é usada nas rotinas de associação a endereços IPv4 Multicast.

```

<bits/in.h>:
struct ip_mreq {
    struct in_addr imr_multiaddr; // Endereço IP multicast
    struct in_addr imr_interface; // Endereço IP unicast da placa de interface
};

```

A estrutura `struct ipv6_mreq` é usada nas rotinas de associação a endereços IPv6 Multicast.

```

<netinet/in.h>:
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr; // Endereço IPv6 multicast
    unsigned int    ipv6mr_interface; // N° de interface (0 se só houver uma)
};

```

A estrutura `struct time_val` é usada nas funções *select* e *gettimeofday*.

```

<sys/time.h >:
struct time_val {
    long tv_sec;      // Segundos
    long tv_usec;     // Microsegundos
};

```

2.1.7. Concorrência baseada em tarefas (*threads*)

Para aumentar o paralelismo numa aplicação, é possível correr várias funções concorrentes (se o processador tiver vários núcleos) em C/C++. Existem dois mecanismos alternativos: as tarefas (*threads*) POSIX, semelhantes às usadas em Java; e processos concorrentes. Esta secção introduz a biblioteca *pthread*, que implementa as tarefas. A seguinte, introduz os subprocessos.

2.1.7.1. Criação de *threads*

Para aumentar o paralelismo numa aplicação é possível lançar várias tarefas concorrentes, utilizando a função `pthread_create`. Cada *thread* é identificada por um identificador do tipo `pthread_t` (em Ubuntu 18.08 e posteriores está definido como *unsigned long int*, mas pode variar noutras implementações), retornado no primeiro parâmetro (*thread*) da função. O parâmetro *attr* permite configurar parâmetros da thread (política de agendamento, prioridade, tamanho stack, etc.) – quando se usa NULL, definem-se os valores por omissão. O parâmetro *thread_routine* recebe um apontador para a função que é corrida, e o parâmetro *arg* é passado como argumento da função *thread_routine*.

```
int pthread_create(pthread_t * thread,
                  const pthread_attr_t * attr,
                  void * (*thread_routine)(void *),
                  void *arg);
```

A assinatura de uma função corrida numa *thread* recebe em *ptr* o valor introduzido em *arg*, na chamada a `pthread_create` e retorna um apontador genérico. Caso não seja necessário retornar nada, pode devolver NULL. Caso contrário, deverá devolver um apontador para memória dinâmica (criada com *malloc*) ou para uma variável global.

```
void *thread_routine ( void *ptr )
{
    /* function code */

    return NULL; /* It can be any pointer, passed to the main thread */
}
```

Na função *thread_routine*, é possível terminar a *thread* em qualquer ponto do código, usando a função `pthread_exit`, indicando o valor retornado.

```
void pthread_exit(void *retval);
```

Quando é necessário recolher dados calculados numa *thread*, é necessário invocar a função `pthread_join`, que se bloqueia à espera que a tarefa com o identificador *th* termine, recolhendo no argumento *thread_return* o valor retornado pela função. Se não forem retornados valores, não é necessário invocar esta função.

```
int pthread_join(pthread_t th, void **thread_return);
```

É possível parar uma *thread* usando o comando `pthread_cancel`, mas não se recomenda a sua utilização. É preferível colocar no código da *thread* a verificação a uma variável global, que permita sair de uma forma limpa (sem interromper a execução de funções deixando o estado global incoerente).

Para compilar código que use POSIX threads é necessário usar a opção `-lpthread` no comando de compilação e linkagem.

Tal como nas tarefas usadas em Java, todas as tarefas partilham as mesmas variáveis. Caso exista a necessidade de sincronizar o acesso a recursos não partilháveis, ou a coordenar a ordem de execução das tarefas, pode ser necessário recorrer a mecanismos adicionais, apresentados na secção seguinte.

2.1.7.2. Sincronização entre threads

Para além da espera pelo fim de uma *thread*, existem outros mecanismos de controlo de concorrência entre *threads*. Neste documento apenas são apresentados os semáforos (*Mutex*), que controlam o acesso exclusivo a troços de código. Para uma introdução rápida a outros mecanismos recomenda-se uma leitura a documentos externos¹.

Um semáforo é criado com a macro `PTHREAD_MUTEX_INITIALIZER` e guardado numa variável do tipo `pthread_mutex_t`. Qualquer bloco de código que só possa ser acedido por uma *thread* deve ser precedido do bloqueio do semáforo com a função `pthread_mutex_lock` e com o desbloqueio no fim, com a função `pthread_mutex_unlock`, como está representado no exemplo seguinte.

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

void function()
{
    ...
    pthread_mutex_lock( &mutex1 );
    /* code that can only be run by one thread */
}
```

¹ Na página [POSIX Threads Libraries](http://YoLinux.com) (YoLinux.com) e [Multithreaded programming \(POSIX pthreads tutorial\)](#) pode encontrar mais informação sobre os mecanismos de sincronização entre threads.


```
pthread_mutex_unlock( &mutex1 );  
...  
}
```

Relembra-se que a utilização de semáforos deve ser pensada de maneira a evitar *deadlocks*, por terem tarefas bloqueadas à espera de semáforos que não vão ser libertados, por existirem dependências cruzadas.

2.1.8. Concorrência baseada em subprocessos

2.1.8.1. Criação de subprocessos

Para aumentar o paralelismo numa aplicação é possível criar vários processos que correm em paralelo, utilizando a função `fork`. Ao contrário das tarefas usadas em Java, **cada processo tem a sua cópia privada das variáveis**, sendo necessário recorrer a canais externos (descritos na secção 2.1.8.2) para se sincronizarem os vários processos. Cada processo é identificado por um inteiro (o *pid* – *process id*), que pode ser consultado na linha de comando com a instrução (`ps axu`).

Quando a função `fork` é invocada, o processo inicial é desdobrado em dois, exatamente com as mesmas variáveis, com os mesmos ficheiros, *pipes* e *sockets* abertos. O valor retornado permite saber se é o processo original (retorna o *pid* do subprocesso criado), ou se é o processo filho (retorna **0**). Em caso de não haver memória para a criação do subprocesso retorna **-1**.

```
#include <sys/types.h>  
#include <unistd.h>  
pid_t fork(void);
```

Os processos filhos correm em paralelo com o processo pai, mas só morrem completamente após o processo pai invocar uma variante da função `wait` (geralmente `wait3`, ou `wait4` quando se pretende bloquear o pai à espera do fim de um subprocesso). Antes disso, ficam num estado *zombie*.

```
#include <sys/types.h>  
#include <sys/time.h>  
#include <sys/resource.h>  
#include <sys/wait.h>  
pid_t wait3(int *status, int options, struct rusage *rusage);  
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

A função `wait3` por omissão é bloqueante, exceto se usar o parâmetro `options` igual a `WNOHANG`. Nesse caso, retorna **-1** caso não exista nenhum subprocesso *zombie* à espera. Após a terminação de um processo filho, é gerado um sinal `SIGCHLD` no processo pai. Pode-se evitar que o processo pai fique bloqueado processando este sinal, e indiretamente, detetando falhas nos subprocessos. As funções retornam o parâmetro `status`, que permite detetar se o processo terminou normalmente invocando a operação `_exit`, ou se terminou com um erro (que gera um sinal associado a uma exceção). No exemplo da secção 3.5 está ilustrado como se pode realizar esta funcionalidade.

2.1.8.2. Sincronização entre processos

Para além dos *sockets*, é possível usar vários outros tipos de mecanismos de sincronização entre processos locais a uma máquina. Quando se usam subprocessos, é comum usar *pipes* ou *sockets* locais para comunicar entre o processo pai e o processo filho. Um *pipe* é um **canal unidirecional** local a uma máquina semelhante a um *socket* TCP – tudo o que se escreve no descritor `p[1]` é enviado para o descritor `p[0]`. A função `pipe` cria dois descritores de ficheiros (equivalentes a *sockets*). Caso o *pipe* seja criado antes de invocar a operação `fork`, ele é conhecido de ambos os processos. Para manter um canal aberto para a comunicação entre um processo pai e o processo filho, eles apenas têm de fechar uma das extremidades (cada um) e comunicar entre eles através do canal criado usando as instruções `read` e `write`. Caso se

pretenda ter **comunicação bidirecional**, pode-se usar a função `socketpair` para criar um par de *sockets*.

```
int p[2], b[2];          // descritor de pipe, ou socket

if (pipe(p))              // Só suporta comunicação    p[1] -> p[0]
    perror("falhou pipe");

if (socketpair(AF_UNIX, SOCK_STREAM, 0, b) < 0) // Canal bidirecional
    perror("falhou o socketpair");
```

Os sinais, para além de serem usados para notificar eventos assíncronos (morte de subprocesso, dados fora de banda, etc.), os sinais também podem ser usados na comunicação entre processos. Existem dois sinais reservados para esse efeito (`SIGUSR1` e `SIGUSR2`), que podem ser gerados utilizando a função `kill`. A utilização de sinais é ilustrada no exemplo da secção 3.5.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

A receção de sinais é realizada através de uma função de *callback* (e.g. `handler`), associada a um sinal através da função `signal`. Após o sinal, o sistema operativo interrompe a aplicação e corre o código da função *callback*, retornando depois ao ponto onde parou. Chamadas de leitura bloqueantes são interrompidas, devolvendo erro, com `errno==EINTR`.

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

2.1.9. Leitura e escrita de ficheiros binários

Pode-se ler e escrever em ficheiros binários utilizando descritores de ficheiro (*int*) ou descritores do tipo “*FILE **”. No exemplo fornecido de seguida para a cópia de ficheiros são usados dois descritores do segundo tipo: um para leitura (*f_{in}*) e outro para escrita (*f_{out}*). Um ficheiro é aberto utilizando a função *fopen*, que recebe como argumento o modo de abertura: leitura “*r*”, escrita no início do ficheiro “*w*”, escrita no fim do ficheiro “*a*”, ou num dos vários modos de leitura escrita (“*r+*”, “*w+*” ou “*a+*”). A leitura é realizada com a função *fread*, onde se especifica o tamanho de cada elemento e o número de elementos a ler, retornando o número de elementos lido. No exemplo, pretende-se ler byte a byte, portanto o tamanho de cada elemento é 1. O valor retornado pela função *fread* pode ser: **>0**, indicando que leu dados com sucesso; **=0**, indicando que se atingiu o fim do ficheiro; **<0**, indicando que houve um erro no acesso ao ficheiro. A escrita é realizada com a função *fwrite*, que também identifica o tamanho dos elementos e o número de elementos a escrever, com o mesmo significado. Esta função retorna o número de elementos escrito, ou **-1** se falhou a operação de escrita. Os descritores de ficheiro devem ser fechados com a operação *fclose*, para garantir que todo o conteúdo escrito é efetivamente copiado para o sistema de ficheiros – caso contrário pode-se perder dados.

```
#include <stdio.h>

gboolean copy_file(const char *from_filename, const char *to_filename) {
    FILE *f_in, *f_out;
    char buf[SND_BUFSIZE];
    int n, m;

    if ((f_in= fopen(from_filename, "r")) == NULL) {
        perror("error opening file for reading");
        return FALSE;
    }
    if ((f_out= fopen(to_filename, "w")) == NULL) {
        perror("error opening file for writing");
```

```

    fclose(f_in);
    return FALSE;
}
do {
    n= fread(buf, 1, SND_BUFLen, f_in);
    if (n>0) { // Leu n bytes
        // Foram lidos n bytes para o buffer 'buf'
        if ((m= fwrite(buf, 1, n, f_out)) != n) {
            // Falhou escrita no ficheiro
            break;
        } // else: Se n==0; atingiu o fim do ficheiro; Se n===-1 ocorreu erro na leitura
    } while (n>0);
    fclose(f_in);
    fclose(f_out);
    return (n==0) || (n==m);
}

```

2.1.10. Temporizadores fora do ambiente gráfico

Existem várias alternativas para realizar temporizadores, num programa em C/C++, fora de um ambiente gráfico que já suporte esta funcionalidade. Uma alternativa é usar o campo *timeout* da função `select`, descrita na página 12. Outra alternativa é usar a função `alarm` para agendar a geração do sinal SIGALRM com uma precisão de segundos. A função `alarm` é usada tanto para armar um temporizador (se *seconds* > 0) como para desarmar (se *seconds* == 0).

2.2. Aplicações com Interface gráfica Gtk+/Gnome

Numa aplicação em modo texto (sem interface gráfica) o programador escreve a rotina principal (`main`) controlando a sequência de ações que ocorrem no programa. Numa aplicação com uma interface gráfica, a aplicação é construída a partir de uma interface gráfica e do conjunto de funções que tratam os eventos gráficos, e caso existam, os restantes eventos assíncronos. Neste caso, a função principal (`main`) limita-se a arrancar com os vários objetos terminando com uma invocação à função `gtk_main()`, que fica em ciclo infinito à espera de interações gráficas, de temporizadores, ou de qualquer outro descritor de ficheiro. Internamente, o Gtk+ realiza esta rotina com a função `select`.

