REDES INTEGRADAS DE TELECOMUNICAÇÕES

2025 / 2026

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

4º ano / 1º ano

Introduction to the development of applications in Gnome 3: Programming dual stack applications using sockets and multicast addresses

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

Index

1. OBJECTIVE

Familiarization with the Linux environment and developing dual stack applications using sockets, the Gtk+3.0 / Gnome graphics library, the Glade-3 tool, and the Eclipse development environment for C/C++. This document includes an initial part describing the programming interface, followed by several example programs. This document describes the examples and the programming method in the development environment. To get the most out of this statement, it is suggested to analyze the code provided and complete the exercises in order to learn to use the tools and the environment.

2. APPLICATION DEVELOPMENT ENVIRONMENT FOR C/C++ IN LINUX

The Linux operating system includes the compilers 'gcc' and 'g++' that are used to develop applications, respectively, in programming languages 'C' and 'C++.' Various graphics libraries and environments may be used to create an application's graphical user interfaces. The two most common are KDE and Gnome, which are also linked to two different graphical environments of the Linux operating system. In the second laboratory work of RIT, the graphical library of Gnome 3, Gtk+3.0, will be used. The application development environment Eclipse is similar to the one used in NetBeans, functioning as an integrated environment (single interface) where interface design, code editing, compiling, and testing can be done. In RIT work, a separate application will be used to design the interfaces (`glade`), which run within the Eclipse graphical environment. To consult the manual for functions and libraries of the programming interfaces, you can use the man command on the command line, the application `DevHelp`, or Web pages can be consulted. Everything else can be done within the Eclipse integrated environment, although other code editors (e.g. netbeans for C/C++, kate, gedit, vi, etc.) could be used alternatively.

You start developing an application in the GUI editor that creates an XML file with the definition of the graphical elements and functions used to handle events (e.g. pressing mouse buttons or keys). Starting from a common C code base structure provided for programs with a Gtk+3 interface, the programmer must define the functions and graphical symbols and produce an XML file. In addition, the programmer must add the non-graphic variables (sockets, files, inter-process communication, etc.) and write the code to initialize variables and handlers of all events.

This section begins by introducing the socket interface, used to send UDP or TCP messages. Then interfaces for process management and inter-process communication (pipes, sockets, and signals) are introduced. The second part introduces application development using glade and the graphics library Gtk+3.0.

2.1. *Sockets*

When the sockets were introduced in Unix in the 70's, it was defined as a low-level interface for inter-process communication, which was adopted in virtually all operating systems. In previous curricular units of the 1st cycle course this interface was used indirectly through complex objects, which were abusively called **sockets**.

A socket allows us to offer a uniform interface to any protocol for communication between processes. There are several domains where we can create sockets. The AF_UNIX domain is defined locally to a machine. The domain AF_INET supports any transport-level protocol that runs over IPv4. The domain AF_INET6 supports any transport-level protocol that runs over IPv6

(or IPv4 with dual stack). A socket is identified by a file descriptor created by the `socket` function. The protocol is defined using two parameters: The first selects the type of service (reliable stream or datagram), and the second the protocol (0 specifies the default protocols: TCP and UDP). In the case of local sockets (AF_UNIX), one can use the function `socketpair` illustrated in section 2.1.8.

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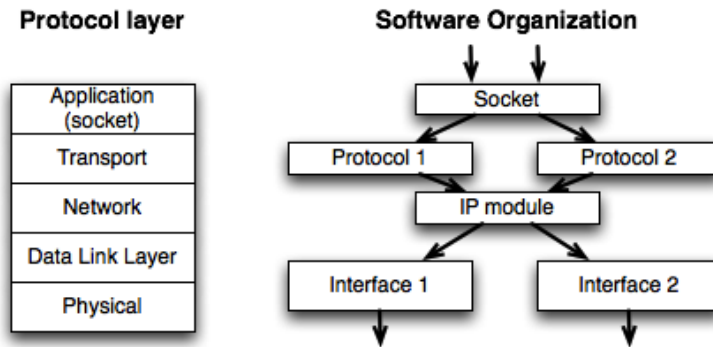
int socket (int domain, int type, int protocol);

Creates a port for asynchronous and bidirectional communication and returns a descriptor (identical to the ones used in files and pipes).

domain – universe where the socket is created, which defines the protocols and the namespace.
    AF_UNIX – Unix domain, local to a machine.
    AF_INET – Ipv4 domain, IPv4 Internet network.
    AF_INET6 – IPv6 domain, Ipv6 Internet, or dual-stack network.
type
    SOCK_STREAM – TCP socket.
    SOCK_DGRAM – UDP socket.
protocol – depends on the domain. Normally, it is set to zero, which indicates the default protocol in the respective domain (TCP, UDP).

