UNIVERSIDADE NOVA DE LISBOA

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Electrotécnica

REDES INTEGRADAS DE TELECOMUNICAÇÕES I

2006 / 2007

Mestrado Integrado em Engenharia Electrotécnica
e de Computadores

4º ano
7º semestre

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

http://tele1.dee.fct.unl.pt **Luis Bernardo**

Índice

1. Objectivo 3
2. Ambiente de desenvolvimento de aplicações em C/C++ no Linux 3
  2.1. Sockets 3
    2.1.1. Sockets datagrama (UDP) 5
    2.1.2. Sockets TCP 6
    2.1.3. Configuração dos sockets 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 8
      2.1.4.4. Definição do alcance de um grupo Multicast 9
      2.1.4.5. Eco dos dados enviados para o socket 9
    2.1.5. Funções auxiliares 9
      2.1.5.1. Conversão entre formatos de endereços 9
      2.1.5.2. Obter o endereço IPv4 ou IPv6 local 10
      2.1.5.3. Obter número de porto associado a um socket 11
      2.1.5.3. Espera em vários sockets em paralelo 12
      2.1.5.4. Obter o tempo actual e calcular intervalos de tempo 13
      2.1.5.5. Outras funções 13
    2.1.6. Estruturas de Dados 13
    2.1.7. Criação de sub-processos 15
    2.1.8. Sincronização entre processos 15
    2.1.9. Temporizadores fora do ambiente gráfico 16
  2.2. Aplicações com Interface gráfica Gtk+/Gnome 16
    2.2.1. Editor de interfaces gráficas Glade-2 17
    2.2.2. Funções auxiliares 20
      2.2.2.1. Tipos de dados auxiliares 21
      2.2.2.2. Referências para objectos gráficos 21
      2.2.2.3. Terminação da aplicação 21
      2.2.2.4. Eventos externos – sockets e pipes 22
      2.2.2.5. Eventos externos – timers 22
  2.3. O ambiente integrado Eclipse para C/C++ 23
  2.4. Configuração do Fedora Core para correr aplicações dual-stack multicast 24
3. Exemplos de Aplicações 24
  3.1. Cliente e Servidor UDP para IPv4 Multicast em modo texto 24
  3.2. Cliente e Servidor UDP para IPv6 Multicast em modo texto 27
  3.3. Cliente e Servidor TCP para IPv6 em modo texto 29
  3.4. Programa com subprocessos em modo texto 31
  3.5. Cliente e Servidor UDP com interface gráfico 32
    3.5.1. Servidor 32
    3.5.2. Cliente 38
    3.5.3. Exercícios 43

1. OBJECTIVO

Familiarização com o ambiente Linux e com o desenvolvimento de aplicações *dual stack* utilizando sockets, a biblioteca gráfica Gtk+/Gnome, a ferramenta Glade-2, 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 trabalho consiste na introdução do código seguindo as instruções do enunciado, aprendendo a utilizar o ambiente e as ferramentas, completando os troços de código omissos.

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, respectivamente 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 RIT1 vai ser usada a biblioteca gráfica do Gnome, designada de Gtk+. A 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-2`), que corre dentro o ambiente gráfico do Eclipse. Para consultar o manual das funções e bibiotecas vai ser usado o comando `man` na linha de comando, ou páginas Web com documentação da interface de programação Gtk+. Tudo o resto pode ser realizado dentro do ambiente integrado Eclipse, embora também pudessem ser usados outros editores de código (e.g. `kate`, `gedit`, `vi`, etc.).

Começa-se o desenvolvimento de uma aplicação no editor da interface gráfica, que cria o código inicial com a inicialização das janelas e com funções de tratamento de eventos (pressão de botões de rato ou teclas). O utilizador tem então 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.

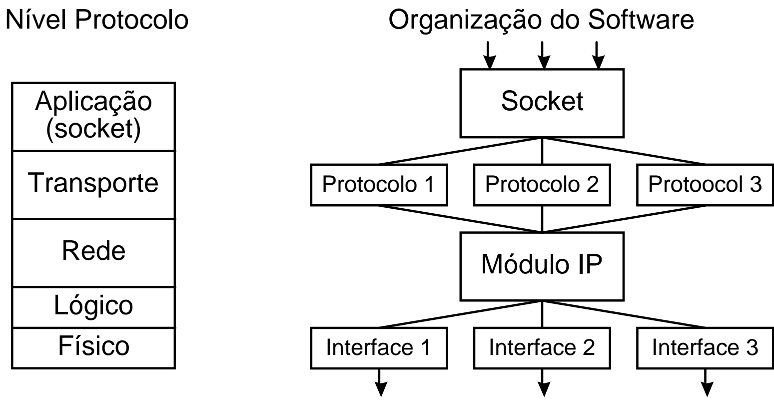
Nesta secção começa-se por introduzir a interface socket, utilizada para enviar mensagens UDP ou TCP. Em seguida introduzem-se a interface de gestão de processos e de comunicação entre processos (*pipes*, *sockets*, e sinais). Na segunda parte introduz-se o desenvolvimento de aplicações usando o glade-2 e a biblioteca gráfica Gtk+.

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 existe hoje disponível em praticamente todos os sistemas operativos. Nas disciplinas anteriores esta interface foi usada indirectamente através de objectos 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. 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 selecciona 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, bidireccional 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 defeito no domínio respectivo (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 respectivamente, 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
  name.sin_port = htons(0);// Atribuição dinâmica
  if (bind(sock, (struct sockadr *)&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):

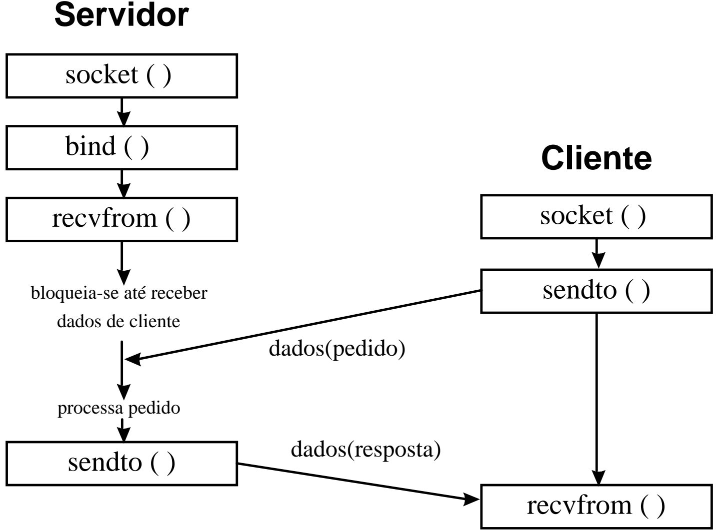
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
…
  struct sockaddr_in6      name;
  unsigned short int       porto;
…
  name.sin6_family = AF_INET6;
  name.sin6_flowinfo= 0;
  name.sin6_port = htons(porto);  /* Porto definido pelo utilizador */
  name.sin6_addr = in6addr_any;  /* 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, excepto 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 de `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 retirá-los do socket;
  `MSG_DONTWAIT` – Não esperar por mensagem.
from – endereço do socket que enviou a mensagem.
fromlen – ponteiro para inteiro inicializado a `sizeof(*from)`.

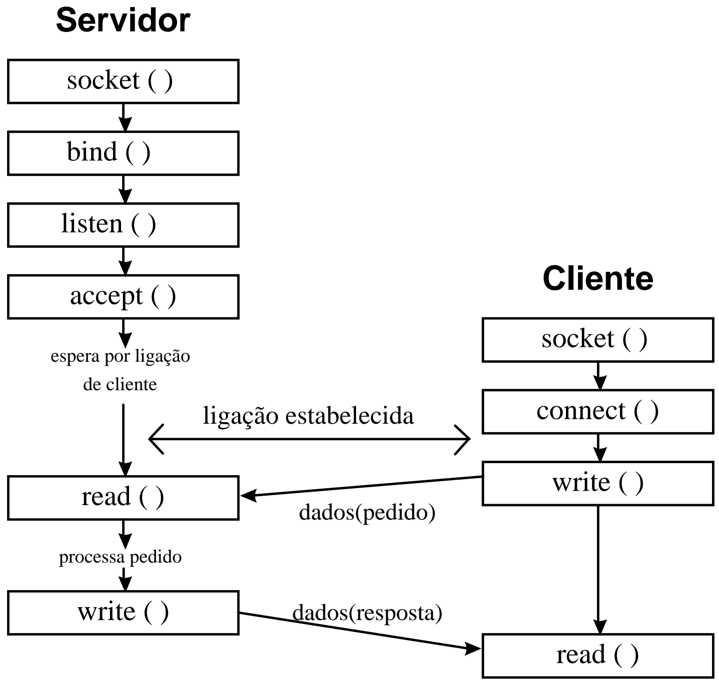
```
int sendto (int s, char *msg, int len, int flags, struct
    sockaddr *to, int tolen);
```

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 efectivamente 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 respectivamente que se quer ler dados fora de banda, ou que pretende espreitar os dados sem os retirar do *buffer*.