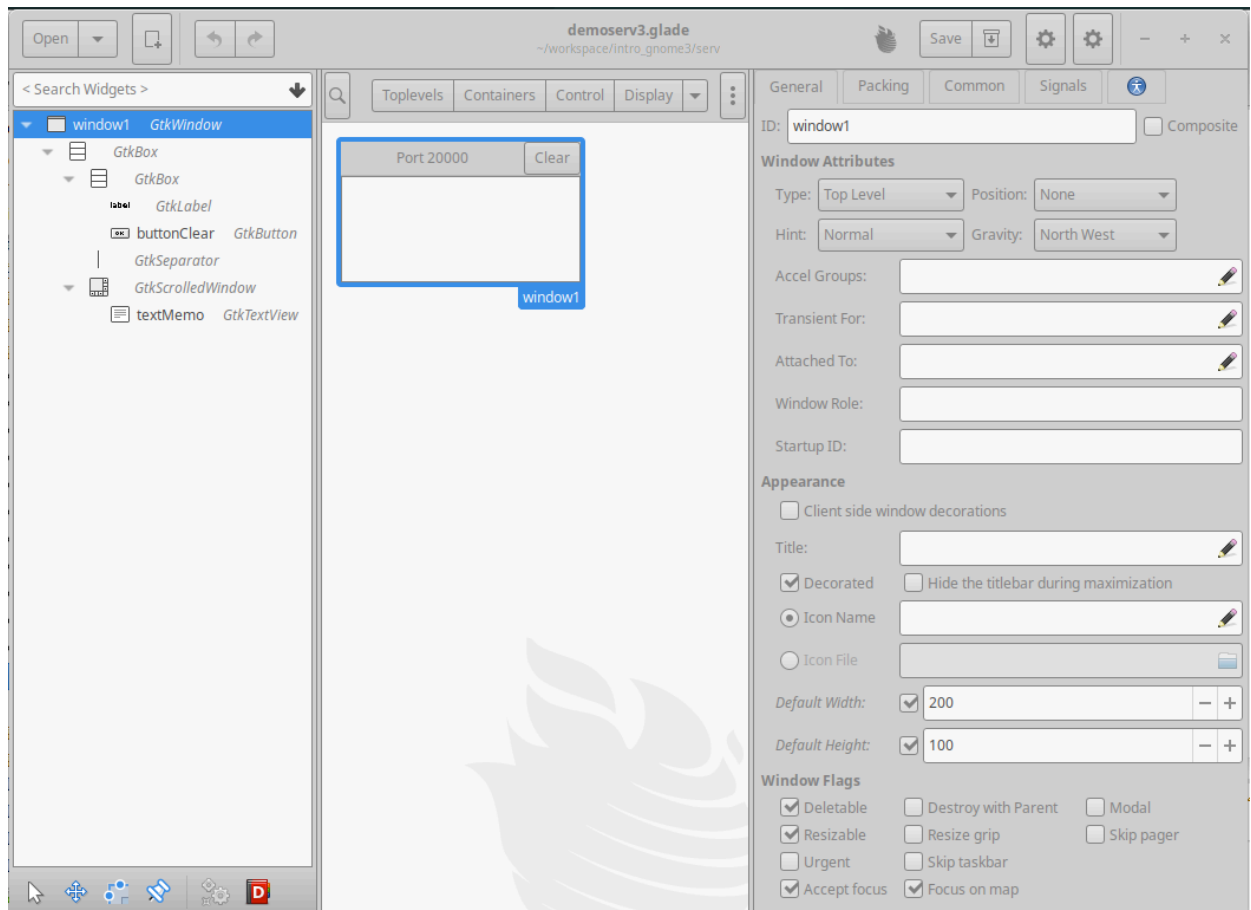
2.2.1. Editor de interfaces gráficas Glade-3


O editor de interfaces, **Glade-3** (comando `glade-3` (Fedora) ou `glade` (Ubuntu)) pode ser usado para desenhar a interface gráfica de aplicações para Gnome desenvolvidas utilizando várias linguagens de programação (C, C++, Python, etc). Esta aplicação está incluída nos pacotes da distribuição Fedora ou Ubuntu do Linux. Também existe noutras distribuições Linux, e mais recentemente, foi portada para outros sistemas operativos.

Ao contrário da abordagem usada na versão anterior (Glade-2), no Glade-3 não é gerado código 'C'; o editor gráfico limita-se a gerar um ficheiro XML que é carregado quando a aplicação arranca utilizando um objeto `GtkBuilder` GTK+ definido na biblioteca Gnome. O código 'C' é depois inteiramente gerado pelo utilizador, sendo neste documento sugerida uma metodologia para realizar o seu desenvolvimento.

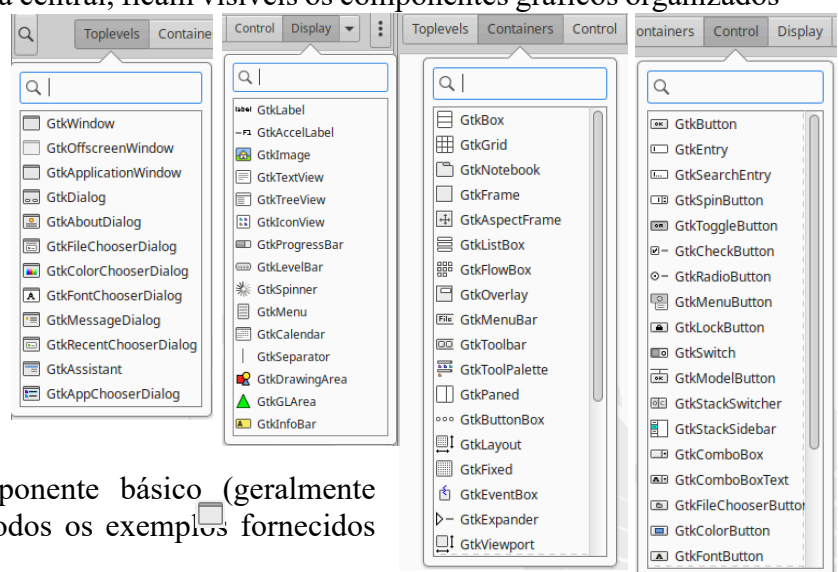
O desenvolvimento de um novo projeto inicia-se com a definição da interface gráfica utilizando o Glade-3. Quando se abre um novo projeto surge um ambiente integrado, representado abaixo, que é composto por três elementos principais: no centro, em baixo existe a janela que se está a desenvolver e em cima um conjunto de botões que permite aceder a quatro paletas de componentes que podem ser usados no desenho de janelas; à esquerda existe uma visão em árvore dos componentes (e.g. janelas) definidos, e à direita existe uma janela de edição de propriedades

do objeto selecionado. Para além disso, inclui os habituais botões de gravação/abertura de ficheiros de especificação de interfaces em ficheiros XML com a extensão `'.glade'`. A figura apresentada exemplifica a configuração da janela usada no programa de demonstração `demoserv`, apresentado na secção 3.6.1 (página 38).





Selecionando os botões na janela central, ficam visíveis os componentes gráficos organizados em quatro grupos. Pode encontrar a informação exhaustiva sobre todos os componentes (funções, callbacks, etc.) utilizando a aplicação *DevHelp* diretamente a partir do Glade-3, selecionando o icon . Para as aplicações que vão ser desenvolvidas na disciplina, vai ser usado apenas um subconjunto destes componentes que é sumariado de seguida².

Todas as interfaces são desenhadas a partir de um componente básico (geralmente `GtkWindow` ou `GtkDialog`). Todos os exemplos fornecidos









² Para quem quiser saber mais sobre o Glade-3 e os seus componentes sugere-se a leitura dos documentos tutoriais em <http://www.micahcarrick.com/gtk-glade-tutorial-part-1.html> e <http://glade.gnome.org/>.



neste documento foram desenvolvidos usando o componente `GtkWindow` ().

Por omissão, uma `GtkWindow` apenas suporta um componente gráfico no seu interior. Para poder ter vários componentes é necessário usar um ou mais componentes estruturantes que subdividam a janela em várias caixas. Alguns destes componentes ( ) permitem respetivamente: a divisão em colunas ou linhas da janela; a divisão da janela numa matriz; e a divisão em pastas. Outros permitem a disposição arbitrária na janela, mas não permitem lidar automaticamente com o redimensionamento das janelas.

Uma vez subdividida a janela, podem-se colocar em cada caixa os restantes componentes gráficos. Os componentes gráficos usados nos exemplos deste documento (da página `Control and Display`) foram:

	<code>GtkTextView</code>	Caixa editável com múltiplas linhas, geralmente usada dentro de um <code>GtkScrolledWindow</code>	
	<code>GtkLabel</code>	Títulos	
	<code>GtkEntry</code>	Caixa com texto editável	
	<code>GtkButton</code>	Botão	
	<code>GtkToggleButton</code>	Botão com estado	

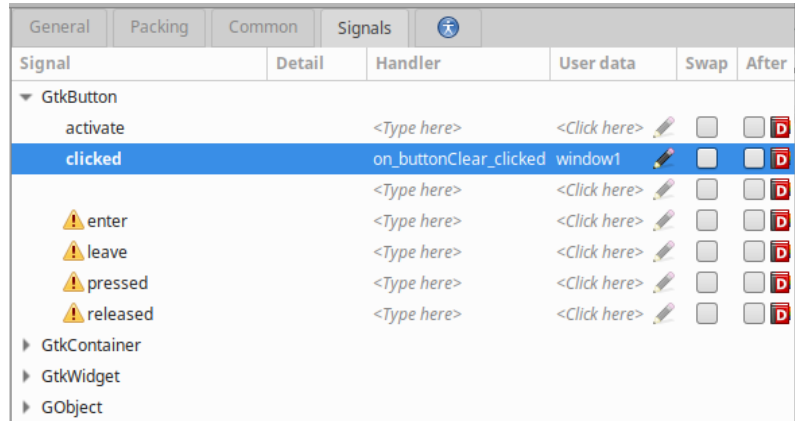
Foi ainda usado um componente gráfico mais complexo para realizar tabelas:



 `GtkTreeView` (também com `GtkScrolledWindow`) – Visualizador de dados em árvore, onde os dados são guardados numa lista do tipo `GtkListStore` , na pasta `Miscellaneous`. Para interligar os dados na lista com o visualizador são usados objetos derivados do tipo `GtkCellRenderer`, que mantêm os dados visualizados coerentes com os dados na base de dados, como está representado na figura ao lado, relativa ao programa de demonstração `democli`, apresentado na secção 3.6.2 (página 43). Pode encontrar um vídeo com um tutorial em <https://www.youtube.com/watch?v=nJloKxELPgI>.

▼ treeview1	<code>GtkTreeView</code>
treeview-selection1	<code>GtkTreeSelection</code> (i)
▼ treeviewcolumn1	<code>GtkTreeViewColumn</code>
cellrenderertext1	<code>GtkCellRendererText</code>
▼ treeviewcolumn2	<code>GtkTreeViewColumn</code>
cellrenderertext2	<code>GtkCellRendererText</code>

Tal como noutros editores integrados usados anteriormente, é possível editar as propriedades iniciais dos componentes gráficos usando a janela no lado direito. No caso representado, estão ilustradas algumas propriedades do objeto `window1` do tipo `GtkWindow`. Esta janela tem cinco páginas. A primeira (**General**) contém a definição do nome do objeto gráfico (`ID:`), e de outras propriedades específicas para cada objeto (e.g. a dimensão inicial da janela em `Default width` e `Default Height`). No caso de uma `GtkEntry` pode-se definir se é editável, se o texto é visível, o texto inicial, o comprimento máximo do texto, etc. A página (**Common**) controla aspetos gráficos, como por exemplo, se expande horizontalmente ou verticalmente, com o redimensionamento da janela. A página "**Packing**" controla a posição relativa do componente quando este se encontra numa linha ou numa coluna. No exemplo da figura, o componente "`buttonClear`" está na segunda posição da segunda "`GtkBox`", podendo-se mudar a posição relativa mudando o valor de "`Position:`".

A pasta **Signals** permite associar funções aos eventos suportados pelos vários componentes gráficos. Para cada componente podem-se associar várias funções a diferentes eventos (designados de "Signals"). O campo "Handler" representa o nome da função, que deve ser criado no código a desenvolver. O campo "User data" permite passar



um argumento na invocação da função com uma referência para um componente gráfico. No caso do exemplo é passado um ponteiro para a janela "textMemo", mas poderia ser qualquer outro componente gráfico. O protótipo da função pode ser obtido consultando o *DevHelp* (). O sinal  está associado a eventos não suportados por todas as versões de Gnome ou obsoletos.

O Glade-3 cria um ficheiro XML com a especificação dos componentes gráficos, com extensão ".glade" (e.g. exemplo.glade). O código C desenvolvido vai ter de iniciar a janela gráfica, de incluir todas as funções de tratamento de eventos especificadas no ficheiro XML, e de ter uma função `main` que fique a correr o ciclo principal do Gnome. Para facilitar a interação com os objetos gráficos recomenda-se que seja criado um ficheiro `gui.h`, onde se defina uma estrutura com apontadores para todos os elementos gráficos que vão ser acedidos no programa (e.g. `GtkEntry` com parâmetros de entrada ou saída, `GtkTreeView` com tabelas, etc.), com uma função que inicialize a janela, e com todas as outras funções usadas na interface gráfica.

```
#include <gtk/gtk.h>

/* store the widgets which may need to be accessed in a typedef struct */
typedef struct
{
    GtkWidget      *window;
    GtkEntry       *entryIP;
    ...
} WindowElements;

gboolean init_app (WindowElements *window, const char *filename);
```

A função `init_app` deverá ser específica para cada interface, pois deve inicializar todos os apontadores da estrutura criada anteriormente após carregar a especificação XML. Estes apontadores só podem ser obtidos durante a fase de criação da janela utilizando-se a função `gtk_builder_get_object` e o nome do objeto gráfico definido com o *Glade*. É usada a conversão explícita de tipos (e.g. `GTK_ENTRY`), que para além de fazer o "cast" do tipo também verifica se o objeto gráfico referenciado é compatível.

```
gboolean init_app (WindowElements *window, const char *filename)
{
    GtkBuilder      *builder;
    GError          *err=NULL;
    PangoFontDescription *font_desc;

    /* use GtkBuilder to build our interface from the XML file */
    builder = gtk_builder_new ();
    if (gtk_builder_add_from_file (builder, filename, NULL) == 0)
    {
        /* Error building interface
        return FALSE;
    }

    /* get the widgets which will be referenced in callbacks */
```

```

window->window = GTK_WIDGET (gtk_builder_get_object (builder, "window1"));
window->entryIP = GTK_ENTRY (gtk_builder_get_object (builder, "entryIP"));

/* connect signals, passing our TutorialTextEditor struct as user data */
gtk_builder_connect_signals (builder, window);

/* free memory used by GtkBuilder object */
g_object_unref (G_OBJECT (builder));

// Do other initializations ...

return TRUE;
}

```

Por fim, a função `main` deve iniciar a estrutura (após alocar previamente a memória), mostrar a janela, ficando bloqueada na função `gtk_main`, que realiza o ciclo principal onde o sistema gráfico processa os eventos. Esta função só é desbloqueada após ser invocada a função `gtk_main_quit`, que termina o programa.

```

int main (int argc, char *argv[])
{
    WindowElements      *editor;

    /* allocate the memory needed by our TutorialTextEditor struct */
    editor = g_slice_new (WindowElements);

    /* initialize GTK+ libraries */
    gtk_init (&argc, &argv);
    if (init_app (editor, "exemplo.glade") == FALSE) return 1; /* error loading UI */

    /* show the window */
    gtk_widget_show (editor->window);

    /* enter GTK+ main loop */
    gtk_main ();

    /* free memory we allocated for TutorialTextEditor struct */
    g_slice_free (WindowElements, editor);
    return 0;
}

```

O código desenvolvido é compilado utilizando o comando `pkg-config` para obter as diretorias e símbolos específicos para cada distribuição. Para compilar o ficheiro `main.c` e gerar o binário app o comando seria:

```
gcc -Wall -g -o app main.c `pkg-config --cflags --libs gtk+-3.0` -export-dynamic
```

No Ubuntu, o Glade-3 está disponível através na linha de comando "`glade [nome do projeto].glade`", ou através do ambiente integrado Eclipse. Vários ficheiros de documentação (incluindo o manual e o FAQ) estão disponíveis em *DevHelp*.