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

Permite fechar uma das direcções 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 das várias camadas. Os parâmetros podem ser lidos e modificados respectivamente 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 respectivamente o tamanho dos buffers de recepção e envio de pacotes para sockets TCP e 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 poder-se-ia usar:

```
int v=1000; // 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 todos 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 activando-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 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 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 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

2.1.4.5. 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 deverá ter uma dimensão igual ou superior a INET_ADDRSTRLEN ou INET6_ADDRSTRLEN (respectivamente 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 Iv6
```

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 Internet*/ … }
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 interacção podem ser descritos por um descritor de ficheiro, a função é usada por quase todas as aplicações. As excepções são as aplicações multi-tarefa, onde pode haver várias tarefas activas 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 com 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 excepções). O campo `width` deve ser preenchido com o maior valor de descritor a considerar na mascara 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 activados, ou 0 caso tenha expirado o temporizador, ou –1 em caso de erro. Os eventos activos 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 activo.
```

O código seguinte ilustra a utilização da função `select` para esperar durante dois segundos sobre com 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;              // milisegundos
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 no socket sd1
      …
    }
    if (FD_ISSET(sd2, &rmask)) { // Há dados disponíveis para leitura no socket 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 indirectamente, pois o Gnome permite registar funções para tratar eventos de leitura, escrita ou tratamento de excepções no ciclo principal da biblioteca gráfica.

2.1.5.4. Obter o tempo actual e calcular intervalos de tempo

Existem várias funções para obter o tempo (time, ftime, gettimeofday, etc.). Utilizando a função gettimeofday pode-se obter 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 combinado-os.

```
struct timezone tz;
struct timeval tv;
if (gettimeofday(&tv, &tz)) {
      perror("erro a obter hora de fim de recepcao");
}
```

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 detectado.

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

Chamada	Descrição
bcmp(void*s1,void*s2, int n)	Compara sequências de bytes; retornando 0 se iguais
bcopy(void*s1,void*s2, int n)	Copia n bytes de s1 para s2
bzero(void *base, int n)	Enche com zeros n bytes começando em base
long htonl(long val)	Converte ordem de bytes de inteiros 32-bit de host para rede
short htons(short val)	Converte ordem de bytes de inteiros 16-bit de host para rede
long ntohl(long val)	Converte ordem de bytes de inteiros 32-bit de rede para host
short ntohs(short val)	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 `gethostname` 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; // Nº 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. Criação de sub-processos

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 2.1.9) 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, exactamente 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 sub-processo criado), ou se é o processo filho (retorna 0). Em caso de não haver memória para a criação do sub-processo 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 sub-processo). 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, excepto se usar o parâmetro `options` igual a `WNOHANG`. Nesse caso, retorna `-1` caso não exista nenhum sub-processo *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 indirectamente, detectando falhas nos sub-processos. As funções retornam o parâmetro `status`, que permite detectar se o processo terminou normalmente invocando a operação `_exit`, ou se terminou com um erro (que gera um sinal associado a uma excepção). No exemplo da secção 3.4 está ilustrado como se pode realizar esta funcionalidade.

2.1.8. 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 sub-processos, é comum usar *pipes* ou sockets locais para comunicar entre o processo pai e o processo filho. Um *pipe* é um **canal unidireccional** 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 bidireccional**, pode-se usar a função `socketpair` para criar um par de sockets.

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

// pipe(p);   Cria pipe - so' suporta comunicação    p[1] -> p[0]

if (socketpair(AF_UNIX, SOCK_STREAM, 0, p) < 0) {              // Canal bidireccional
```

```
  perror("falhou o socketpair");
  …
}
```

Para além de funcionarem como sinais de notificação de eventos assíncronos (morte de sub-processo, 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 3.4.

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

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

A recepção 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. 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).

```
#include <unistd.h >

unsigned int alarm(unsigned int seconds);
```

O timer é iniciado associando uma função ao sinal e armando o sinal.

```
// Espera por resposta
    if (signal(SIGALRM, alarm_reaper) == SIG_ERR) {
      perror("RCV>erro a definir alarme"); ...
    }
    alarm(2);       // arma temporizador para daqui a 2 segundos
    // … várias acções …
    alarm(0);       // Desarma o temporizador
```

O processamento do sinal corre assincronamente na função `alarm_reaper`.

```
void alarm_reaper(int sig) {
    // disparou o temporizador
}
```

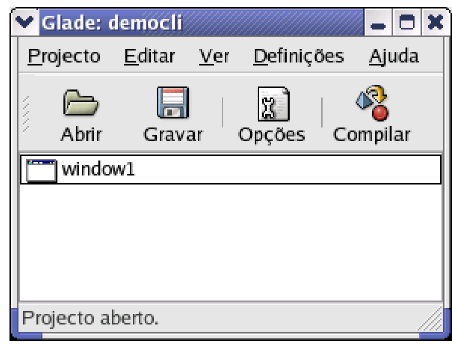
2.2. Aplicações com Interface gráfica Gtk+/Gnome

Numa aplicação em modo texto (sem interface gráfica) o programador escreve a rotina principal (`main`) controlando a sequência de acções 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 objectos terminando com uma invocação à função `gtk_main()`, que fica em ciclo infinito à espera de interacções gráficas, de temporizadores, ou de qualquer outro descritor de ficheiro. Internamente, o Gtk+ realiza esta rotina com a função `select`.

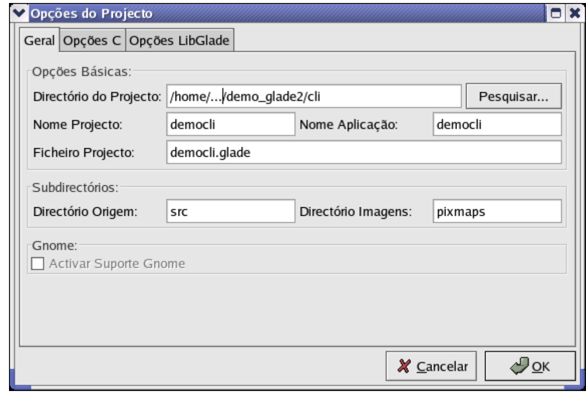
2.2.1. Editor de interfaces gráficas Glade-2

Para facilitar o desenho da interface gráfica de aplicações, foi desenvolvido o editor de interfaces, **Glade-2** (comando `glade-2`). Esta aplicação está incluída nos pacotes da distribuição Fedora ou Red Hat.

Quando se abre um novo projecto surge a janela principal, com as opções para gravar o projecto com a descrição da interface (num ficheiro com terminação '`.glade`'), para abrir um projecto e para gerar o código 'C' (`Build`). A janela mostra ainda a lista de janelas criadas no projecto.



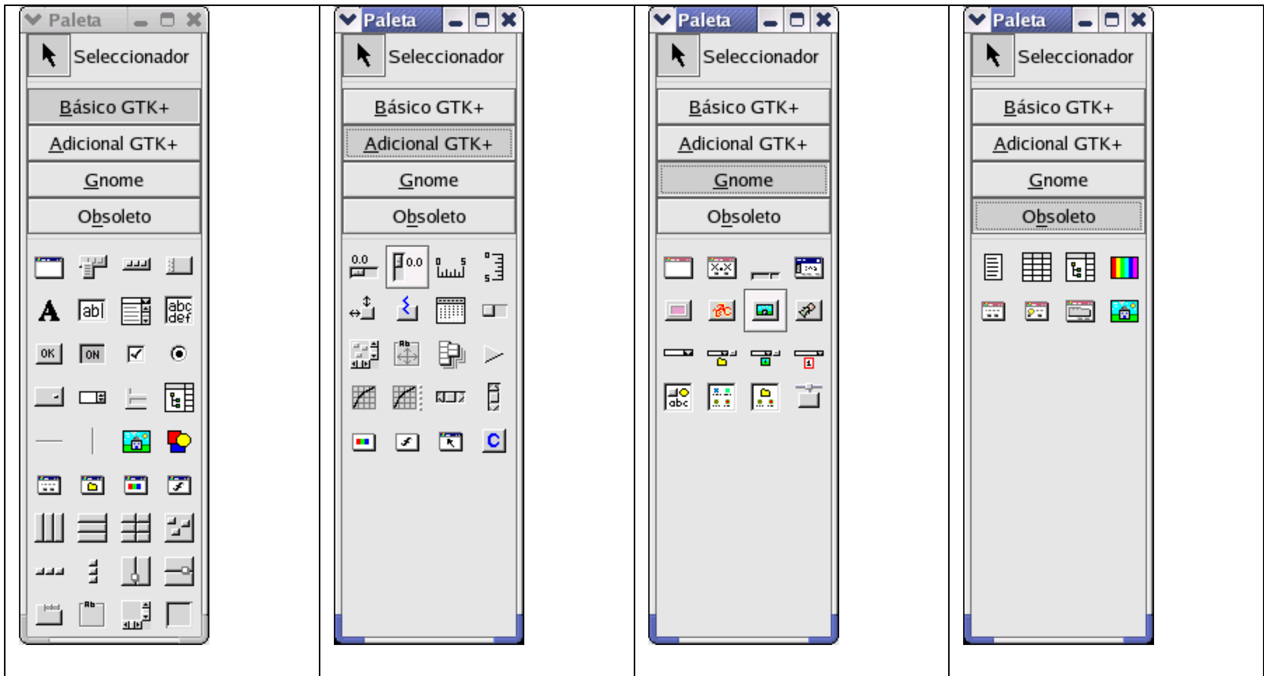
O botão `Opções` permite definir o nome do projecto, do executável gerado, e um conjunto de opções de compilação (linguagem, etc.). Nesta disciplina vai ser usada a linguagem 'C'. A figura seguinte exemplifica a configuração para o programa de demonstração `democli`, apresentado na secção 3.3.2 (página 38).



Na opção "`Ver`" do menu principal é possível activar um conjunto de janelas, também importantes no desenvolvimento de aplicações. A janela `Paleta` apresenta o conjunto de componentes gráficos que é possível usar para criar interfaces. Esta janela tem quatro páginas, representadas abaixo.

Todas as interfaces são desenhadas a partir de um componente básico (`GtkWindow`, `GnomeApp` ou `GtkDialog`). Todos os exemplos fornecidos neste documento foram desenvolvidos usado 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. Estes componentes ( ) permitem respectivamente: a divisão em colunas da janela; a divisão em linhas; a divisão da janela numa matriz; e a colocação em posições arbitrárias dos vários componentes. A desvantagem do último componente é que não permite 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 `Básico GTK+`) foram:

`GtkTextView` (com `GtkScrolledWindow`) – Caixa editável com múltiplas linhas.

`GtkLabel` – Títulos;

`GtkEntry` – Caixa com texto editável;

`GtkButton` – Botão;

`GtkToggleButton` – Botão com estado;

`GtkFileSelection` – Janela de selecção de ficheiros.

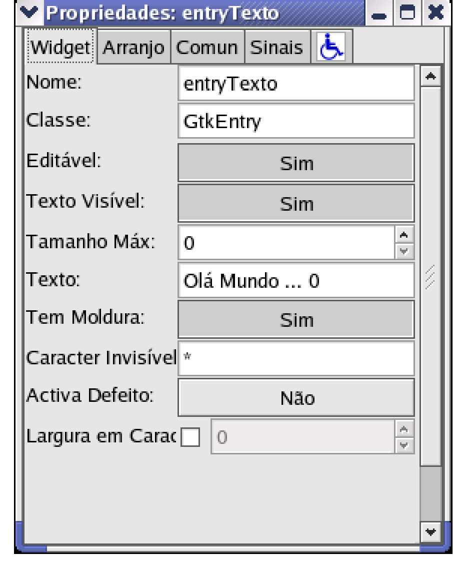
Foi ainda usado um componente gráfico da página `Obsoleto`:

`GtkCList` (com `GtkScrolledWindow`) – Tabelas com múltiplas colunas.

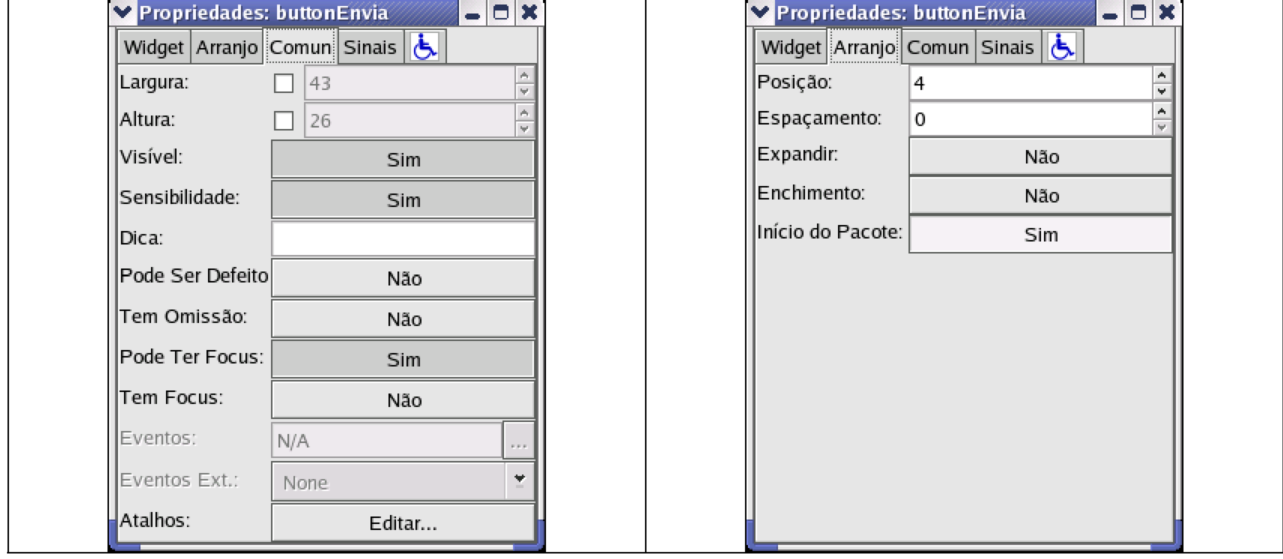
A organização dos componentes da interface tem uma estrutura em árvore que pode ser visualizada na janela "`Widget Árvore`" acessível a partir do menu principal "`Ver`". Por exemplo, para o cliente (exemplo da página 38) tem-se a árvore representada à esquerda.

Tal como noutros editores integrados usados anteriormente, é possível editar as propriedades iniciais dos componentes gráficos na janela "**`Propriedades`**" do objecto seleccionado na janela "`Widget Árvore`". Esta janela tem cinco páginas. A primeira

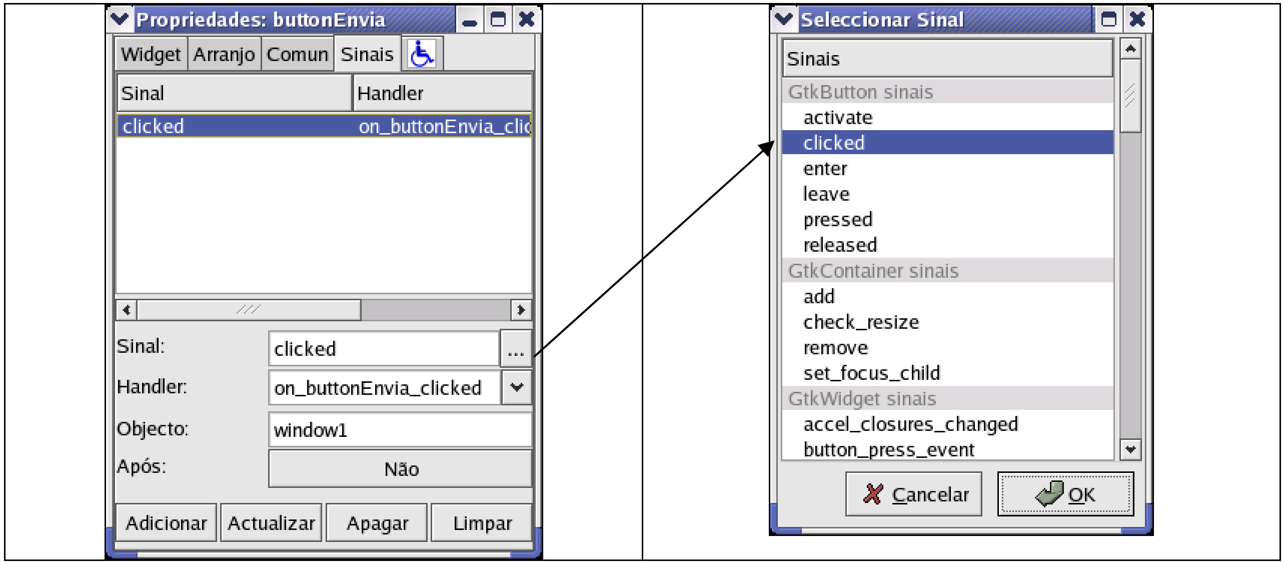
("Widget") contém a definição do nome do objecto gráfico (`Nome:`), do tipo do objecto (`Clase:`), e de várias propriedades específicas para cada objecto. No caso de uma `GtkEntry` pode-se definir se é editável, se o texto é visível, o texto inicial, o comprimento máximo do texto, etc.



A página "Comun" controla aspectos gráficos como o comprimento e a largura da caixa. Também é possível usar o editor gráfico para definir os valores destas propriedades. A página "Arranjo" controla a posição relativa do componente quando este se encontra numa linha ou numa coluna. No exemplo da figura, o componente "`buttonEnvia`" está na quinta posição da "`hbox1`", podendo-se mudar a posição relativa mudando o valor de "`Posição:`".