Nas secções 3.6 e 3.7 deste documento, a partir da página 38, são apresentados dois exemplos programas desenvolvidos utilizando o Glade-3.

2.2.2. Funções auxiliares

Para desenvolver uma aplicação em Gtk+/Gnome é necessário usar várias funções auxiliares para aceder aos objetos gráficos. Adicionalmente, existem funções para lidar com os descritores ativos no ciclo principal, para trabalhar com *strings*, listas, etc. A descrição deste conjunto de funções está disponível através da aplicação *DevHelp*, sendo, para além disso, fornecidos dois exemplos de programas que usam algumas das funcionalidades. Nesta secção são apresentadas algumas funções que lidam com aspetos mal documentados desta biblioteca.

2.2.2.1. Tipos de dados da biblioteca Gnome (glib) – lista (GList)

O Gnome redefine um conjunto de tipos básicos (int, bool, etc.) para tipos equivalentes com um nome com o prefixo (g): gint, gboolean, etc. Adicionalmente, o Gnome define vários tipos estruturados, incluindo **o tipo GList, que define uma lista**. Uma lista começa com um ponteiro a NULL (para GList) e pode ser manipulada com as seguintes funções:

```
GList *list= NULL; // A lista começa com um ponteiro a NULL
GList* g_list_append(GList *list, gpointer data); // Acrescenta 'data' ao fim da lista
GList* g_list_insert(GList *list, gpointer data, gint position); // Acrescenta 'data' na posição 'position'
GList* g_list_remove(GList *list, gpointer data); // Remove elemento
void g_list_free (GList *list); // Liberta lista, não liberta memória
// alocada nos membros da lista
guint g_list_length (GList *list); // comprimento da lista
GList* g_list_first (GList *list); // Devolve primeiro elemento da lista
GList* g_list_last(GList *list); // Devolve último membro da lista
GList *g_list_previous(list); // Retorna membro anterior ou NULL
GList *g_list_next(list); // Retorna membro seguinte ou NULL
GList* g_list_nth(GList *list, guint n); // Retorna o n-ésimo membro da lista
// Os dados são acedidos através do campo data: (Tipo)pt->data
```

2.2.2.2. Funções para manipular strings

A biblioteca Gnome duplica muitas das funções de <string.h>, como por exemplo, a função g_strdup. Caso exista a necessidade de criar strings complexas a partir de vários elementos, pode-se usar a função g_strdup_printf para alocar espaço e formatar uma string com uma sintaxe igual à função printf.

2.2.2.3. Aceder a objetos gráficos

O acesso a cada objeto gráfico é realizado através de funções de interface específicas. Por exemplo, para obter o texto dentro da caixa de texto entryIP referida anteriormente, poder-se-ia usar a seguinte função:

```
const char *textIP= gtk_entry_get_text(editor->entryIP);
```

A operação de escrita seria:

```
gtk_entry_set_text(editor->entryIP, "127.0.0.1")
```

2.2.2.4. Terminação da aplicação

Um programa apenas termina quando se invoca a operação gtk_main_quit(), provocando o fim do ciclo principal.

Por omissão, o fechar de todas as janelas de um programa não termina o executável. Para garantir que isso acontece é necessário associar uma função ao evento "delete_event" na janela principal que retorne FALSE. O conteúdo da função poderá ser:

```
gboolean
on_window1_delete_event (GtkWidget *widget,
                          GdkEvent *event,
                          gpointer user_data)
{
    /* Fechar todos os dados específicos da aplicação */ ...
    gtk_main_quit();
    return FALSE;
}
```

2.2.2.5. Eventos externos – sockets e pipes

O Gnome permite registar funções de tratamento de eventos de leitura, escrita ou exceções de sockets no ciclo principal através de um descritor de canal (tipo GIOChannel). As funções são registadas com a função g_io_add_watch, indicando-se o tipo de evento pretendido. Um exemplo de associação de uma rotina callback_dados aos eventos de leitura (G_IO_IN), de

escrita (G_IO_OUT) e de exceções (G_IO_NVAL, G_IO_ERR) para um descritor de *socket* sock seria:

```
GIOChannel *chan= NULL;           // Descritor do canal do socket
guint chan_id;                    // Número de canal
...
if ( (chan= g_io_channel_unix_new(sock)) == NULL) {
    g_print("Falhou criação de canal IO\n"); ...
}
if (! (chan_id= g_io_add_watch(chan,
    G_IO_IN|G_IO_OUT|G_IO_NVAL|G_IO_ERR, /* após eventos de leitura e erros */
    callback_dados /* função chamada */,
    NULL /* parâmetro recebido na função*/)) ) {
    g_print("Falhou ativação de recepção de dados\n"); ...
}
```

A função `callback_dados` deve ter a seguinte estrutura:

```
gboolean callback_dados (GIOChannel *source, GIOCondition condition, gpointer data)
{
    if (condition == G_IO_IN ) {
        /* Recebe dados ...*/
        return TRUE; /* a função continua ativa */
    } else if (condition == G_IO_OUT ) {
        /* Há espaço para continuar a escrever dados ...*/
        return TRUE; /* a função continua ativa */
    } else if ((condition == G_IO_NVAL) || (condition == G_IO_ERR)) {
        /* Trata erro ... */
        return FALSE; /* Deixa de receber evento */
    }
}
```

Pode desligar-se a associação da função ao evento utilizando a função `g_source_remove` com o número de canal como argumento.

```
/* Retira socket do ciclo principal do Gtk */
g_source_remove(chan_id);
/* Liberta canal */
g_io_channel_shutdown(chan, TRUE, NULL); // Termina ligação, deixando esvaziar buffer
g_io_channel_unref(chan);                // Decrementa número de utilizadores de canal;
// O canal é libertado quando atinge zero referências
```

2.2.2.6. Eventos externos – *timers*

O Gnome permite armar temporizadores que invocam periodicamente uma função. Um temporizador é armado usando a função `g_timeout_add`:

```
guint t_id;
t_id= g_timeout_add(atraso, /* Número de milissegundos */
    callback_timer, /* Função invocada */
    NULL); /* Argumento passado à função */
```

A função de tratamento do temporizador deve obedecer à seguinte assinatura:

```
gboolean callback_timer (gpointer data)
{
    // data - parâmetro definido em g_timeout_add
    return FALSE; // retira função do ciclo principal
ou return TRUE; // continua a chamar a função periodicamente
}
```

Pode-se cancelar um temporizador com a função `g_source_remove` usando o valor de `t_id` no argumento.

2.2.3. Utilização de *threads* em aplicações com interface gráfica

Para aceder a funções da biblioteca GLib a partir de threads concorrentes é necessário os mecanismos de gestão de paralelismo disponibilizados pela GDK (*Gnome Dev. Kit*), descrita em

<https://gtk.developpez.com/doc/en/gdk/gdk-Threads.html>, que permite correr o ambiente gráfico com uma *thread* dedicada e coordenar o acesso ao ambiente gráfico.

O primeiro método³ baseia-se na aquisição de um semáforo antes de aceder à interface gráfica quando a invocação é feita a partir de uma *thread* diferente, ou após receber um evento assíncrono. No caso de ações síncronas desencadeadas a partir da interface gráfica (e.g. premir um botão), estas já decorrem no contexto da *thread* gráfica. Qualquer tentativa de adquirir o semáforo leva ao bloqueio da aplicação (pois, já foi adquirido antes para processar o botão).

Na função `main` é necessário inicializar a *thread* gráfica com a instrução `gdk_threads_init`.

```
gdk_threads_init (); // Initialize the GLib threads
```

Todos os acessos às funções da biblioteca GLib e Gtk+3 devem ser feitos após bloquear o semáforo da GLib com a função `gdk_threads_enter`, devendo-se desbloquear o semáforo com a função `gdk_threads_leave` após a operação.

```
gdk_threads_enter (); // locks the GLib semaphore

// code with interaction with GLib and Gtk+3 data structures

gdk_threads_leave (); // unlocks the Glib semaphore
```

Este método baseado em semáforos foi recentemente considerado obsoleto e em alternativa foi proposto um método baseado no registo de uma função que é chamada pela *thread* da Glib para invocar os métodos das bibliotecas GLib e Gtk+3 e atualizar a interface gráfica. Desta forma, estas funções de *callback* são chamadas no contexto da *thread* gráfica. Para registar uma função de *callback* gráfica é usada a função `g_idle_add`.

```
g_idle_add ( GSourceFunc function, gpointer data);
```

De forma a poder passar mais de um argumento para a função de *callback*, é necessário definir uma estrutura com todos os dados necessário, alocar e inicializar uma variável dessa estrutura, e passar o apontador no parâmetro `data`. Na função de *callback* (`worker`, no exemplo seguinte), deve-se recuperar o apontador inicial e correr o código gráfico. No final, o valor retornado pela função determina se a *callback* se mantém ativa (TRUE) ou se é desligada (FALSE).

```
struct worker_arguments {
    // ... data elements to use in graphical update ...
};

gboolean worker(gpointer data) {
    struct worker_arguments *args = (struct worker_arguments *)data;

    // ... code with interaction with GLib and Gtk+3 data structures ...
    // It must be programmed as a machine state, updating everything since last invocation!

    free(args);
    // In the end the programmer may return TRUE or FALSE:
    return FALSE;    // Stop the callback
    return TRUE;     // Keep the callback alive
}
```

A atualização da interface gráfica é desencadeada após ativar a *callback* e aguardar que a função `worker` seja invocada – com a desvantagem de se perder o controlo sobre a ordem por que as funções de *callback* são chamadas.

```
...
struct worker_arguments *args = malloc (sizeof(struct worker_arguments));
args->... = ...; // Initialize the struct elements
g_idle_add(worker, args); // worker function will be called after some time
...
```

Neste trabalho vai ser usada a abordagem baseada na aquisição do semáforo antes de aceder

³ Tem mais informações em <https://gtk.developpez.com/doc/en/gdk/gdk-Threads.html>

às funções da biblioteca Gtk+3.

2.2.4. Utilização de subprocessos em aplicações com interface gráfica

Podem ser utilizados subprocessos em aplicações com interface gráfica, desde que **apenas um processo** escreva na interface gráfica. Após a operação *fork*, todos os processos partilham a mesma ligação ao servidor gráfico (X), que apenas aceitar comandos numerados sequencialmente. A solução para realizar esta integração é manter o processo pai como responsável pela interação gráfica, utilizando os mecanismos de IPC para suportar a comunicação entre o processo pai e os sub-processos filhos. Os *pipes* e *sockets* AF_UNIX, são descritos por descritores de ficheiros, podendo-se associar uma callback à sua utilização.

O código seguinte ilustra um excerto da função de tratamento da receção de dados do subprocesso. Pode-se usar o campo `ptr` para identificar o subprocesso emissor (no exemplo é passado o *process id* como argumento durante o registo da *callback*). Pode-se também obter o descritor do *pipe*, com o código representado.

```
/* Regista socket */
gboolean callback_pipe(GIOChannel *chan, GIOCondition cond, gpointer ptr) {
    int pid= (int)ptr; // PID do processo que mandou dados
    int p= g_io_channel_unix_get_fd(chan); // Obtém descritor de ficheiro do pipe
    // Ver estrutura da função callback_dados das páginas 24 e 41.
    // Cancela callback com 'return FALSE; // Mantém ativa com 'return TRUE;'
```

O código do processo filho é chamado na função de lançamento do subprocesso, que segue a estrutura apresentada anteriormente, na secção 2.1.7 e 2.1.8, e é exemplificada adiante, no exemplo 3.5. A grande modificação ocorre no código relativo ao processo pai, que não pode ficar bloqueado. Assim, deve registar a callback de processamento dos dados do subprocesso.

```
gboolean start_subprocesso( ... ) {
    int n; // pid de subprocesso
    int p[2]; // descritor de pipe (ou socket AF_UNIX)
    guint chan_id; // numero interno de canal Gtk+
    GIOChannel *chan; // estrutura de canal Gtk+

    pipe(p);
    n= fork(); // Inicia subprocesso
    ...
    if (n == 0) {
        /***** PROCESSO FILHO *****/
        close (p[0]); // Escreve para p[1]
        ...
        exit(0); // Termina sub-processo
        /***** Fim do PROCESSO FILHO *****/
    }
    /***** PROCESSO PAI *****/
    fprintf(stderr, "Arrancou filho leitura com pid %d\n", n);
    close(p[1]); // Pai usa p[0]
    if (!put_socket_in_mainloop(p[0], (void *)n/*passa o pid como parametro*/, &chan_id,
        &chan, callback_pipe)) {
        Log("falhou insercao de pipe no ciclo Gtk+\n");
        return FALSE;
    }
    ...
}
```

2.3. O ambiente integrado Eclipse para C/C++

O ambiente de desenvolvimento Eclipse permite desenvolver aplicações C/C++ de uma forma simplificada, a nível de edição e debug. O mecanismo de indexação permite-lhe mostrar onde é que uma função ou variável é declarada, facilitando muito a visualização de código existente. O *debugger* integrado permite depois visualizar os valores de uma variável

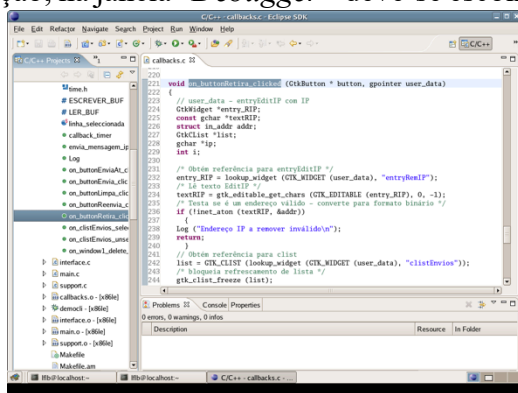
posicionando o rato sobre elas, ou introduzir pontos de paragem. O ambiente é especialmente indicado para C e C++.