A página Sinais permite associar funções aos eventos suportados pelos vários componentes gráficos. Para cada componente podem-se associar várias funções a diferentes eventos (designados de "Sinais"). Ao premir o botão "…" abre-se a janela representada à direita, com a lista de eventos suportados pelo componente. O campo "`Handler:`" representa o nome da função, criado automaticamente a partir do nome do evento e do objecto. O campo "`Objecto:`" permitem passar um argumento na invocação da função. No caso do exemplo é passado um ponteiro para a janela principal, "`window1`", mas poderia ser qualquer outro componente gráfico.



O `Glade-2` cria o código do ciclo principal dos programas e os guiões de instalação automática e de compilação dos programas. O `Glade-2` gera sete ficheiros de código a partir da descrição da interface. Por defeito são gerados os seguintes ficheiros:

Ficheiro	Descrição	Modificável
src/interface.h src/interface.c	Funções para criação das janelas e caixas de diálogo	NÃO
src/support.h src/support.c	Funções auxiliares do Glade-2, incluindo a função `lookup_widget`	NÃO
src/main.c	Função `main`. Não é reescrito pelo Glade-2	SIM
src/callbacks.h src/callbacks.c	Funções vazias de tratamento dos eventos. O Glade-2 permite acrescentar novas rotinas ou botões, mantendo o código.	SIM
src/Makefile.am	Guião para geração da Makefile definitiva	SIM

A partir do código gerado pelo Glade-2, o programador apenas tem de modificar o guião de compilação (`Makefile.am`) acrescentando todos os nomes dos ficheiros de código externos ao Glade-2, de programar o conteúdo das rotinas de tratamento de eventos (`callbacks.c`), e de modificar o ciclo principal do programa, se necessário. Antes de executar o programa, tem de se correr o ficheiro de comandos "**`autogen.sh`**", e de compilar o código com o comando "**`make`**".

O Glade-2 está disponível através na linha de comando "`glade-2 [nome do projecto]`", ou depois de configurado, através do ambiente integrado Eclipse. Vários ficheiros de documentação (incluindo o manual e o FAQ) estão disponíveis no menu "Ajuda" em formato HTML.

Na secção 3.5 deste documento, a partir da página 32, são apresentados dois exemplos programas desenvolvidos utilizando o Glade-2.

2.2.2. Funções auxiliares

Para desenvolver uma aplicação em Gtk+/Gnome é necessário usar várias funções auxiliares para aceder aos objectos gráficos. Adicionalmente, existem funções para lidar com os descritores activos no ciclo principal, para trabalhar com strings, listas, etc. A descrição deste conjunto de funções está num ficheiro comprimido (`Gnome2_API.tar.gz`), sendo, para além disso, fornecidos dois exemplos de programas que usam algumas das funcionalidades. Para

descomprimir o ficheiro pode-se usar o comando (gunzip Gnome2_API.tar.gz; tar xvf Gnome2_API.tar). Nesta secção são apresentadas algumas funções que lidam com aspectos mal documentados desta biblioteca.

2.2.2.1. Tipos de dados auxiliares

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 pointeiro 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, gconstpointer 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_lastGList *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);    // Retorn n-ésimo membro da lista
// OS dados são acedidos através do campo data: (Tipo)pt->data
```

A lista completa de tipos suportados está nos ficheiros da directoria Glib do ficheiro (Gnome2_API.tar.gz).

2.2.2.2. Referências para objectos gráficos

O código gerado pelo Glade-2 cria as janelas em funções "create_window1, create_window2, …" declaradas nos ficheiros "interface.c". Por omissão cria todas as janelas na função "main" (excepto as GtkFileSelection) retornando apenas a referência para o objecto base de cada janela, mas é possível modificar o ficheiro "main.c". Para aceder aos componentes gráficos internos a uma janela é necessário usar a função "lookup_widget" que realiza uma pesquisa através do nome do objecto.

```
GtkWidget* lookup_widget(GtkWidget *widget, const gchar *widget_name);
```

Por exemplo, para obter uma referência para a caixa "entryIP" na janela "window1" poder-se-ia usar a seguinte função. Note-se a conversão explícita de tipo (GTK_ENTRY).

```
GtkWidget *entry_RIP= lookup_widget(window1, "entryIP");
const char *textRIP= gtk_entry_get_text(GTK_ENTRY(entry_RIP));
```

As janelas de selecção de ficheiros (GtkFileSelection) têm de ser arrancadas e paradas explicitamente com o excerto de código seguinte, e têm de ser programadas rotinas para processar o premir dos botões OK e Cancelar. O nome completo do ficheiro seleccionado pode ser obtido usando a função gtk_file_selection_get_filename.

```
GtkFileSelection *fs= NULL;
// Para criar:
fs= GTK_FILE_SELECTION(create_fileselection1());
gtk_widget_show (GTK_WIDGET(fs));
// Para fechar:
gtk_widget_destroy(GTK_WIDGET(fs));
// Para obter nome:
const gchar *nome= gtk_file_selection_get_filename(fs);
```

2.2.2.3. 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;
}
```

2.2.2.4. Eventos externos – sockets e pipes

O Gnome permite registar funções de tratamento de eventos de leitura, escrita ou excepções 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 excepções (`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 /* parametro recebido na função*/)) ) {
    g_print("Falhou activação de recepção de dados\n"); …
}
```

A função `callback_dados` deverá 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 activa */
  }else if (condition == G_IO_OUT ) {
    /* Há espaço para continuar a escrever dados …*/
    return TRUE; /* a função continua activa */
  } else if ((condition == G_IO_NVAL) || (condition == G_IO_ERR)) {
  } else if ((condition == G_IO_NVAL) || (condition == G_IO_ERR)) {
    /* Trata erro … */
    return FALSE; /* Deixa de receber evento */
  }
}
```

Pode-se desligar 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_close(chan);
```

2.2.2.5. 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 milisegundos */
                    callback_timer,/* Função invocada */
                    NULL);  /* Argumento passado à função */
```

A rotina 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.

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

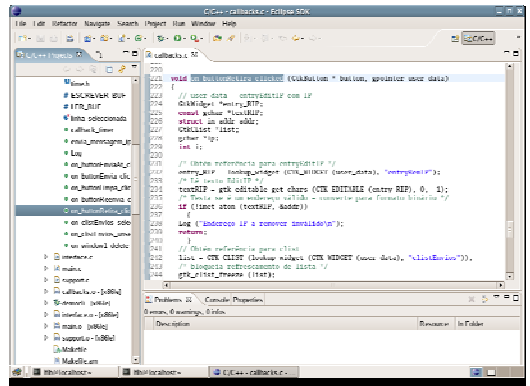
O ambiente de desenvolvimento Eclipse permite desenvolver aplicações C/C++ de uma forma simplificada, a nível de edição e debug. O mecanismo de indexação permite-lhe mostrar onde é que uma função ou variável é declarada, facilitando muito a visualização de código existente. O debugger integrado permite depois visualizar os valores de uma variável posicionando o rato sobre elas, ou introduzir pontos de paragem. O ambiente é especialmente indicado para C++, mas também é útil para C. No entanto, durante a preparação do trabalho foi identificado um erro na versão 3.1.1., distribuída com o linux Fedora Core 4: o editor bloqueia cada vez que se introduz ":". Para evitar este problema deve-se desactivar a interpretação automática do "::", seguindo-se a sequência de menus e submenus: *"Window"; "Preferences"; "C/C++"; "Editor"; "Code Assist"* - nessa caixa deve-se desligar a opção *"Auto activaton" Enable "::" as trigger*.

Por omissão, o Eclipse não reconhece o `glade-2` como o editor de ficheiros "*.glade*". Para fazer edição integrada de ficheiros "*.glade*" deve-se associar a aplicação "*/usr/bin/glade-2*" à extensão "*.glade*", no menu: *"Window"; "Preferences"; "General"; "Editors"; "File Associations"*.

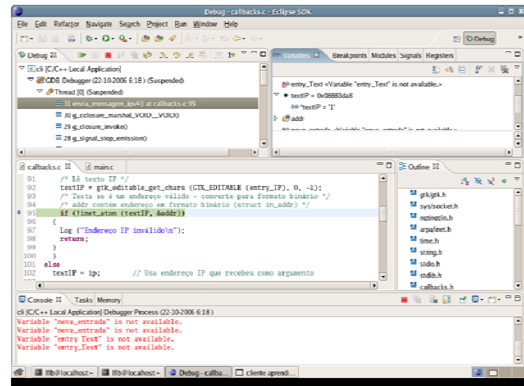
Um problema que por vezes ocorre em projectos grandes é a falta de memória. Por omissão, o eclipse arranca com 256 MBytes de memória, mas é possível aumentar essa memória, definindo um ficheiro de comandos para arrancar o eclipse, com o seguinte conteúdo:

```
/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 *"Debug"*, 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-se escolher o projecto e o executável da aplicação; na janela "*Debugger"* deve-se escolher o *"GDB Debugger"*, que corre a aplicação `gdb`.



(a) Edição de ficheiros



(b) Dubugging

Nas duas imagens anteriores ilustra-se o aspecto 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 Fedora Core para correr aplicações dual-stack multicast

Para conseguir correr aplicações de pilha dupla, com multicast, é necessário activar o suporte de IPv6, acrescentando a seguinte linha ao ficheiro */etc/sysconfig/network*:

```
NETWORKING_IPV6=yes
```

Para além disso, é necessário associar pelo menos um endereço IPv6 de âmbito global. Numa rede com um encaminhador a correr um servidor `radvd` (*Linux IPv6 Router Advertisement Daemon*) ou `dhcpv6d` (Dynamic *Host Configuration Protocol Server daemon for IPv6*), o endereço é obtido dinamicamente. Caso contrário, é necessário definir o endereço manualmente. No caso do dispositivo *eth0*, isso pode ser feito editando o ficheiro */etc/sysconfig/network-scripts/ifcfg-eth0*:

```
DEVICE=eth0
…
IPV6INIT=yes
//   Para endereço dinâmico
IPV6_AUTOCONF=yes
//   Para endereço fixo igual a 2001:690:1fff:bb::1 :
IPV6_AUTOCONF=no
IPV6ADDR=2001:690:1fff:bb::1/120
```

Por omissão, a tabela de encaminhamento não suporta endereços IPv4 multicast. Para passar a suportar é necessário acrescentar introduzir o comando seguinte. Este comando pode ser colocado no ficheiro */etc/rc.local* para correr sempre que o computador arranca.

```
route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0
```

A firewall do sistema pode também bloquear o funcionamento das aplicações. Nesse caso dever-se-á desactivar temporariamente a firewall com o comando *"iptables –F"*, ou acrescentar regras para que os portos das aplicações sejam aceites.

3. Exemplos de Aplicações

Nesta secção são fornecidos três 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, respectivamente 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 vários sub-processos. A secção 3.5 descreve o desenvolvimento da interface gráfica e a sua integração com sockets. É fornecido um projecto 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 (sugere-se o editor `kate`). 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`" será:

```
#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 defeito o porto é atribuído pelo sistema */
  int  reuse= 1;
  /* Multicast */
  struct ip_mreq imr;
  char loop = 1;
  u_char ttl= 1; /* rede local */
  /* recepção */
  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 defeito
  name.sin_port = htons(porto);
  if (bind(sock, (struct sockaddr *)&name, sizeof(name)) < 0) {
    perror("Falhou binding de socket datagrama");
    exit(1);
  }
  /* Configuracoes 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 defeito */
  /* 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 defeito 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`" será:

```
#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 "Ola mundo ..." /* 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 *) &ttl,
      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 objectivo 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`" será:

```
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, exceptuando 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 *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`" será:

```
#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;
  /* recepcao */
  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`" será:

```
#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 "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.

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.

O código do cliente "`clitcpv6.c`" será:

```
#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;
      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);
  }
  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 sub-processos, 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`" será:

```
#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) {
    static char buf[100];
    inet_ntop(AF_INET6, addr, buf, sizeof(buf));
    return buf;
}

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 {
```

```
            printf("Connection from %s - %d\n",
                   addr_ipv6(&server.sin6_addr),
                   ntohs(server.sin6_port));
            bzero(buf, sizeof(buf));
            if (read(msgsock, buf, 1024) < 0)
                   perror("receiving stream data");
            else
                   printf("-->%s\n", buf);
            close(msgsock);
      }
 }
}
```

3.4. Programa com subprocessos em modo texto

Este exemplo ilustra a criação e terminação de um sub-processo e a utilização de um *pipe* para enviar uma mensagem do processo filho para o processo pai. A função `reaper` analisa o motivo porque o processo filho terminou.

O código do programa "`demofork.c`" será:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <signal.h>
#include <sys/wait.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.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[256];
  int n, m;

  signal(SIGCHLD, reaper);                        // arma callback para sinal SIGCHLD
  n= socketpair(AF_UNIX, SOCK_STREAM, 0, p);      // Cria par de sockets 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
    write(p[1], msg, strlen(msg)+1);// Envia msg para pai
    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], buf, sizeof(buf));      // Espera por mensagem do filho
  } while ((m == -1) && (errno == EINTR));// Repete se for interrompido por sinal
  fprintf(stderr, "Child process %d returned :%s\n", n, buf);

}
```

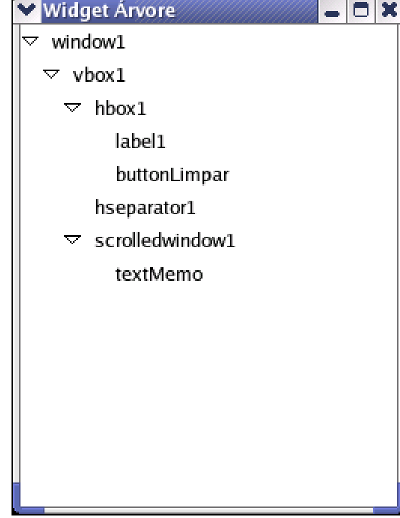
3.5. Cliente e Servidor UDP com interface gráfico

Esta segunda aplicação distribuída é constituída por dois executáveis, criados a partir de dois ficheiros Glade-2. 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 milisegundos configurável. O cliente regista todos os envios efectuados, podendo reenviar uma mensagem para um endereço IP anterior.

3.5.1. Servidor

O servidor deve ser programado numa directoria diferente do cliente. A primeira parte da programação do servidor consiste no desenho da interface gráfica do servidor utilizando a aplicação Glade-2. Deve-se criar o projecto "`demoserv.glade`"

O servidor é desenvolvido a partir de uma janela `GtkWindow` (`window1`), subdividindo-se a janela em três linhas utilizando uma `GtkVBox` (`vbox1`) e a primeira linha em duas colunas com um `GtkHBox` (`hbox1`). Alternativamente, poderá usar-se o componente que permite colar componentes gráficos em posições arbitrárias. Em seguida é introduzido um `GtkLabel` com "porto 20000", um botão (`buttonLimpar`) e um `GtkTextView` (`textMemo`) com *scrollbar*, devendo-se mudar o nome de acordo com a árvore de componentes representada à esquerda.



Para realizar a primeira compilação deverá começar por criar o código no Glade-2, e em seguida, na linha de comando deve correr o comando "`./autogen.sh`", compilar a aplicação com "`make`"; e correr o executável "`./src/democli`".

Numa segunda fase, voltando ao Glade-2, vão-se criar as funções de tratamento dos eventos gráficos.

Vai-se associar uma rotina ao sinal "**`delete`**" da janela principal "**`window1`**", de forma a parar o executável quando se fecha a janela. Depois de gerar o código no Glade-2 deve-se editar o ficheiro "`src/callbacks.c`" acrescentando o seguinte código:

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

Também se vai associar uma rotina ao evento "**`clicked`**" do botão "**`buttonLimpar`**", de forma a limpar o conteúdo da caixa "`textMemo`". Depois de gerar o código no Glade-2 deve-se editar o ficheiro "`src/callbacks.c`" acrescentando o seguinte código:

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

/* Obtém referência para "textMemo" a partir do objecto botão */
  textbox= GTK_TEXT_VIEW(lookup_widget(GTK_WIDGET(button), "textMemo"));
/* Obtém referência para modelo com dados */
  textbuf= GTK_TEXT_BUFFER(gtk_text_view_get_buffer(textbox));
/* 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`, que deverá ser declarada em `callbacks.c`.