Caso o Eclipse não reconheça o `glade-3` como o editor de ficheiros “*glade*”, para fazer edição integrada de ficheiros “*glade*”, pode associar-se a aplicação “`/usr/bin/glade`” à extensão “*glade*”, no menu: “*Window*”; “*Preferences*”; “*General*”; “*Editors*”; “*File Associations*”.

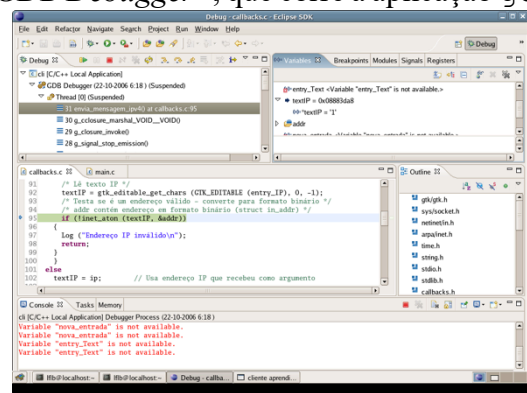
Um problema que por vezes ocorre em projetos grandes é a falta de memória. Por omissão, o eclipse arranca com 256 MBytes de memória, mas é possível aumentar esta memória, definindo um ficheiro de comandos para arrancar o eclipse, com o seguinte conteúdo:

```
/usr/bin/eclipse -vm [path para java] -vmargs -Xmx[memória (e.g. 512M)]
```

Para correr uma aplicação dentro do *debugger* é necessário acrescentar a aplicação ao menu de aplicações. No menu “*Run*”, escolhendo “*Debug ...*”, abre-se uma janela onde do lado esquerdo aparece uma lista de “*Configurations:*”. Nessa lista deve-se criar uma nova aplicação local “*C/C++ Local Application*”. Na janela “*Main*” deve escolher-se o projeto e o executável da aplicação; na janela “*Debugger*” deve-se escolher o “*GDB Debugger*”, que corre a aplicação `gdb`.



(a) Edição de ficheiros



(b) Debugging

Nas duas imagens anteriores ilustra-se o aspeto gráfico do Eclipse, em modo de edição e em modo de Debug. No primeiro caso (a) temos acesso à lista de ficheiros, e à lista de funções, variáveis e inclusões por ficheiro. Na janela “*Problems*”, tem-se uma lista de hiper ligações para os problemas identificados pelo ambiente no código. Na janela de *debugging* (b) pode-se visualizar a pilha de chamada de funções (*Debug*), e os valores das variáveis, depois de se atingir um ponto de paragem (*breakpoint*).

2.4. Configuração do Linux para correr aplicações *dual-stack* multicast

Os sistemas mais recentes Linux suportam IPv6 a nível do serviço *NetworkManager*, vulgarmente associado ao ícone de rede no canto superior direito do ecrã. Para ativar o suporte IPv6 basta aceder à aplicação gráfica de configuração de rede. Dentro da aplicação deve-se configurar o dispositivo de rede que estiver a usar (geralmente é o ‘`eth0`’, mas com o comando ‘`ifconfig`’ pode ver a lista de dispositivos), e entrar na edição de propriedades, selecionando a opção de suporte de IPv6, configurando o endereço IPv6 estaticamente, no caso de estar numa rede só IPv4. Sugere-se que use um endereço na gama 2001:690:1fff:bb::X/120, onde X pode ter um valor entre 1 e ffff. Caso pretenda testar a utilização da rede IPv6, pode consultar a página <https://wiki.ubuntu.com/IPv6> para saber como pode configurar túneis para a Internet IPv6.

A firewall do sistema pode também bloquear o funcionamento das aplicações. Nesse caso pode-se desativar temporariamente a firewall com os comandos “*iptables -F*” (IPv4) e “*ip6tables -F*” (IPv6), ou acrescentar regras para que os portos das aplicações sejam aceites.

3. EXEMPLOS DE APLICAÇÕES

Nesta secção são fornecidos cinco exemplos de aplicações cliente-servidor realizados com *sockets*. As secções 3.1 e 3.2 descrevem aplicações com *sockets* UDP realizadas sem uma interface gráfica, respetivamente para IPv4 e para IPv6. A secção 3.3 descreve uma aplicação com *sockets* TCP sem interface gráfica. A secção 3.4 descreve uma aplicação com threads POSIX. A secção 3.5 descreve uma aplicação com vários subprocessos. A secção 3.6 descreve o desenvolvimento da interface gráfica e a sua integração com *sockets* UDP. Finalmente, a secção 3.7 descreve as modificações ao exemplo da secção 3.6 para integrar *sockets* TCP. É fornecido um projeto eclipse com o código fonte contido nesta secção.

3.1. Cliente e Servidor UDP para IPv4 Multicast em modo texto

A programação da aplicação em modo texto resume-se à transcrição do código utilizando um editor de texto. Neste exemplo, o servidor arranca no porto 20000, associa-se ao endereço IPv4 Multicast "225.1.1.1", e fica bloqueado à espera de receber uma mensagem. A mensagem tanto pode ser recebida através do endereço IP Multicast como do endereço unicast da máquina local.

O código do "servv4.c" é:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

main (int argc, char *argv[])
{
    int sock, length;
    struct sockaddr_in name;
    char buf[1024];
    short int porto= 0; /* Por omissão o porto é atribuído pelo sistema */
    int reuse= 1;
    /* Multicast */
    struct ip_mreq imr;
    char loop = 1;
    /* receção */
    struct sockaddr_in fromaddr;
    int fromlen= sizeof(fromaddr);

    /* Cria o socket. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("Erro a abrir socket datagrama");
        exit(1);
    }
    if (argc == 2) { /* Introduziu-se o número de porto como parâmetro */
        porto= (short int)atoi(argv[1]);
    }
    /* Torna o IP do socket partilhável - permite que existam vários servidores associados ao
    mesmo porto na mesma máquina */
    if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *) &reuse, sizeof(reuse)) < 0) {
        perror("Falhou setsockopt SO_REUSEADDR");
        exit(-1);
    }
    /* Associa socket a porto */
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = htonl(INADDR_ANY); // IP local por omissão
    name.sin_port = htons(porto);
    if (bind(sock, (struct sockaddr *)&name, sizeof(name)) < 0) {
        perror("Falhou binding de socket datagrama");
        exit(1);
    }
    /* Configurações Multicast */
    if (!inet_aton("225.1.1.1", &imr.imr_multiaddr)) {
        perror("Falhou conversão de endereço multicast");
    }
}
```

```

    exit(1);
}
imr.imr_interface.s_addr = htonl(INADDR_ANY); /* Placa de rede por omissão */
/* Associa-se ao grupo */
if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *) &imr,
    sizeof(struct ip_mreq)) == -1) {
    perror("Falhou associação a grupo IP multicast");
    abort();
}
/* Configura socket para receber eco dos dados multicast enviados */
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));

/* Descobre o número de porto atribuído ao socket */
length = sizeof(name);
if (getsockname(sock, (struct sockaddr *)&name, &length)) {
    perror("Erro a obter número de porto");
    exit(1);
}
printf("O socket no endereço IP Multicast 225.1.1.1 tem o porto %d\n", htons(name.sin_port));
/* Lê uma mensagem do socket */
if (recvfrom(sock, buf, 1024, 0/* Por omissão fica bloqueado*/, (struct sockaddr *)&fromaddr,
    &fromlen) < 0)
    perror("Erro a receber pacote datagrama");
printf("Recebido de %s:%d -->%s\n", inet_ntoa(fromaddr.sin_addr),
    ntohs(fromaddr.sin_port), buf);
printf("O servidor desligou-se\n");
close(sock);
}

```

O código do cliente é comparativamente mais simples. O cliente limita-se a criar um *socket*, definir o IP e porto de destino e enviar a mensagem. Observe-se que, para além da definição do TTL enviado no pacote, nada varia no envio de um pacote para um endereço IPv4 multicast e para um endereço IPv4 unicast de uma máquina.

O código do "cliv4.c" é:

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

/* Aqui é enviado um datagrama para um receptor definido a partir da linha de comando */

#define DATA "Hello world!" /* Mensagem estática */

main (int argc, char *argv[])
{
    int sock;
    struct sockaddr_in name;
    struct hostent *hp;
    u_char ttl= 1; /* envia só para a rede local */

    /* Cria socket */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("Erro na abertura de socket datagrama");
        exit(1);
    }
    /* Constrói nome, do socket destinatário. Gethostbyname retorna uma estrutura que inclui o
    endereço IP do destino, funcionando com "pc-1" ou "10.1.55.1". Com a segunda classe de
    endereços também poderia ser usada a função inet_aton. O porto é obtido da linha de
    comandos */
    if (argc<=2) {
        fprintf(stderr, "Utilização: %s ip porto\n", argv[0]);
        exit(2);
    }
    hp = gethostbyname(argv[1]);
    if (hp == 0) {
        fprintf(stderr, "%s: endereço desconhecido\n", argv[1]);
        exit(2);
    }
}

```

```

bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
name.sin_family = AF_INET;
name.sin_port = htons(atoi(argv[2])); /* converte para formato rede */
/* Configura socket para só enviar dados multicast para a rede local */
if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, (char *) &tctl,
    sizeof(ttl)) < 0) {
    perror("Falhou setsockopt IP_MULTICAST_TTL");
}
/* Envia mensagem */
if (sendto(sock, DATA, strlen(DATA)+1 /* para enviar '\0'*/, 0,
    (struct sockaddr *)&name, sizeof(name)) < 0)
    perror("Erro no envio de datagrama");
close(sock);
}

```

Falta, por fim, criar um ficheiro com o nome "Makefile" para automatizar a criação dos executáveis. Neste ficheiro descreve-se na primeira linha o objetivo a concretizar (`all: cli serv`) – a criação de dois executáveis. Nas duas linhas seguintes indica-se quais os ficheiros usados para criar cada executável (`cli: cli.c` – `cli` é criado a partir de `cli.c`) e a linha de comando para criar o executável (`gcc -g -o cli cli.c`) precedida de uma tabulação.

O texto do ficheiro "Makefile" é:

```

all: cliv4 servv4

cliv4: cliv4.c
    gcc -g -o cliv4 cliv4.c

servv4: servv4.c
    gcc -g -o servv4 servv4.c

```

3.2. Cliente e Servidor UDP para IPv6 Multicast em modo texto

A programação da aplicação IPv6 é muito semelhante à aplicação IPv4, excetuando a utilização de funções específicas para IPv6. Utilizando os endereços `::ffff:a.b.c.d` esta aplicação pode comunicar com a aplicação desenvolvida em 3.1, dizendo-se por isso, que funciona em modo de pilha dupla (*dual stack*). Neste exemplo, o servidor arranca no porto 20000, associa-se ao endereço IP multicast `ff18:10:33::1` caso sejam omitidos os dois parâmetros, e fica bloqueado à espera de receber uma mensagem. A mensagem tanto pode ser recebida através do endereço IP multicast como do endereço unicast da máquina local.