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

  Assert(str != NULL);
  /* Obtém referência para 'textMemo' a partir da variável global 'main_window' a
      iniciar em 'main.c' */
  textbox= GTK_TEXT_VIEW(lookup_widget(main_window, "textMemo"));
  textbuf= GTK_TEXT_BUFFER(gtk_text_view_get_buffer(textbox));
  gtk_text_buffer_get_iter_at_offset(textbuf, &tend, -1);
  /* Acrescenta caracteres à textMemo */
  gtk_text_buffer_insert(textbuf, &tend, str, strlen(str));
}
```

Todo o código para suportar a comunicação com o socket UDP tem de ser realizado fora do ambiente gráfico Glade-2. Para facilitar o desenvolvimento vai ser criado um módulo (`sock.c` e `sock.h`), com um conjunto de funções auxiliares para lidar com sockets. Estes ficheiros devem ser criados na directoria `src`.

O texto do ficheiro "`sock.c`" será:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <assert.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

#include "sock.h"

// Inicializa socket
// Retorna: -1 - erro;  >0 - n de socket
int init_socket_ipv4(int porto)        // Inicializa socket
{
  struct sockaddr_in name;
  int s;

  // Cria socket IPv4 UDP
  s= socket(AF_INET, SOCK_DGRAM, 0);
  if (s < 0) {
    perror("abertura socket datagrama");
    return -1;
  }
  // Associa socket a nmero de porto
  name.sin_family= AF_INET;// Domínio endereço IPv4
  name.sin_addr.s_addr= INADDR_ANY; // Endereço IP local (0.0.0.0)
  name.sin_port= htons((short)porto); // Número de porto
  if ( bind(s, (struct sockaddr *)&name, sizeof(name)) ) {
    if (errno == EINVAL) {
      g_print("O socket já está associado a um porto\n");
    }
    perror("associação a número de porto");
    return -1;
  }
  return s;
}

// lê dados, retorna número bytes lidos (<0 em caso de erro), ip e porto de emissor
int read_dados_ipv4(int sock, char *buf, int n, struct in_addr *ip,
                short int *porto)
{
  int m;
  struct sockaddr_in from;
  int fromlen= sizeof(from);

  assert(buf != NULL);
  assert(n > 0);
  assert(ip != NULL);
  assert(porto != NULL);
  if (sock < 0)
    return -1;
  if ((m= recvfrom(sock, buf, n, MSG_DONTWAIT /* nao bloqueante */,
      (struct sockaddr *)&from, &fromlen)) < 0)
    return m;
  *ip= from.sin_addr;  // IP no formato rede
  *porto= ntohs(from.sin_port);  // Converte n porto para formato máquina
  return m;
}

gboolean put_socket_in_mainloop(int sock, void *ptr, int *chan_id,
            GIOChannel **chan,
            gboolean (*callback) (GIOChannel *, GIOCondition, gpointer))
{
  /* Teste parâmetros */
  assert(sock >= 0);
  assert(chan_id != NULL);
  assert(chan != NULL);
  assert(callback != NULL);

  /* adiciona socket ao ciclo principal do Gtk */
  if ( (*chan= g_io_channel_unix_new(sock)) == NULL) {
    g_print("Falhou criação de canal IO\n");
    return FALSE;
  }
  if (! (*chan_id= g_io_add_watch(*chan,
      G_IO_IN|G_IO_NVAL|G_IO_ERR, /* após eventos de leitura e erros */
      callback /* função chamada */,
      ptr /* passa ponteiro no parametro da função chamada */)) ) {
    g_print("Falhou activação de recepção de dados\n");
```

```
    return FALSE;
  }
  return TRUE;
}

void remove_socket_from_mainloop(int sock, int chan_id, GIOChannel *chan)
{
  assert(sock >= 0);
  assert(chan != NULL);
  /* Retira socket do ciclo principal do Gtk */
  g_source_remove(chan_id);
  /* Liberta canal */
  g_io_channel_close(chan);
}

void close_socket(int sock)                // Fecha socket
{
  if (sock < 0)
    return;
  close(sock);
}
```

O texto do ficheiro "`sock.h`" será:

```
#ifndef _INCL_SOCK_
#define _INCL_SOCK_

#include <netinet/in.h>
#include <gtk/gtk.h>
#include <glib.h>

/* Macro para facilitar recepção de dados */
/* pt - ponteiro de leitura */
/* var - ponteiro para variável */
/* n - número de bytes a ler */
#define LER_BUF(pt, var, n)  bcopy(pt, var, n); pt+= n

/* Macro para facilitar escrita de dados */
/* pt - ponteiro de leitura */
/* var - ponteiro para variável */
/* n - número de bytes a ler */
#define ESCREVER_BUF(pt, var, n)  bcopy(var, pt, n); pt+= n

int init_socket_ipv4(int porto); // Inicializa socket

int read_dados_ipv4(int sock, char *buf, int n, struct in_addr *ip,
            short int *porto);
// lê dados, retorna número de bytes lidos ou -1 e endereço e porto do emissor

gboolean put_socket_in_mainloop(int sock, void *ptr, int *chan_id,
            GIOChannel **chan,
            gboolean (*callback) (GIOChannel *, GIOCondition, gpointer));

void remove_socket_from_mainloop(int sock, int chan_id, GIOChannel *chan);

void close_socket(int sock);               // Fecha socket

#endif
```

Para compilar e integrar o módulo `sock` é necessário modificar o ficheiro `Makefile.am`, acrescentando os dois ficheiros à lista de dependências, conforme o seguinte texto:

```
## Process this file with automake to produce Makefile.in

INCLUDES = \
      -DPACKAGE_DATA_DIR=\""$(datadir)"\" \
      -DPACKAGE_LOCALE_DIR=\""$(prefix)/$(DATADIRNAME)/locale"\" \
      -g \
      @PACKAGE_CFLAGS@

bin_PROGRAMS = demoserv

demoserv_SOURCES = \
      main.c \
      support.c support.h \
      interface.c interface.h \
      callbacks.c callbacks.h \
      sock.c sock.h
```

```
demoserv_LDADD = @PACKAGE_LIBS@
```

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:

```
/*
 * Ficheiro main.c inicial gerado pelo Glade. Edite segundo as
 * suas necessidades. Glade não irá¡ substituir este ficheiro.
 */

#ifdef HAVE_CONFIG_H
#  include <config.h>
#endif

#include <gtk/gtk.h>
#include <stdio.h>
#include <stdlib.h>

#include "interface.h"
#include "support.h"

#include "sock.h"
#include "callbacks.h"

/* Variáveis públicas */
GtkWidget *main_window; // Janela príncipal
int sock= -1;       // descritor do socket
GIOChannel *chan= NULL; // Descritor do canal do socket
guint chan_id;      // Número de canal

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

#ifdef ENABLE_NLS
  bindtextdomain (GETTEXT_PACKAGE, PACKAGE_LOCALE_DIR);
  bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
  textdomain (GETTEXT_PACKAGE);
#endif

  gtk_set_locale ();
  gtk_init (&argc, &argv);

  add_pixmap_directory (PACKAGE_DATA_DIR "/" PACKAGE "/pixmaps");

  /*
   * The following code was added by Glade to create one of each component
   * (except popup menus), just so that you see something after building
   * the project. Delete any components that you don't want shown initially.
   */
  main_window = create_window1 ();
  gtk_widget_show (main_window);

  /* Inicialização do socket */
  if ((sock= init_socket_ipv4(20000)) == -1)
    exit(-1);
  if (!put_socket_in_mainloop(sock, main_window, &chan_id, &chan,
      callback_dados))
    exit(-2);

  /* Ciclo principal */
  gtk_main (); /* só sai após gtk_main_quit() ser chamado */
  return 0;
}
```

Para terminar a programação do servidor falta apenas programar a rotina que recebe os dados do socket. A opção foi de incluir este código no ficheiro "`callbacks.c`", gerado inicialmente pelo Glade-2, obrigando a acrescentar as seguintes linhas:

```
#include <gtk/gtk.h>
#include <arpa/inet.h>
#include <assert.h>
#include "callbacks.h"
#include "interface.h"
#include "support.h"
#include "sock.h"
#include <time.h>
#include <memory.h>
#include <stdio.h>
```

A rotina de leitura recorre à macro **LER_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`. A rotina de tratamento dos eventos do socket tem o seguinte código (no ficheiro `callbacks.c`):

```
static char write_buf[1024];

gboolean callback_dados (GIOChannel *source, GIOCondition condition, gpointer data)
{
  static char buf[MAX_COMP_MENSAGEM];  // buffer para leitura de dados
  struct in_addr ip;
  short unsigned int porto;
  int n;
  GtkTextView *textbox;

  if (condition == G_IO_IN ) {
    /* Obtém referência para 'textMemo' a partir da variável global main_window */
    textbox= GTK_TEXT_VIEW(lookup_widget(main_window, "textMemo"));
    /* Lê dados */
    n= read_dados_ipv4(sock, buf, MAX_COMP_MENSAGEM, &ip, &porto);
    if (n<=0) {
      /* Acrescenta linha à Memo */
      Log("Falhou leitura de dados do socket\n");
      return TRUE;  // Continua à espera de mais dados
    } else {
      time_t tbuf;
      short unsigned int m;
      char *pt;
      /* Escreve data e origem dos dados */
      time(&tbuf);  // Obtém data actual
      sprintf(write_buf,"%sRecebidos %d bytes de %s:%hu\n", ctime(&tbuf), n,
          inet_ntoa(ip), porto);
      Log(write_buf);
      /* Lê dados */
      pt= buf;
      LER_BUF(pt, &m, 2);  // Lê e avança ponteiro
      m= ntohs(m);  // Converte para formato host
      if (m != n-2) {
        sprintf(write_buf, "Campo 'comprimento' inválido (%d != %d)\n", m,
            n-2);
      Log(write_buf);
      return TRUE;  // Continua à espera de mais dados
      }
      /* Escreve conteúdo da mensagem na memo  - admite que termina com '\0' */
      Log(pt); Log("\n");
      return TRUE;  // Continua à espera de mais dados
    }

  } else if ((condition == G_IO_NVAL) || (condition == G_IO_ERR)) {
    Log("Detectado erro no socket\n");
    remove_socket_from_mainloop(sock, chan_id, chan);
    chan= NULL;
    /* Fecha socket */
    close_socket(sock);
    sock= -1;
    /* Sai da aplicação */
    gtk_main_quit();
    return FALSE; // Remover do ciclo principal
  } else {
    assert(0); // Não deve nunca chegar a esta linha - aborta aplicação com erro
    return FALSE; // Remover do ciclo principal
  }
}
```

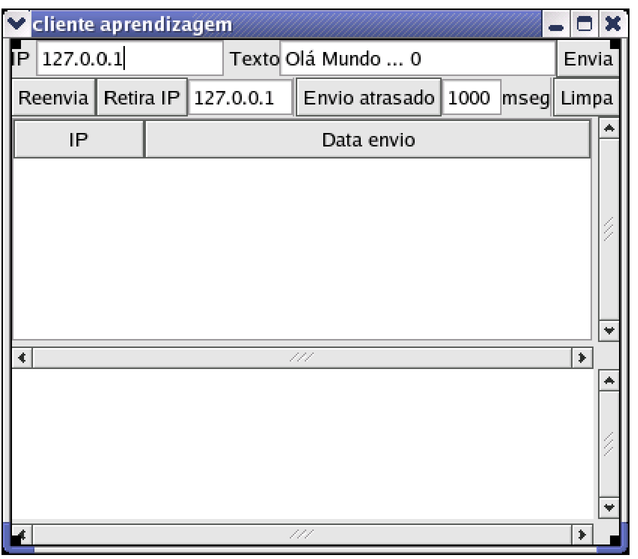
Observe-se que a mensagem é composta por dois bytes com o comprimento e pelo conteúdo da string. Falta apenas declarar o valor do comprimento máximo da mensagem (MAX_COMP_MENSAGEM) e a assinatura das funções criadas e das variáveis globais acrescentando o seguinte texto ao ficheiro "callbacks.h":

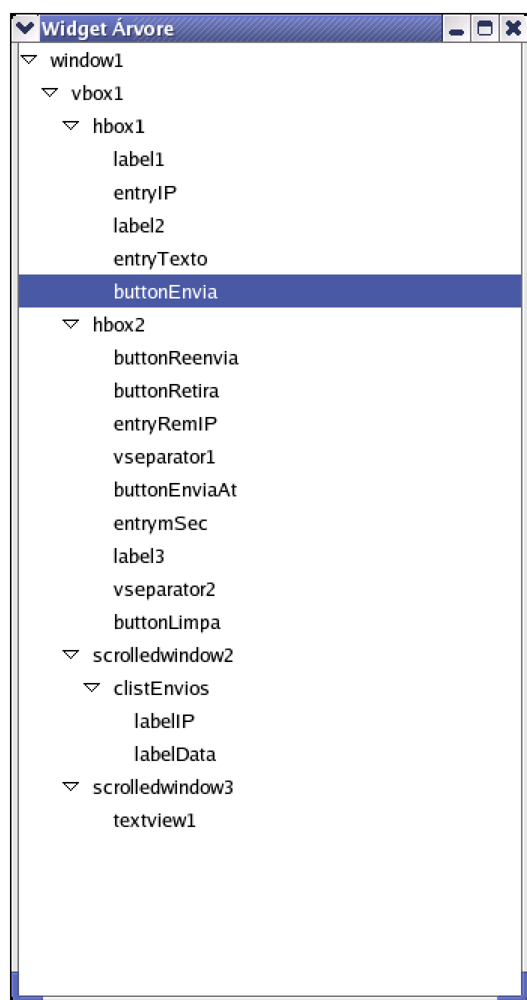
```
#include <gtk/gtk.h>
#include <glib.h>

// Variáveis declaradas e initializadas em 'main.c'
extern GtkWidget *main_window;
extern int sock;      // Descritor do socket
extern GIOChannel *chan; // Descritor do canal do socket
extern guint chan_id;     // Número de canal

gboolean
on_window1_delete_event               (GtkWidget       *widget,
                                        GdkEvent        *event,
                                        gpointer         user_data);

void
on_buttonLimpar_clicked               (GtkButton       *button,
                                        gpointer         user_data);

#define MAX_COMP_MENSAGEM   9000

/* Callback para receber dados do socket */
gboolean callback_dados (GIOChannel *source, GIOCondition condition, gpointer data);
```

Para compilar a aplicação basta correr o comando "make" na directoria src. O executável é criado nessa directoria.

3.5.2. Cliente

Crie agora uma nova directoria para o cliente. A primeira parte da programação do cliente é novamente o desenho da interface gráfica do servidor utilizando a aplicação Glade-2. Deve-se criar o projecto "democli.glade"





A interface gráfica é realizada a partir de uma janela GtkWindow (window1) que é dividida em quatro linhas (vbox1). A primeira linha encontra-se dividida em 5 colunas, respectivamente: uma GtkLabel (inicializada com "IP"), uma

GtkEntry ("entryIP" com o texto "127.0.0.1"); uma GtkLabel (inicializada com " Texto"); uma GtkEntry ("entryTexto" com o texto por defeito "Olá Mundo ... 0"); e um GtkButton ("buttonEnvia" com o texto "Envia"). A segunda linha encontra-se dividida em 9 colunas, respectivamente: um GtkButton ("buttonReenvia" com o texto "Reenvia"); um GtkButton ("buttonRetira" com o texto "Retira IP"); uma GtkEntry ("entryRemIP") com o texto "127.0.0.1"; um separador vertical (vseparator1); um GtkButton ("buttonEnviaAt" com o texto "Envio atrasado"); uma GtkEntry ("entrymSec" com o texto "1000"); um GtkLabel (label3 inicializada com "mseg"); um separador; e um GtkButton ("buttonLimpa" com o texto "Limpa"). A penúltima linha contém uma GtkCList ("clistEnvios") com duas colunas. A última linha contém uma GtkTextView (textview1).

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

Neste exemplo apenas é fornecido o código relacionado com a interacção gráfica. **Não é fornecido o código de inicialização do socket e de envio de mensagens, que é deixado para exercício.**

Para facilitar a escrita de mensagens na caixa de diálogo, vai-se usar novamente a função Log, apresentada na página 33, que deve ser declarada no código do cliente (callbacks.c) e no ficheiro com os cabeçalhos (callbacks.h). 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 "entryTexto" 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 envia_mensagem_ipv4 (gpointer data /* referência para uma Widget da janela*/,
                 const gchar *ip);
```

Para facilitar a escrita num buffer é usada a macro ESCREVER_BUF, apresentada anteriormente no ficheiro "sock.h", na página 35, que copia n bytes a partir do endereço var para pt, incrementando pt. O código parcial da função (sem a composição e envio de mensagens) deverá ser programado no ficheiro "callbacks.c" depois da declaração da macro ESCREVER_BUF:

```
/* Função auxiliar para enviar a mensagem */
void envia_mensagem_ipv4 (gpointer data /* referência para uma Widget da janela*/,
  const gchar *ip)
{
  GtkWidget *entry_IP;
  GtkWidget *entry_Text;
  const gchar *textText;   // Texto a enviar
  const gchar *textIP;     // Endereço IP
  struct in_addr addr;
  GtkCList *list;
  char *nova_entrada[2];
  time_t tbuf;

  if (ip == NULL) {
    /* Obtém referência para entryIP a partir de uma janela */
    entry_IP= lookup_widget(GTK_WIDGET(data), "entryIP");
    /* Lê texto com endereço IP */
    textIP= gtk_entry_get_text(GTK_ENTRY (entry_IP));
    /* Testa se é um endereço válido - converte para formato binário */
    /* addr contém endereço em formato binário (struct in_addr) */
    if (!inet_aton(textIP, &addr)) {
      Log("Endereço IP inválido\n");
      return;
    }
  } else
```

```
    textIP= ip;  // Usa endereço IP que recebeu como argumento

  entry_Text= lookup_widget(GTK_WIDGET(data), "entryTexto");
  /* Lê texto a enviar */
  textText= gtk_entry_get_text (GTK_ENTRY(entry_Text));
  /* Testa texto */
  if ((textText==NULL) || !strlen(textText)) {
    Log("Texto vazio\n");
    return;
  }
  /* Cria e envia mensagem */
  /* NOTA PARA OS ALUNOS ACRESCENTEM O CÓDIGO PARA ENVIAR A MENSAGEM AQUI */
  Log("ENVIA MENSAGEM ... exercício ...\n");