O código do "servv6.c" é:

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

/* Retorna uma string com o endereço IPv6 */
char *addr_ipv6(struct in6_addr *addr) {
    static char buf[100];
    inet_ntop(AF_INET6, addr, buf, sizeof(buf));
    return buf;
}

main (int argc, char *argv[])
{
    int                sock, length;
    struct sockaddr_in6 name;
    short int         porto= 0;
    int                reuse= 1;
    /* Multicast */
    char*              addr_mult= "FF18:10:33::1"; // endereço por omissão
    struct ipv6_mreq    imr;

```

```

char                loop = 1;
/* recepção */
char                buf[1024];
struct sockaddr_in6 fromaddr;
int                 fromlen= sizeof(fromaddr);

/* Cria o socket. */
sock = socket(AF_INET6, SOCK_DGRAM, 0);
if (sock < 0) {
    perror("Erro a abrir socket datagrama ipv6");
    exit(1);
}
if (argc >= 2) {
    /* Pode-se introduzir o número de porto como parâmetro */
    porto= (short int)atoi(argv[1]);
}
if (argc == 3) {
    /* Segundo parametro é o endereço IPv6 multicast */
    addr_mult= argv[2];
}
/* Torna o IP do socket partilhável - permite que existam vários servidores
 * associados ao mesmo porto na mesma máquina */
if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *) &reuse,
    sizeof(reuse)) < 0) {
    perror("Falhou setsockopt SO_REUSEADDR");
    exit(-1);
}
/* Associa socket a porto */
name.sin6_family = AF_INET6;
name.sin6_flowinfo= 0;
name.sin6_port = htons(porto);
name.sin6_addr = in6addr_any; /* IPv6 local por defeito */
if (bind(sock, (struct sockaddr *)&name, sizeof(name)) < 0) {
    perror("Falhou binding de socket datagrama");
    exit(1);
}

/* Configuracoes Multicast */
if (!inet_pton(AF_INET6, addr_mult, &imr.ipv6mr_multiaddr)) {
    perror("Falhou conversão de endereço multicast");
    exit(1);
}
imr.ipv6mr_interface = 0; /* Interface 0 */
/* Associa-se ao grupo */
if (setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, (char *) &imr,
    sizeof(imr)) == -1) {
    perror("Falhou associação a grupo IPv6 multicast");
    abort();
}
/* Configura socket para receber eco dos dados multicast enviados */
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop, sizeof(loop));

/* Descobre o número de porto atribuído ao socket */
length = sizeof(name);
if (getsockname(sock, (struct sockaddr *)&name, &length)) {
    perror("Erro a obter número de porto");
    exit(1);
}
printf("O socket no endereço IP Multicast %s tem o porto #%d\n",
    addr_ipv6(&imr.ipv6mr_multiaddr), htons(name.sin6_port));
/* Lê uma mensagem do socket */
if (recvfrom(sock, buf, 1024, 0/* Por defeito fica bloqueado*/, (struct sockaddr *)&fromaddr,
    &fromlen) < 0)
    perror("Erro a receber pacote datagrama");
printf("Recebido de %s##d -->%s\n", addr_ipv6(&fromaddr.sin6_addr), ntohs(fromaddr.sin6_port),
    buf);
printf("O servidor desligou-se\n");
close(sock);
}

```

O código do cliente IPv6 é comparativamente mais simples. O cliente limita-se a criar um *socket*, definir o IP e porto de destino e enviar a mensagem.

O código do "cliv6.c" é:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

/* Aqui é enviado um datagrama para um recetor definido a partir da linha de comando */

#define DATA "Olá mundo ..." /* Mensagem estática */

char *addr_ipv6(struct in6_addr *addr) {
    static char buf[100];
    inet_ntop(AF_INET6, addr, buf, sizeof(buf));
    return buf;
}

main (int argc, char *argv[])
{
    int          sock;
    struct sockaddr_in6 name;
    struct hostent *hp;

    /* Cria socket */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("Erro na abertura de socket datagrama ipv6");
        exit(1);
    }
    /*
     * Constrói nome do socket destinatário. gethostbyname2 retorna uma estrutura
     * que inclui o endereço IPv6 do destino, funcionando com "pc-1" ou "2001:690:2005:10:33::1".
     * Com o segundo endereço poderia ser usada a função inet_pton.
     * O porto é obtido da linha de comando
     */
    if (argc <= 2) {
        fprintf(stderr, "Utilização: %s ip porto\n", argv[0]);
        exit(2);
    }
    hp = gethostbyname2(argv[1], AF_INET6);
    if (hp == 0) {
        fprintf(stderr, "%s: endereço desconhecido\n", argv[1]);
        exit(2);
    }
    bcopy(hp->h_addr, &name.sin6_addr, hp->h_length);
    name.sin6_family = AF_INET6;
    name.sin6_port = htons(atoi(argv[2])); /* converte para formato rede */
    /* Envia mensagem */
    fprintf(stderr, "Enviando pacote para %s ; %d\n", addr_ipv6(&name.sin6_addr),
        (int)ntohs(name.sin6_port));
    if (sendto(sock, DATA, strlen(DATA)+1 /* para enviar '\0'*/, 0,
        (struct sockaddr *)&name, sizeof(name)) < 0)
        perror("Erro no envio de datagrama");
    fprintf(stderr, "Fim do cliente\n");
    close(sock);
}
```

A criação do ficheiro Makefile é deixada para exercício.

Sugere-se como um **exercício** adicional, uma modificação do programa para limitar o tempo máximo de espera por pacotes a 10 segundos.

3.3. Cliente e Servidor TCP para IPv6 em modo texto

A programação da aplicação com *sockets* TCP é um pouco mais complicada por ser orientada à ligação. Comparando com o cliente do exemplo anterior, a diferença está no estabelecimento e terminação da ligação. Todas as restantes inicializações são idênticas. Este exemplo ilustra como

se pode enviar uma mensagem com duas componentes – enviando uma componente de cada vez.

O código do cliente "clitcpv6.c" é:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

#define DATA "Half a league, half a league ..."

char *addr_ipv6(struct in6_addr *addr) {
    static char buf[100];
    inet_ntop(AF_INET6, addr, buf, sizeof(buf));
    return buf;
}

/*
 * This program creates a socket and initiates a connection
 * with the socket given in the command line. One message
 * is sent over the connection and then the socket is
 * closed, ending the connection.
 */

main (int argc, char *argv[])
{
    int                sock, msg_len;
    struct sockaddr_in6 server;
    struct hostent *hp, *gethostbyname();

    /* Create socket */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }
    /* Connect socket using name specified by command line. */
    hp = gethostbyname2(argv[1], AF_INET6);
    if (hp == 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    server.sin6_family = AF_INET6;
    server.sin6_flowinfo = 0;
    server.sin6_port = htons(atoi(argv[2]));
    bcopy(hp->h_addr, &server.sin6_addr, hp->h_length);

    if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("connecting stream socket");
        exit(1);
    }
    // Envia o comprimento dos dados, seguido dos dados
    msg_len= htonl(strlen(DATA)+1); // Em formato rede, com '\0'
    if (write (sock, &msg_len, sizeof(msg_len)) < 0)
        perror ("writing on stream socket");
    if (write(sock, DATA, strlen(DATA)+1) < 0)
        perror("writing on stream socket");
    close(sock);
}
```

O código do servidor é bastante mais complicado porque vão coexistir várias ligações em paralelo no servidor. Neste exemplo, o servidor associa-se ao porto 20000 e prepara-se para receber ligações (com a função `listen`). A partir daí, fica num ciclo bloqueado à espera de receber ligações (com a função `accept`), escrevendo o conteúdo da primeira mensagem recebida de cada ligação, e fechando-a em seguida. Utilizando subprocessos, ou a função `select`, teria sido possível receber novas ligações e dados em paralelo a partir das várias ligações.

O código do servidor "servtcpv6.c" é:

```
#include <sys/types.h>
```

```

#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

char *addr_ipv6(struct in6_addr *addr) { ... ver cliente ... }

main ()
{
    int                sock, msgsock, length;
    struct sockaddr_in6 server;
    char               buf[1024];

    /* Create socket on which to read. */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }
    /* Create name with wildcards. */
    server.sin6_family = AF_INET6;
    server.sin6_addr = in6addr_any;
    server.sin6_port = 0;
    server.sin6_flowinfo = 0;
    if (bind(sock, (struct sockaddr *)&server, sizeof(server))) {
        perror("binding stream socket");
        exit(1);
    }
    /* Find assigned port value and print it out. */
    length = sizeof(server);
    if (getsockname(sock, (struct sockaddr *)&server, &length)) {
        perror("getting socket name");
        exit(1);
    }
    if (server.sin6_family != AF_INET6) {
        perror("Invalid family");
        exit(1);
    }
    printf("Socket has ip %s and port %d\n", addr_ipv6(&server.sin6_addr),
           ntohs(server.sin6_port));

    /* Start accepting connections */
    listen (sock, 5);
    while (1) {
        msgsock = accept(sock, (struct sockaddr *)&server, &length);
        if (msgsock == -1)
            perror("accept");
        else {
            int n, msg_len;
            printf("Connection from %s - %d\n", addr_ipv6(&server.sin6_addr),
                   ntohs(server.sin6_port));
            bzero(buf, sizeof(buf));
            if (read (msgsock, &msg_len, sizeof(msg_len)) < 0) {
                perror ("receiving stream data");
                close (msgsock);
                continue;
            }
            msg_len= ntohs(msg_len);
            if ((n= read (msgsock, buf, msg_len)) < 0)
                perror ("receiving stream data");
            else
                printf ("--> (%d/%d bytes) %s\n", n, msg_len, buf);
            close (msgsock);
        }
    }
}

```

3.4. Programa com *threads* em modo texto

Este exemplo ilustra a criação e terminação de tarefas (*threads* POSIX) para correr atividades concorrentes.

O código do programa "demopthread.c" é:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

/** Source code executed in the thread */
void *thread_routine ( void *ptr )
{
    char *ret_value = malloc(10);           // Allocate the return value
    strcpy(ret_value, "hello world");       // set the return value contents

    // ptr was initialized in pthread_create argument
    int *state= (int *)ptr;
    /* function code */
    fprintf (stderr, "thread started\n");
    *state= 2;                             // Changing the state
    fprintf (stderr, "thread is sleeping for 5 seconds\n");
    sleep (5);                             // Sleep 2 seconds
    fprintf (stderr, "thread stopped\n");

    pthread_exit((void*)ret_value); // Finishes the thread and returns a string
}

Int main (int argc, char *argv[])
{
    pthread_t tid;           // thread id
    int state= 1;
    char *return_value;

    fprintf (stderr, "main: The value of state before running the thread is %d\n", state);
    /* Start the thread */
    if (pthread_create(&tid, NULL, thread_routine, (void *) &state)) {
        fprintf(stderr, "main: error starting thread\n");
        return 1;
    }
    fprintf (stderr, "main: started thread with id %u\n", (unsigned)tid);
    /* Wait for the thread to finish */
    if(pthread_join(tid, (void *)&return_value)) {
        fprintf(stderr, "main: error joining thread\n");
        return 2;
    }
    fprintf(stderr, "main: Thread %u returned : %s\n", (unsigned)tid, return_value);
    free(return_value); // Free the return value memory allocated in the thread
    fprintf (stderr, "main: The value of state after the thread end is %d\n", state);
    return 0;
}
```

3.5. Programa com subprocessos em modo texto

Este exemplo ilustra a criação e terminação de um subprocesso e a utilização de um *pipe* para enviar uma mensagem com dois elementos do processo filho para o processo pai. A função *reaper* analisa o motivo porque o processo filho termina.

O código do programa "demofork.c" é:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <signal.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

void reaper(int sig)           // callback para tratar SIGCHLD
{
    sigset_t set, oldset;
```

```

pid_t pid;
union wait status;

// bloqueia outros sinais SIGCHLD
sigemptyset(&set);
sigaddset(&set, SIGCHLD);
sigprocmask(SIG_BLOCK, &set, &oldset);
//
fprintf(stderr, "reaper\n");
while ((pid= wait3(&status, WNOHANG, 0)) > 0) { // Enquanto houver filhos zombie
    if (WIFEXITED(status))
        fprintf(stderr, "child process (pid= %d) ended with exit(%d)\n",
            (int)pid, (int)WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        fprintf(stderr, "child process (pid= %d) ended with kill(%d)\n",
            (int)pid, (int)WTERMSIG(status));
    else
        fprintf(stderr, "child process (pid= %d) ended\n", (int)pid);
    continue;
}
// Reinstalar tratamento de signal
signal(SIGCHLD, reaper);
//Desbloquear signal SIGCHLD
sigemptyset(&set);
sigaddset(&set, SIGCHLD);
sigprocmask(SIG_UNBLOCK, &set, &oldset);
//
fprintf(stderr, "reaper ended\n");
}

int main (int argc, char *argv[])
{
    int p[2]; // descritor de pipe
    char *buf;
    int n, m;

    signal(SIGCHLD, reaper); // arma callback para sinal SIGCHLD
    if (pipe(p)) { perror(pipe); exit(-1); } // Cria par de pipes locais
    n= fork(); // Cria sub-processo
    if (n == -1) {
        perror("fork failed");
        exit(-1);
    }
    if (n == 0) {
/*****
// Código do processo filho
    char *msg= "Ola";
    fprintf(stderr, "filho (pid = %d)\n", (int)getpid());
    close(p[0]); // p[0] é usado pelo pai
    sleep(2); // Dorme 2 segundos
    // Envia mensagem ao pai
    write (p[1], &msg_len, sizeof(msg_len)); // comprimento da mensagem
    write (p[1], msg, strlen (msg) + 1); // dados da mensagem
    close(p[1]);
    fprintf(stderr, "morreu filho\n");
    _exit(1); // Termina processo filho
*****/
    }
    // Código do processo pai
    fprintf(stderr, "pai: arrancou filho com pid %d\n", n);
    close(p[1]); // p[1] é usado pelo filho
    do {
        m = read (p[0], &msg_len, sizeof(msg_len));
    } while ((m == -1) && (errno == EINTR)); // Repete se for interrompido por sinal
    buf= (char *)malloc(msg_len);
    do {
        m = read (p[0], buf, msg_len); // Espera por mensagem do filho
    } while ((m == -1) && (errno == EINTR)); // Repete se for interrompido por sinal
    fprintf (stderr, "Child process %d sent %d/%d bytes: '%s'\n", n, m, msg_len, buf);
}

```

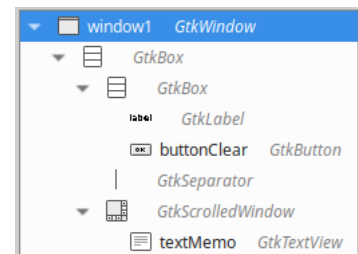
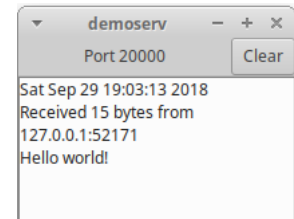
3.6. Cliente e Servidor UDP com interface gráfico

Esta primeira aplicação gráfica distribuída é constituída por dois executáveis, criados a partir de dois ficheiros Glade-3. O servidor cria um *socket* UDP e associa-o ao porto 20000, ficando a partir daí à espera de mensagens de um cliente. O conteúdo das mensagens recebidas é escrito numa caixa *GtkTextView*, que pode ser limpa recorrendo-se a um botão. O cliente permite enviar o conteúdo de uma caixa de texto (*GtkEntry*) para um *socket* remoto. Para além do envio imediato, é suportado o envio diferido após um número de milissegundos configurável. O cliente regista todos os envios efetuados, podendo reenviar uma mensagem para um endereço IP anterior.

3.6.1. Servidor

O servidor deve ser programado numa diretoria diferente do cliente. A primeira parte da programação do servidor consiste no desenho da interface gráfica do servidor utilizando a aplicação Glade-3. Deve-se criar o projeto "demoserv3.glade".

O servidor é desenvolvido a partir de uma janela *GtkWindow* (*window1*), subdividindo-se a janela em três linhas utilizando uma *GtkBox* e a primeira linha em duas colunas com outra *GtkBox*. Alternativamente, pode usar-se o componente que permite colar componentes gráficos em posições arbitrárias. Em seguida é introduzido um *GtkLabel* com "Port 20000", um botão (*buttonClear*) e um *GtkTextView* (*textMemo*) com *scrollbar*, devendo-se mudar o nome de acordo com a árvore de componentes representada à direita.



O desenvolvimento do código inicia-se com a criação do ficheiro "gui.h" com a definição da estrutura de apontadores para componentes gráficos e a declaração das funções globais usadas.