  /* Regista envio na lista : */
  /* Obtém referência para lista */
  list= GTK_CLIST(lookup_widget(GTK_WIDGET(data), "clistEnvios"));
  /* Pára a actualização de janela */
  gtk_clist_freeze(list);
  /* Prepara nova entrada */
  time(&tbuf);
  nova_entrada[0]= g_strdup(textIP);
  nova_entrada[1]= g_strdup(ctime(&tbuf));
  /* Acrescenta entrada no inicio da lista */
  gtk_clist_insert(list, 0 /* primeira linha */, nova_entrada);
  /* Volta a refrescar ecran */
  gtk_clist_thaw(list);
}
```

Observe-se que os elementos da tabela são alocados usando a função `g_strdup`. Caso houvesse a necessidade de criar strings complexas a partir de vários elementos, poder-se-ia usar a função `g_strdup_printf` para alocar espaço e formatar uma string com uma sintaxe igual à função `printf`.

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

```
void
on_buttonEnvia_clicked                 (GtkButton       *button,
                                        gpointer         user_data)
{
  envia_mensagem_ipv4(user_data, NULL);
}
```

O evento "**clicked**" do botão "**buttonEnviaAt**" deve ser associado a uma função com um argumento "**Objecto**" igual a "`window1`". Esta função envia a mensagem com um atraso igual ao número de milisegundos indicado na caixa "`entrymSec`" usando um temporizador. O código para a função do temporizador e da função do evento deve ser inserido no ficheiro "`callbacks.c`":

```
gboolean callback_timer (gpointer data)
{
  // data - contém ponteiro para "main_window"
  envia_mensagem_ipv4(data, NULL);
  return FALSE;  // retira função do ciclo principal
  // Se retornasse TRUE continuava a receber invocações peridicamente
}

void
on_buttonEnviaAt_clicked               (GtkButton       *button,
                                        gpointer         user_data)
{
  // user_data - referência para Widget na janela
  GtkWidget *entry_atraso;
  const gchar *textAtraso;
  u_int atraso= 0;
  char *pt;
  guint id;

  entry_atraso= lookup_widget(GTK_WIDGET(user_data), "entrymSec");
  /* Obtém texto da entry box */
  textAtraso= gtk_entry_get_text(GTK_ENTRY(entry_atraso));
```

```
  /* testa se texto é válido */
  if ((textAtraso == NULL) || (strlen(textAtraso)==0)) {
    Log("Número de msegundos não definido\n");
    return;
  }
  /* Converte para inteiro (Help: escrever na linha de comando: man strtoul) */
  atraso= strtoul(textAtraso, &pt, 10);
  if ((pt==NULL) || (*pt)) {
    Log("Número de msegundos inválido\n");
    return;
  }
  /* Adia envio de mensagem - arma callback */
  id= g_timeout_add(atraso, callback_timer, user_data);
}
```

A declaração da assinatura da função `callback_timer` deve ser acrescentada ao ficheiro "`callbacks.h`":

```
gboolean callback_timer (gpointer data);
```

O grupo seguinte de funções destina-se a lidar com a lista de envios anteriores. 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 "`clistEnvios`" e reenviar uma mensagem para o IP nessa linha. Esta funcionalidade é obtida definindo a variável global "`linha_seleccionada`" no ficheiro "`callbacks.c`", que contém o número de linha seleccionado, ou –1 caso não esteja seleccionada. Esta variável vai ser actualizada pelas rotinas de tratamento dos eventos "**`select_row`**" e "**`unselect_row`**" de "**`clistEnvios`**", com o código apresentado em seguida:

```
static int linha_seleccionada= -1;  // Variável privada do módulo

void
on_clistEnvios_select_row              (GtkCList        *clist,
                                        gint             row,
                                        gint             column,
                                        GdkEvent        *event,
                                        gpointer         user_data)
{
  linha_seleccionada= row;  // Memoriza última linha seleccionada
}

void
on_clistEnvios_unselect_row            (GtkCList        *clist,
                                        gint             row,
                                        gint             column,
                                        GdkEvent        *event,
                                        gpointer         user_data)
{
  linha_seleccionada= -1;  // Nenhuma linha seleccionada
}
```

A rotina associada ao evento "**`clicked`**" do botão "**`buttonReenvia`**" (com um argumento "**`Object`**" igual a "`window1`") usa o valor da variável anterior para obter o endereço IP e enviar a mensagem:

```
void
on_buttonReenvia_clicked               (GtkButton       *button,
                                        gpointer         user_data)
{
  GtkCList *list;
  gchar *ip;

  if (linha_seleccionada == -1) {
    Log("Não há nenhuma linha seleccionada\n");
    return;
  }
  // Obtém referência para clist
  list= GTK_CLIST(lookup_widget(GTK_WIDGET(user_data), "clistEnvios"));
  /* Põe a actualização da lista */
  gtk_clist_freeze(list);
  // Obtém dados da linha
  if (!gtk_clist_get_text(list, linha_seleccionada, 0 /*coluna*/, &ip)) {
```

```
    /* Volta a refrescar lista */
    gtk_clist_thaw(list);
    Log("Não conseguiu obter última linha seleccionada\n");
    return;
  }
  /* Reenvia dados */
  envia_mensagem_ipv4(user_data, ip);
  /* Volta a refrescar lista */
  gtk_clist_thaw(list);
}
```

A rotina associada ao evento "**`clicked`**" do botão "**`buttonRetira`**" (com um argumento "**`Object`**" igual a "`window1`") percorre "`clistEnvios`" retirando todas as entradas iguais ao endereço IP na caixa "`entryRemIP`":

```
void
on_buttonRetira_clicked                (GtkButton       *button,
                                        gpointer         user_data)
{
  // user_data - entryEditIP com IP
  GtkWidget *entry_RIP;
  const gchar *textRIP;
  struct in_addr addr;
  GtkCList *list;
  gchar *ip;
  int i;

  /* Obtém referência para entryEditIP */
  entry_RIP= lookup_widget(GTK_WIDGET(user_data), "entryRemIP");
  /* Lê texto EditIP */
  textRIP= gtk_entry_get_text(GTK_ENTRY(entry_RIP));
  /* Testa se é um endereço válido - converte para formato binário */
  if (!inet_aton(textRIP, &addr)) {
    Log("Endereço IP a remover inválido\n");
    return;
  }
  // Obtém referência para clist
  list= GTK_CLIST(lookup_widget(GTK_WIDGET(user_data), "clistEnvios"));
  /* bloqueia refrescamento de lista */
  gtk_clist_freeze(list);
  i= 0;
  while (gtk_clist_get_text(list, i, 0 /*coluna*/, &ip)) {
    if (!strcmp(ip, textRIP))
      /* Retira linha */
      gtk_clist_remove(list, i);
    else
      i++;
  }
  /* Volta a refrescar lista */
  gtk_clist_thaw(list);
}
```

A rotina associada ao evento "**`clicked`**" do botão "**`buttonLimpa`**" (com um argumento "**`Object`**" igual a "`window1`") limpa o conteúdo de "`clistEnvios`" e de "`textview1`":

```
void
on_buttonLimpa_clicked                 (GtkButton       *button,
                                        gpointer         user_data)
{
  GtkCList *list;
  GtkTextView *textbox;
  GtkTextBuffer *textbuf;
  GtkTextIter tbegin, tend;

// LIMPAR CList
  /* Obtém referência para lista */
  list= GTK_CLIST(lookup_widget(GTK_WIDGET(user_data), "clistEnvios"));
  /* Pára actualização da lista */
  gtk_clist_freeze(list);
  /* Limpa lista */
  gtk_clist_clear(list);
  /* Volta a refrescar lista */
  gtk_clist_thaw(list);

// LIMPAR TextView
  textbox= GTK_TEXT_VIEW(lookup_widget(GTK_WIDGET(button), "textview1"));
```

```
  textbuf= GTK_TEXT_BUFFER(gtk_text_view_get_buffer(textbox));
  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 ainda editar o ficheiro "callbacks.c" acrescentando a lista de ficheiros a incluir:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "callbacks.h"
#include "interface.h"
#include "support.h"
```

Falta ainda modificar o ficheiro "main.c" de maneira a memorizar a janela principal na variável main_window (de forma semelhante à descrita na página 36) e a iniciar o socket.

Após compilar este código (usando o comando make na linha de comandos), terá um executável incompleto, que apenas escreve que envia as mensagens. Como primeiro exercício deverá completar o exemplo, acrescentando o código de inicialização do socket na função "main", e completando a função "envia_mensagem" de forma que o executável comunique com o servidor apresentado na secção 3.5.1.

3.5.3. Exercícios

1) Complete o código do cliente fornecido de maneira a que seja enviado um pacote datagrama IP através de um socket UDP. O pacote deve ter constituído por dois campos: um inteiro (short unsigned int) com o comprimento da mensagem no formato rede, seguido dos bytes da mensagem terminados pelo terminar de string (caracter '\0').
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 IPv4 multicast. **Sugestão:** Analise o código apresentado na secção 3.1.
4) 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.
5) Modifique o código do cliente e do servidor de maneira a passarem a trabalhar com sockets TCP, utilizando sub-processos para enviar e para receber dados através dos sockets. **Sugestão:** Analise o código apresentado nas secções 3.3 e 3.4.