```
#include <gtk/gtk.h>

typedef struct
{
    GtkWidget          *window;
    GtkTextView         *text_view;
} ServWindowElements;

/**** Global variables ****/
// Pointer to the structure with all the elements of the GUI
extern ServWindowElements *main_window;

/**** Global methods ****/
// Handles 'Clear' button - clears the text box and the table with sent messages
void on_buttonClear_clicked (GtkButton * button, gpointer user_data);
// Writes a message in the screen and in the command line
void Log (const gchar * str);
// We call error_message() any time we want to display an error message to the
// user. It will both show an error dialog and log the error to the terminal window.
void error_message (const gchar *message);
// Initializes all the windows and graphical callbacks of the application
gboolean init_app (ServWindowElements *window, const char *glade_file);
```

A função de inicialização inicializa a janela e em caso de erro, abre uma janela com o erro usando a função *error_message* definida no módulo *gui_g3.c*, fornecido com o enunciado:

```
gboolean init_app (ServWindowElements *window, const char *glade_file)
{
    GtkBuilder          *builder;
    GError               *err=NULL;
```

```

PangoFontDescription    *font_desc;

/* use GtkBuilder to build our interface from the XML file */
builder = gtk_builder_new ();
if (gtk_builder_add_from_file (builder, "demoserv3.glade", &err) == 0)
{
    error_message (err->message);
    g_error_free (err);
    return FALSE;
}

/* get the widgets which will be referenced in callbacks */
window->window = GTK_WIDGET (gtk_builder_get_object (builder, "window1"));
window->text_view = GTK_TEXT_VIEW (gtk_builder_get_object (builder, "textMemo"));

/* connect signals, passing our TutorialTextEditor struct as user data */
gtk_builder_connect_signals (builder, window);

/* free memory used by GtkBuilder object */
g_object_unref (G_OBJECT (builder));

return TRUE;
}

```

De seguida, vão-se programar as funções de tratamento dos eventos gráficos. Vai-se associar uma *callback* ao sinal "**delete**" da janela principal "**window1**", de forma a parar o executável quando se fecha a janela:

```

gboolean on_window1_delete_event      (GtkWidget      *widget,
                                       GdkEvent        *event,
                                       gpointer          user_data)
{
    gtk_main_quit();    // Fecha ciclo principal do Gtk+
    return FALSE;      // Termina o programa
}

```

Também se vai associar uma rotina ao evento "**clicked**" do botão "**button1**", de forma a limpar o conteúdo da caixa "textMemo". Deve ser acrescentado o seguinte código:

```

void on_buttonLimpar_clicked           (GtkButton        *button,
                                       gpointer          user_data)
{
    /* Limpa textMemo */
    GtkTextBuffer *textbuf;
    GtkTextIter tbegin, tend;

    /* Obtém referência para modelo com dados */
    textbuf = GTK_TEXT_BUFFER (gtk_text_view_get_buffer (main_window->text_view));
    /* define 2 limites e apaga janela */
    gtk_text_buffer_get_iter_at_offset (textbuf, &tbegin, 0);
    gtk_text_buffer_get_iter_at_offset (textbuf, &tend, -1);
    gtk_text_buffer_delete (textbuf, &tbegin, &tend);
}

```

Para facilitar a escrita de mensagens, é criada uma função auxiliar Log:

```

void Log(const gchar *str)
{
    GtkTextBuffer *textbuf;
    GtkTextIter tend;

    assert ((str != NULL) && (main_window->text_view != NULL));
    textbuf = GTK_TEXT_BUFFER (gtk_text_view_get_buffer (main_window->text_view));
    //Gets a reference to the last position in textbox and adds the message.
    gtk_text_buffer_get_iter_at_offset (textbuf, &tend, -1);
    gtk_text_buffer_insert (textbuf, &tend, g_strdup (str), strlen (str));
}

```

Para facilitar o desenvolvimento da comunicação com o *socket* UDP é fornecido um módulo (`sock.c` e `sock.h`), com um conjunto de funções auxiliares para lidar com *sockets*. As funções e variáveis disponibilizadas neste módulo estão definidas no ficheiro "`sock.h`". Pode consultar os ficheiros para ver os detalhes da realização das funções.

```

// Variables with local IP addresses
extern struct in_addr local_ipv4; // Local IPv4 address
extern gboolean valid_local_ipv4; // TRUE if it obtained a valid local IPv4 address
extern struct in6_addr local_ipv6; // Local IPv6 address
extern gboolean valid_local_ipv6; // TRUE if it obtained a valid local IPv4 address

/* Macro used to read data from a buffer */
/* pt - reading pointer */
/* var - variable pointer */
/* n - number of bytes to read */
#define READ_BUF(pt, var, n) bcopy(pt, var, n); pt+= n

/* Macro used to write data */
/* pt - writing pointer */
/* var - variable pointer */
/* n - number of bytes to write */
#define WRITE_BUF(pt, var, n) bcopy(var, pt, n); pt+= n

gboolean init_local_ipv4(struct in_addr *ip); // Return local IPv4 address
gboolean init_local_ipv6(struct in6_addr *ip); // Return local IPv6 address
// Returns TRUE if 'ip_str' is a local address (it only supports one address per host)
gboolean is_local_ip(const char *ip_str);

void set_local_IP(); // Sets variables with local IP addresses

// Reads an IPv6 Multicast address
gboolean get_IPv6(const gchar *textIP, struct in6_addr *addrv6);
// Reads an IPv4 Multicast address
gboolean get_IPv4(const gchar *textIP, struct in_addr *addrv4);

char *addr_ipv4(struct in_addr *addr); // Returns a string with an IPv4 address value
char *addr_ipv6(struct in6_addr *addr); // Returns a string with an IPv6 address value

// Initializes an IPv4 socket
int init_socket_ipv4(int dom, int porto, gboolean partilhado);
// Initializes an IPv6 socket
int init_socket_ipv6(int dom, int porto, gboolean partilhado);

int get_portnumber(int s); // Returns the port number associated to a socket

// Reads data from IPv4, returns the number of byte read or -1 and the address and port of the
// sender
int read_dados_ipv4(int sock, char *buf, int n, struct in_addr *ip,
    short unsigned int *porto);
// Reads data from IPv6, returns the number of byte read or -1 and the address and port of the
// sender
int read_dados_ipv6(int sock, char *buf, int n, struct in6_addr *ip,
    short unsigned int *porto);

// Associates a callback function to a socket sock with the channel chan, and passing the
// parameter pt during each callback
gboolean put_socket_in_mainloop(int sock, void *pt, guint *chan_id, GIOChannel **chan,
    GIOCondition cond,
    gboolean (*callback) (GIOChannel *, GIOCondition, gpointer));
// Cancels the callback association
void remove_socket_from_mainloop(int sock, int chan_id, GIOChannel *chan);

// Closes the socket
void close_socket(int sock);

```

A criação do *socket* e o registo da função de tratamento do *socket* no ciclo principal é feito na função `main`, garantindo-se que a partir do momento que o executável arranca está pronto para receber mensagens. O texto do ficheiro `main.c` fica então

```

#define GLADE_FILE      "demoserv3.glade"

/* Public variables */
int sock = -1; // socket descriptor
GIOChannel *chan = NULL; // socket's IO channel descriptor
guint chan_id; // IO Channel number
// Pointer to the structure with all the elements of the GUI
ServWindowElements *main_window; // Has pointers to all elements of main window

int main (int argc, char *argv[]) {

```



```

/* allocate the memory needed by our ServWindowElements struct */
main_window = g_slice_new (ServWindowElements);

/* initialize glib threads */
gdk_threads_init ();
/* initialize GTK+ libraries */
gtk_init (&argc, &argv);

if (init_app (main_window, GLADE_FILE) == FALSE) return 1; /* error loading UI */

/* Socket initialization */
if ((sock = init_socket_ipv4 (SOCK_DGRAM, 20000, FALSE)) == -1)
    return 1;
if (!put_socket_in_mainloop (sock, main_window, &chan_id, &chan, G_IO_IN,
    callback_dados))
    return 2;

gtk_widget_show (main_window->window);
gdk_threads_enter ();      // get GTK thread lock
gtk_main ();              // Launches Gtk main loop - loops forever until the end of the program
gdk_threads_leave ();      // release GTK thread lock

/* free memory we allocated for ServWindowElements struct */
g_slice_free (ServWindowElements, main_window);

return 0;
}

```

Para terminar a programação do servidor falta apenas programar a rotina que recebe os dados do *socket*. A rotina de leitura recorre à macro **READ_BUF** para ler campo a campo da mensagem recebida. A macro lê *n* bytes de um *buffer* para o endereço *var* e incrementa o ponteiro de leitura *pt*. No caso do código, lê 2 bytes para a variável *m* e avança o apontador *pt* para o terceiro byte da mensagem. A rotina de tratamento dos eventos do *socket* tem o seguinte código (no ficheiro `callbacks.c`):

```

gboolean callback_dados (GIOChannel *source, GIOCondition condition, gpointer data)
{
    static char write_buf[1024];
    static char buf[MAX_MESSAGE_LENGTH];      // buffer for reading data
    struct in_addr ip;
    short unsigned int porto;
    int n;

    if (condition == G_IO_IN)
    {
        /* Read data from the socket */
        n = read_data_ipv4 (sock, buf, MAX_MESSAGE_LENGTH, &ip, &porto);
        if (n <= 0)
        {
            Log ("Read from socket failed\n");
            return TRUE;      // Keeps waiting for more data
        } else { // n > 0
            time_t tbuf;
            short unsigned int m;
            char *pt;
            /* Writes date and sender of the packet */
            time (&tbuf);      // Gets current date
            sprintf (write_buf, "%sReceived %d bytes from %s:%hu\n",
                ctime (&tbuf), n, inet_ntoa (ip), porto);
            Log (write_buf);
            /* Read the message fields */
            pt = buf;
            READ_BUF (pt, &m, sizeof(m));      // Reads short (2 bytes) to m and moves pointer
                                                // pt points to the 3rd byte
            m = ntohs (m);      // Converts the number to host format
            if (m != n - 2)
            {
                sprintf (write_buf, "Invalid 'length' field (%d != %d)\n", m, n - 2);
                Log (write_buf);
                return TRUE;      // Keeps waiting for more data
            }
            /* Writes data to the memo box - assumes that it ends with '\0' */

```

```

        Log (pt); // pt points to the first byte of the string
        Log ("\n");
        return TRUE;      // Keeps waiting for more data
    }
}
else if ((condition == G_IO_NVAL) || (condition == G_IO_ERR))
{
    Log ("Detected socket error\n");
    remove_socket_from_mainloop (sock, chan_id, chan);
    chan = NULL;
    close_socket (sock);
    sock = -1;
    /* Stops the application */
    gtk_main_quit ();
    return FALSE;      // Removes socket's callback from main cycle
} else {
    assert (0);      // Must never reach this line - aborts application with a core dump
    return FALSE;      // Removes socket's callback from main cycle
}
}
}

```

Observe-se que a mensagem é composta por dois octetos com o comprimento e pelo conteúdo da string. Falta apenas declarar o valor do comprimento máximo da mensagem (MAX_MESSAGE_LENGTH) e a assinatura das funções criadas e das variáveis globais acrescentando o seguinte texto ao ficheiro de definições "callbacks.h":

```

#include <gtk/gtk.h>

#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE (!FALSE)
#endif

// Maximum message length
#define MAX_MESSAGE_LENGTH    5000

/**** Global variables ****/

// Variables declared and initialized in 'main.c'
extern int sock;                // socket descriptor
extern GIOChannel *chan;        // socket's IO channel descriptor
extern guint chan_id;           // IO Channel number

/**** Global functions ****/

/* Callback function that handles reading events from the socket.
 * It returns TRUE to keep the callback active, and FALSE to disable the callback */
gboolean callback_dados (GIOChannel * source, GIOCondition condition, gpointer data);

// GUI callback function called when the main window is closed
gboolean on_window1_delete_event (GtkWidget* widget, GdkEvent* event,
    gpointer user_data);

```

Para automatizar a compilação do programa foi criado um ficheiro Makefile, com as instruções para compilar todos os módulos e a lista de dependências:

```

APP_NAME= demoserv
GNOME_INCLUDES= `pkg-config --cflags --libs gtk+-3.0`
CFLAGS= -Wall -Wno-deprecated-declarations -g

all: $(APP_NAME)

clean:
    rm -f $(APP_NAME) *.o *.xml

demoserv: main.c sock.o gui_g3.o callbacks.o gui.h sock.h callbacks.h
    gcc $(CFLAGS) -o $(APP_NAME) main.c sock.o gui_g3.o callbacks.o $(GNOME_INCLUDES) -export-dynamic
sock.o: sock.c sock.h gui.h
    gcc $(CFLAGS) -c $(GNOME_INCLUDES) sock.c -export-dynamic
gui_g3.o: gui_g3.c gui.h

```

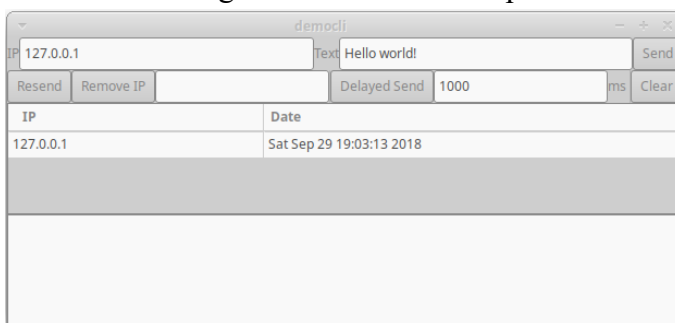
```
gcc $(CFLAGS) -c $(GNOME_INCLUDES) gui_g3.c -export-dynamic
callbacks.o: callbacks.c callbacks.h sock.h
gcc $(CFLAGS) -c $(GNOME_INCLUDES) callbacks.c -export-dynamic
```

Para compilar a aplicação basta correr o comando "make". O executável deve ser corrido na mesma diretoria onde está o ficheiro *democli.glade*, ou o nome do ficheiro deve incluir o caminho completo.

3.6.2. Cliente

O cliente é criado numa diretoria nova. A primeira parte da programação do cliente é novamente o desenho da interface gráfica do servidor com a aplicação Glade-3, no projeto "democli3.glade".

A interface gráfica é realizada a partir de uma

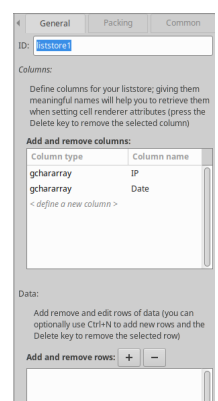
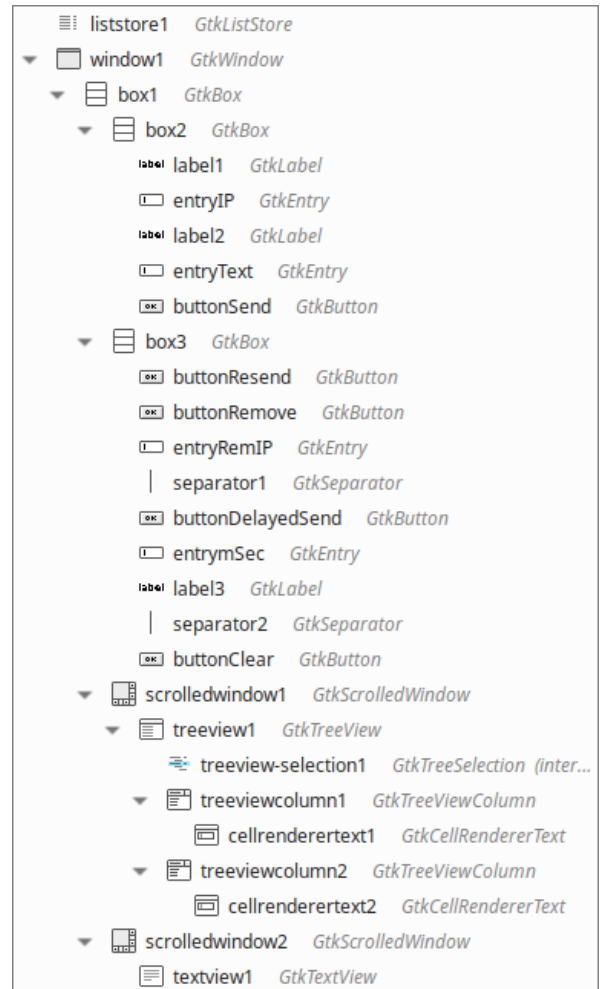


janela `GtkWindow` (`window1`) que é dividida em quatro linhas (`box1`). A primeira linha encontra-se dividida em 5 colunas, respetivamente: uma `GtkLabel` (inicializada com "IP"), uma `GtkEntry` ("entryIP" com o texto "127.0.0.1"); uma `GtkLabel` (inicializada com "Text"); uma `GtkEntry` ("entryText" com o texto por omissão "Hello world!"); e um `GtkButton` ("buttonSend" com o texto "Send").

A segunda linha encontra-se dividida em 9 colunas, respetivamente: um `GtkButton` ("buttonResend" com o texto "Resend"); um `GtkButton` ("buttonRemove" com o texto "Remove IP"); uma `GtkEntry` ("entryRemIP"); um separador vertical (`vseparator1`); um `GtkButton` ("buttonDelayedSend" com o texto "Delayed Send"); uma `GtkEntry` ("entrymSec" com o texto "1000"); um `GtkLabel` (`label3` inicializada com "ms"); um separador; e um `GtkButton` ("buttonClear" com o texto "Clear"). A penúltima linha contém uma `GtkTreeView` ("treeview1") com duas colunas e a última linha contém uma `GtkTextView` (`textview1`), ambas dentro de uma `GtkScrolledWindow`.

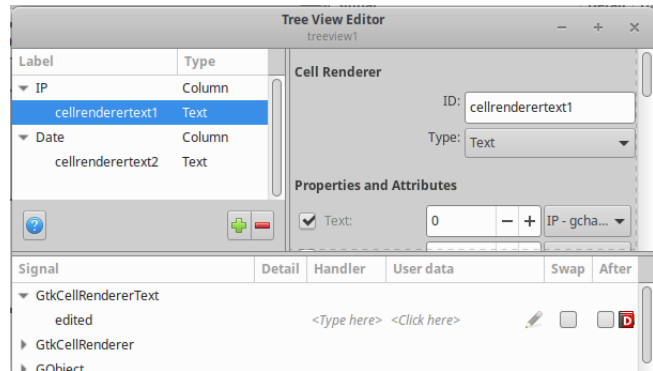
Para obter o aspeto ilustrado é necessário configurar a altura e largura de cada elemento da janela, usando a janela de edição de propriedades.

Os dados da tabela são guardados na `liststore1`, criada juntamente com o `GtkTextView`. No Glade é necessário usar os editores para definir dois campos na `liststore1` do tipo `GCHARARRAY` (i.e. string) respetivamente com os nomes IP e Date, tal



como está representado à direita.

De seguida, é necessário editar a `treeview1`, premindo-se o botão direito do rato e seleccionando a opção “Edit...”. Entra-se então no editor de *Tree View*, que vai ser usado para acrescentar e configurar as duas colunas (com “add column”). Numa fase posterior, para cada coluna, devem ser associados os visualizadores de texto, com “add child text”. É neste passo, representado na figura ao lado, que se define o que vai ser apresentado em cada coluna. No caso, define-se no campo “Text” que na coluna IP se visualiza o campo “IP-gchararray” do objeto `liststore1` associado. Desta forma, qualquer modificação que se faça na lista é atualizada automaticamente na interface gráfica.



O desenvolvimento do código inicia-se com a criação do ficheiro “gui.h” com a definição da estrutura de apontadores para componentes gráficos e a declaração das funções globais usadas.

```
/* store the widgets which may need to be accessed in a typedef struct */
typedef struct
{
    GtkWidget          *window1;
    GtkEntry           *entryIP;
    GtkEntry           *entryText;
    GtkEntry           *entryRemIP;
    GtkEntry           *entrymSec;
    GtkTextView        *textView1;
    GtkTreeView        *treeview1;
    GtkListStore       *liststore1;
} CliWindowElements;

/**** Global variables ****/

// Pointer to the structure with all the elements of the GUI
extern CliWindowElements *main window; // Pointers to all elements of main window
```

Novamente, a função de inicialização inicializa a janela e a estrutura, e modifica a fonte usada na caixa de texto `textView`:

```
gboolean
init_app (CliWindowElements *window, const char *glade_file)
{
    GtkBuilder          *builder;
    GError              *err=NULL;
    PangoFontDescription *font_desc;

    /* use GtkBuilder to build our interface from the XML file */
    builder = gtk_builder_new ();
    if (gtk_builder_add_from_file (builder, glade_file, &err) == 0)
    {
        error_message (err->message);
        g_error_free (err);
        return FALSE;
    }

    /* get the widgets which will be referenced in callbacks */
    window->window1 = GTK_WIDGET (gtk_builder_get_object (builder,
        "window1"));
    window->entryIP = GTK_ENTRY (gtk_builder_get_object (builder,
        "entryIP"));
    window->entryText = GTK_ENTRY (gtk_builder_get_object (builder,
```

```

        "entryText"));
window->entryRemIP = GTK_ENTRY (gtk_builder_get_object (builder,
"entryRemIP"));
window->entrymSec = GTK_ENTRY (gtk_builder_get_object (builder,
"entrymSec"));
window->textview1 = GTK_TEXT_VIEW (gtk_builder_get_object (builder,
"textview1"));
window->treeview1 = GTK_TREE_VIEW (gtk_builder_get_object (builder,
"treeview1"));
window->liststore1 = GTK_LIST_STORE(gtk_builder_get_object (builder,
"liststore1"));
/* connect signals, passing our WindowElements struct as user data */
gtk_builder_connect_signals (builder, window);

/* free memory used by GtkBuilder object */
g_object_unref (G_OBJECT (builder));

/* set the text view font */
font_desc = pango_font_description_from_string ("monospace 10");
gtk_widget_modify_font (GTK_WIDGET(window->textview1), font_desc);
pango_font_description_free (font_desc);

return TRUE;
}

```

Para facilitar a escrita de mensagens na caixa de diálogo, vai usar-se novamente a função Log, apresentada na página 39. Note-se que é necessário fazer uma pequena modificação, pois o nome da caixa de texto no cliente é "textview1" em vez de "textMemo".

Uma vez que o envio de mensagens pode ser feito a partir de várias funções, é criada uma função que realiza o envio do conteúdo de "entryText" para um endereço IP definido pelo parâmetro ip, ou se o parâmetro for NULL, pelo conteúdo de "entryIP". A assinatura da função deve ser acrescentada ao ficheiro "callbacks.h":

```
void send_message_ipv4 ( const gchar * ip );
```

Para facilitar a escrita num buffer é usada a macro WRITE_BUF, apresentada anteriormente no ficheiro "sock.h", no cliente, que copia n bytes a partir do endereço var para pt, incrementando pt. O código da função é programado no ficheiro "callbacks.c", enviando-se o comprimento da mensagem antes da mensagem:

```

// Function used to send a message with 'entryText' contents to the address in 'entryIP'
void send_message_ipv4 ( const gchar * ip )
{
    const gchar *textText;
    const gchar *textIP;
    struct in_addr addr;

    if (ip == NULL)
    {
        /* Reads IP address from entryIP box */
        textIP = gtk_editable_get_chars (GTK_EDITABLE (main_window->entryIP), 0, -1);
        /* Tests if it is a valid address converting it into binary format:
         *      addr contains the binary address (format struct in_addr) */
        if (!inet_aton (textIP, &addr))
        {
            Log ("Invalid IP address\n");
            return;
        }
    } else
        textIP = ip;          // Uses the address received as argument

    /* Read text string to send */
    textText = gtk_editable_get_chars (GTK_EDITABLE (main_window->entryText), 0, -1);
    /* Tests text */
    if ((textText == NULL) || !strlen (textText))
    {
        Log ("Empty text\n");
        return;
    }
}

```

```

struct sockaddr_in name;
static char buf[MAX_MESSAGE_LENGTH];          // buffer to write message
char *pt = buf;
short unsigned int len;
struct hostent *hp;

/* Creates message */
len= htons(strlen(textText)+1);/* Length (with '\0') in network format */
WRITE_BUF(pt, &len, sizeof(len)); // Adds length field to the message
WRITE_BUF(pt, textText, strlen(textText)+1); // Adds text
// pt points to the byte after the message - the number of bytes written is pt-buf
/* Defines destination address */
hp = gethostbyname(textIP);
if (hp == 0) {
    perror("Invalid destination address");
    return;
}
// Prepares struct sockaddr_in variable 'name', with destination data
bcopy(hp->h_addr, &name.sin_addr, hp->h_length); // define IP
name.sin_port = htons(20000); // define Port
name.sin_family = AF_INET; // define IPv4
/* Sends message */
if (sendto(sock, buf, pt-buf /* message length */, 0,
    (struct sockaddr *)&name, sizeof(name)) != pt-buf) {
    perror("Error sending datagram");
    Log("Error sending datagram\n");
}
name.sin_family= AF_INET;

/* Writes message into sending tableview */
GtkListStore *list= main_window->liststore1;
GtkTreeIter iter;
/* Gets local time */
time_t tbuf;
time(&tbuf);
char *time_buf= strdup(ctime (&tbuf));
time_buf[strlen(time_buf)-1]= '\0';
// Adds entry to the data store associated to the tableview
gtk_list_store_append(list, &iter);
gtk_list_store_set(list, &iter, 0, textIP, 1, time_buf, -1);
// Frees temporary memory
free(time_buf);
}

```

Passa-se de seguida à programação dos vários eventos gráficos. O evento **"clicked"** do botão **"buttonSend"** deve ser associado a uma função, com o código seguinte:

```

void on_buttonSend_clicked (GtkButton * button, gpointer user_data)
{
    send_message_ipv4 (NULL);
}

```

O evento **"clicked"** do botão **"buttonDelayedSend"** deve ser associado a uma função com o código representado abaixo. Esta função envia a mensagem com um atraso igual ao número de milissegundos indicado na caixa "entrymSec" usando um temporizador. Repare-se que a função `callback_timer` retorna `FALSE` para parar o timer; se retornasse `TRUE` ficava a ser chamada periodicamente.

```

// Timer callback used to implement delayed message sending
gboolean callback_timer (gpointer data)
{
    send_message_ipv4 (NULL);
    return FALSE;          // turns timer off after the first time
}

// GUI callback function called when 'Delayed Send' button is clicked
void on_buttonDelayedSend_clicked (GtkButton * button, gpointer user_data)
{
    // user_data - not used
    const gchar *textDelay;
    guint Delay = 0;
}

```

```

char *pt;

/* Gets text from entry box */
textDelay = gtk_editable_get_chars (GTK_EDITABLE (main_window->entrymSec), 0, -1);
/* tests if text is valid */
if ((textDelay == NULL) || (strlen (textDelay) == 0)) {
    Log ("Undefined number of mseconds\n");
    return;
}
/* Converts to integer */
Delay = strtoul (textDelay, &pt, 10);
if ((pt == NULL) || (*pt)) {
    Log ("Invalid number of mseconds\n");
    return;
}
/* Delays sending message - starts timer */
g_timeout_add (Delay, callback_timer, user_data);
}

```

Sempre que uma mensagem é enviada, acrescentou-se uma linha à tabela com o IP e a data de envio. Pretende-se que o utilizador possa seleccionar uma linha da tabela "tableview1" e reenviar uma mensagem para o IP nessa linha. A rotina associada ao evento "clicked" do botão "buttonResend" obtém a linha seleccionada e envia uma mensagem:

```

void on_buttonResend_clicked (GtkButton * button, gpointer user_data) {
    GtkTreeSelection *selection;
    GtkTreeModel *model;
    GtkTreeIter iter;
    gchar *ip;

    selection= gtk_tree_view_get_selection(main_window->treeview1);
    if (gtk_tree_selection_get_selected(selection, &model, &iter)) {
        gtk_tree_model_get (model, &iter, 0, &ip, -1);
        g_print ("Selected ip is: %s\n", ip);
    } else {
        Log ("No line selected\n");
        return;
    }
    send_message_ipv4 (ip); // Resend message
    g_free(ip);
}

```

A callback associada ao evento "**clicked**" do botão "**buttonRemove**" foi realizada com uma função auxiliar (foreach_func) chamada com gtk_tree_model_foreach para todos elementos da lista, de forma a devolver uma lista de todas as linhas com o endereço contido na variável remove_ip local ao módulo. Esta variável é previamente preenchida com o valor da caixa "entryRemIP". De seguida, apaga todas as linhas da lista.

```

// Module variable used within function foreach_func, to compare the address value
static const gchar *remove_ip= NULL;

// Callback function called by 'gtk_tree_model_foreach' for all members of table
// It is used to remove multiple rows in one go, returned in the list 'rowref_list'
gboolean
foreach_func (GtkTreeModel *model, GtkTreePath *path, GtkTreeIter *iter, GList **rowref_list)
{
    const gchar *ip;
    if (remove_ip == NULL)
        return TRUE; // Stop walking the store
    g_assert ( rowref_list != NULL );
    gtk_tree_model_get (model, iter, 0, &ip, -1);
    if ( !strcmp(ip, remove_ip) ) {
        GtkTreeRowReference *rowref;
        rowref = gtk_tree_row_reference_new(model, path);
        *rowref_list = g_list_append(*rowref_list, rowref);
    }
    return FALSE; /* do not stop walking the store, call us with next row */
}

// GUI callback function called when 'Remove IP' button is clicked
void on_buttonRemove_clicked (GtkButton * button, gpointer user_data)

```

```

{
    const gchar *textRIP;
    struct in_addr addr;

    /* Reads text in 'RemoveIP' edit box */
    textRIP = gtk_editable_get_chars (GTK_EDITABLE (main_window->entryRemIP), 0, -1);
    /* Tests if it is a valid address - converts it to binary format */
    if (!inet_aton (textRIP, &addr))
    {
        Log ("Invalid IP address in Remove\n");
        return;
    }
    remove_ip= textRIP;

    GList *rr_list = NULL;    /* list of GtkTreeRowReferences to remove */
    GList *node;
    gtk_tree_model_foreach(GTK_TREE_MODEL(main_window->liststore1), (GtkTreeModelForeachFunc)
        foreach_func, &rr_list);
    for ( node = rr_list; node != NULL; node = node->next ) {
        GtkTreePath *path;
        path = gtk_tree_row_reference_get_path((GtkTreeRowReference*)node->data);
        if (path) {
            GtkTreeIter iter;
            if (gtk_tree_model_get_iter(GTK_TREE_MODEL(main_window->liststore1), &iter, path)) {
                gtk_list_store_remove(main_window->liststore1, &iter);
            }
            gtk_tree_path_free (path);
        }
    }
    g_list_foreach(rr_list, (GFunc) gtk_tree_row_reference_free, NULL);
    g_list_free(rr_list);

    remove_ip= NULL;
}

```

A rotina associada ao evento "**clicked**" do botão "**buttonClear**" limpa o conteúdo de "liststore1" (logo da tabela) e de "textview1":

```

void on_buttonClear_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuf;
    GtkTextIter tbegin, tend;

    // Clear table with messages sent
    gtk_list_store_clear(main_window->liststore1);

    // Clear TextView
    textbuf = GTK_TEXT_BUFFER (gtk_text_view_get_buffer (main_window->textview1));
    gtk_text_buffer_get_iter_at_offset (textbuf, &tbegin, 0);
    gtk_text_buffer_get_iter_at_offset (textbuf, &tend, -1);
    gtk_text_buffer_delete (textbuf, &tbegin, &tend);
}

```

Para garantir que a aplicação termina que se fecha a janela deve-se associar uma rotina ao evento "**delete_event**" da janela principal "**window1**":

```

gboolean on_window1_delete_event(GtkWidget *widget, GdkEvent *event, gpointer user_data)
{
    gtk_main_quit();    // Fecha ciclo principal do Gtk
    return FALSE;
}

```

Falta modificar o ficheiro "main.c" de maneira a memorizar a janela principal na variável main_window (de forma semelhante à descrita na página 40) e a iniciar o *socket*. Por fim, falta preparar o ficheiro Makefile de forma semelhante ao exemplo anterior.

3.6.3. Exercícios

- 1) Modifique o código do cliente e do servidor de maneira a passarem a trabalhar com endereços IPv4 multicast. **Sugestão:** Analise o código apresentado na secção 3.1.

- 2) Modifique o código do cliente e do servidor de maneira a passarem a trabalhar com endereços IPv6 unicast. **Sugestão:** Analise o código apresentado na secção 3.2.
- 3) Modifique o código do cliente e do servidor de maneira a passarem a trabalhar com endereços IPv6 multicast. **Sugestão:** Analise o código apresentado na secção 3.2.

3.7. Cliente e Servidor TCP com interface gráfico

Esta segunda aplicação gráfica distribuída usa a mesma interface gráfica e a maior parte do código da primeira aplicação, apresentada na secção 3.5. Apenas modifica as funções estritamente necessárias para suportar o serviço orientado à ligação.

3.7.1. Servidor

Num socket TCP, as mensagens são recebidas num socket criado para a ligação. Desta forma, são necessários dois passos: no primeiro passo recebe-se a ligação no socket servidor (que aceita ligações) e cria-se um socket de dados; no segundo passo recebe-se a mensagem no socket de dados. Assim, torna-se necessário usar uma função de callback para cada fase.

A criação do *socket* e o registo da função de tratamento do *socket* no ciclo principal são feitos na função *main* e são muito semelhantes ao apresentado na secção 3.5.1. Apenas se muda a inicialização do socket e a associação do *handler*, que agora é responsável por receber ligações:

```
int main (int argc, char *argv[]) {
...
    /* Socket initialization */
    if ((sock = init_socket_ipv4 (SOCK_STREAM, 20000, FALSE)) == -1)
        return 1;
    listen(sock, 1); // Set socket to receive connections
    if (!put_socket_in_mainloop (sock, main_window, &chan_id, &chan, G_IO_IN,
        callback_connection))
        return 2;
...
}
```

A função *callback_connection* associada ao tratamento da receção de ligações no *socket* *sock* tem o código em baixo (no ficheiro *callbacks.c*). O evento *G_IO_IN* está associado à disponibilidade de uma nova ligação. Após criar a nova ligação de dados (*msgsock*), é associada a callback de dados *callback_data* ao evento *G_IO_IN* para tratar a receção de dados. Note-se que podem ser criados vários sockets, ficando o sistema com uma callback ativa para cada socket de dados.

```
gboolean callback_connection (GIOChannel *source, GIOCondition condition, gpointer data) {
    static char write_buf[1024];
    struct sockaddr_in server;
    int msgsock;
    unsigned int length= sizeof(server);
    GIOChannel *dchan = NULL; // socket's IO channel descriptor
    guint dchan_id;           // IO Channel number

    if (condition & G_IO_IN) {
        time_t tbuf;
        time (&tbuf);           // Gets current date
        /* Accept the connection and create a new socket */
        msgsock = accept (sock, (struct sockaddr *) &server, &length);
        if (msgsock == -1) {
            perror("accepting connection");
            return TRUE;
        }
        sprintf(write_buf, "%sConnection from %s - %hu in %d\n", ctime (&tbuf),
            addr_ipv4 (&server.sin_addr), ntohs (server.sin_port), msgsock);
        Log(write_buf);
        /* Prepare new callback for reading incoming data */
    }
```

```

    if (!put_socket_in_mainloop (msgsock, main_window, &dchan_id, &dchan, G_IO_IN,
        callback_data)) {
        Log("Failed to set data callback\n");
        close(msgsock);
    }
    // Wait for the G_IO_IN event!
    return TRUE;

} else if ((condition & G_IO_NVAL) || (condition & G_IO_ERR)) {
    Log ("Detected server socket error\n");
    // It should free the GIO device before removing it
    return FALSE;    // Removes socket's callback from main cycle
}
}

```

A rotina de tratamento dos eventos dos *sockets* de dados tem o código seguinte (no ficheiro `callbacks.c`). A função `g_io_channel_unix_get_fd` é usada para obter o descritor do socket com dados. De seguida, os dados são lidos diretamente do socket, libertando-se o canal quando a ligação termina.

```

gboolean callback_data (GIOChannel * source, GIOCondition condition, gpointer data) {
    static char write_buf[1024];
    static char buf[MAX_MESSAGE_LENGTH];    // buffer for reading data
    int n;
    int s = g_io_channel_unix_get_fd(source); // Get the socket file descriptor

    if (condition & G_IO_IN) {
        /* Read data from the socket */
        n = recv(s, buf, MAX_MESSAGE_LENGTH, 0);
        if (n < 0) {
            perror("read failed");
            Log ("Read from socket failed\n");
            free_gio_channel(source);
            return FALSE;    // Removes socket's callback from main cycle
        }
        else if (n == 0) {
            /* Reached end of connection */
            sprintf(write_buf, "Connection %d closed\n", s);
            Log(write_buf);
            free_gio_channel(source);
            return FALSE;    // Removes socket's callback from main cycle
        }
        else {
            time_t tbuf;
            /* Writes date and sender of the packet */
            time (&tbuf);    // Gets current date
            sprintf (write_buf, "%sReceived %d bytes from socket %d:\n", ctime (&tbuf), n, s);
            Log (write_buf);
            Log(buf);    // Write the message received
            Log("\n");
            return TRUE; // Keeps waiting for more data
        }
    }
    else if ((condition & G_IO_NVAL) || (condition & G_IO_ERR)) {
        Log ("Detected socket error\n");
        remove_socket_from_mainloop (sock, chan_id, chan);
        chan = NULL;
        close_socket (sock);
        sock = -1;
        /* Stops the application */
        gtk_main_quit ();
        return FALSE;    // Removes socket's callback from main cycle
    }
}

```

3.7.2. Cliente

Tal como no servidor, o cliente usa a interface gráfica e todo o código apresentado na secção 3.5.2. Apenas modifica a função `send_message_ipv4` que envia a mensagem, que agora necessita de abrir a ligação, enviar a mensagem e fechar a ligação. O novo código da função é

```

void send_message_ipv4 ( const gchar * ip )
{
    const gchar *textText;
    const gchar *textIP;
    struct in_addr addr;
    int n;

...
// Read ip to textIP; and textText with the message
...

    struct sockaddr_in name;
    struct hostent *hp;

    /* Defines destination address */
    hp = gethostbyname(textIP);
    if (hp == 0) {
        perror("Invalid destination address");
        return;
    }
    // Prepares struct sockaddr_in variable 'name', with destination data
    bcopy(hp->h_addr, &name.sin_addr, hp->h_length); // define IP
    name.sin_port = htons(20000); // define Port
    name.sin_family = AF_INET; // define IPv4

    /* Socket initialization */
    if ((sock = init_socket_ipv4 (SOCK_STREAM, 0, FALSE)) == -1) {
        perror("socket creation");
        return;
    }

    /* Connect to remote host */
    if (connect (sock, (struct sockaddr *) &name, sizeof (name)) < 0) {
        perror ("connecting stream socket");
        close(sock);
        return;
    }

    /* Send message */
    n= write (sock, textText, strlen (textText) + 1);
    if (n < 0)
        perror ("writing on stream socket");

    /* Remember that for large files n may be shorter than the message length: */
    /*     if n is shorter, the remaining bytes must be stored to be sent later */
    /*     when a G_IO_OUT event is received, signaling that there is space to write */
    close (sock);

...
// Write message to the sending tableview
}

```

Observe-se que esta abordagem apenas funciona com um socket no modo bloqueante, onde a operação de escrita bloqueia o programa até estar concluída. Quando se usa um socket no modo não bloqueante é necessário controlar o envio de dados através de uma callback associada ao evento `G_IO_OUT`. A escrita é realizada após receber o evento e até o socket deixar de aceitar a totalidade dos bytes escritos; nessas condições é necessário comparar o número de bytes escritos no socket com o número lido, e reenviar posteriormente os dados que não foram enviados.

3.7.3. Exercícios

- 1) Modifique o código do cliente e do servidor de maneira a passarem a trabalhar com endereços IPv6 unicast. **Sugestão:** Analise o código apresentado na secção 3.3.
- 2) O exemplo usa sockets bloqueantes. Modifique o código do cliente e do servidor de maneira a passarem a trabalhar com sockets não-bloqueantes, preparando-os para lidar com o erro `EWOULDBLOCK` que ocorre quando não há espaço no socket para escrever, ou não há dados para ler. No cliente deve-se definir um tempo máximo para a receção de dados de 10 segundos e deve-se executar as operações de escrita apenas após o evento `G_IO_OUT`. No servidor, deve-se preparar o código para receber os dados em várias

evocações de `G_IO_IN`.

- 3) Modifique o código do cliente e do servidor de maneira a passarem a trabalhar com *threads* para enviar e para receber dados através dos *sockets*. **Sugestão:** Analise o código apresentado nas secções 3.3 e 3.4.
- 4) Modifique o código do cliente e do servidor de maneira a passarem a trabalhar com subprocessos para enviar e para receber dados através dos *sockets*. **Sugestão:** Analise o código apresentado nas secções 3.3 e 3.5.