By default, a socket has no port number assigned. The bind function associates a socket with a port number. The port number can be zero, meaning it is dynamically assigned by the system.

---

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

Associates a name to a socket already created.

s – socket number.
name – name depends on the area where the socket was created. In UNIX domain corresponds to a "pathname". In domains AF_INET and AF_INET6 are, respectively, the types struct sockaddr_in and struct sockaddr_in6, which are made by a machine address, protocol, and port number.
namelen – integer equal to `sizeof(*name)`

Example of port number assignment with a dynamic value for an IPv4 socket:

```
#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;  // Local address (0.0.0.0)
  name.sin_port = htons(0);           // Dynamic port number
  if (bind(sock, (struct sockadr *)&name, sizeof(name))) {
      perror("Error in port association");   …
  }
```

Example of port number assignment with a dynamic value defined by the system for an IPv6 or dual stack (IPv6+IPv4) socket:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
…
  struct sockaddr_in6  name;
  unsigned short int   port;
…
  name.sin6_family = AF_INET6;
  name.sin6_flowinfo= 0;
  name.sin6_port = htons(port);              // Port defined by the user
  name.sin6_addr = in6addr_any;              // Local IPv6 by default
  if (bind(sock, (struct sockaddr *)&name, sizeof(name)) < 0) {
      perror("Error in port association ");  …
  }
```

Like any other file descriptor, a socket is closed using the function `close`:

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

2.1.1. Datagram sockets (UDP)

Once created, a UDP socket is ready to receive messages using the functions `recvfrom`, `recv`, or `read`. These functions are blocking, except when a packet is waiting on the socket or if you select the option (flag) `MSG_DONTWAIT`. Sending messages is done using the function `sendto`.
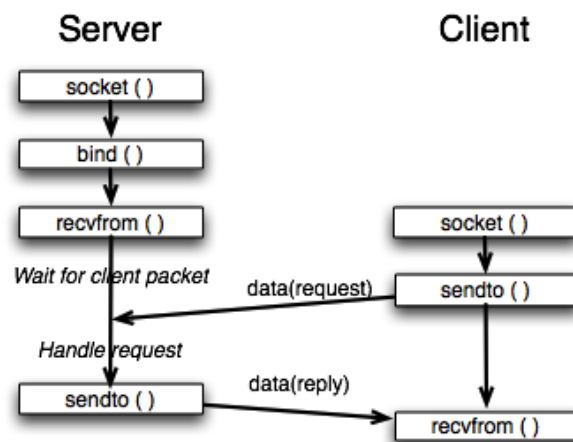

```
int recvfrom (int s, char *buf, int len, int flags, struct
     sockaddr *from, int *fromlen);
ou
int recv (int s, char *buf, int len, int flags);
ou
int read (int s, char *buf, int len);
```

Receives a message through socket s from a remote socket. Returns the number of
    bytes read or –1 in case of error.

`buf` – buffer for incoming messages.
`len` – buffer dimension.
`flags` :
    MSG_OOB   – Out of band;
    MSG_PEEK   – Read without removing messages from the socket;

```
        MSG_DONTWAIT        – Don't wait for messages.
```
from – socket address of the message sender (returned by the function).
fromlen – <u>pointer to an integer</u> initialized with `sizeof(*from)` (returns the bytes written).

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

Sends a message through socket `s` to the socket specified by `to`.

msg – message to send.
len – message length.
flags – 0 (no options)
to – address of the destination socket.
tolen – integer equals to `sizeof(*to)`

2.1.2. TCP sockets

With TCP sockets is necessary to establish a connection before they can exchange data. Participants play two different roles. A TCP socket server needs to prepare to receive the establishment of connections (`listen`) before it can receive a connection (`accept`). A TCP socket client needs to create a connection using the `connect` function. After connecting, data is received with the functions `recv` or `read` and sent using `send` or `write` functions.
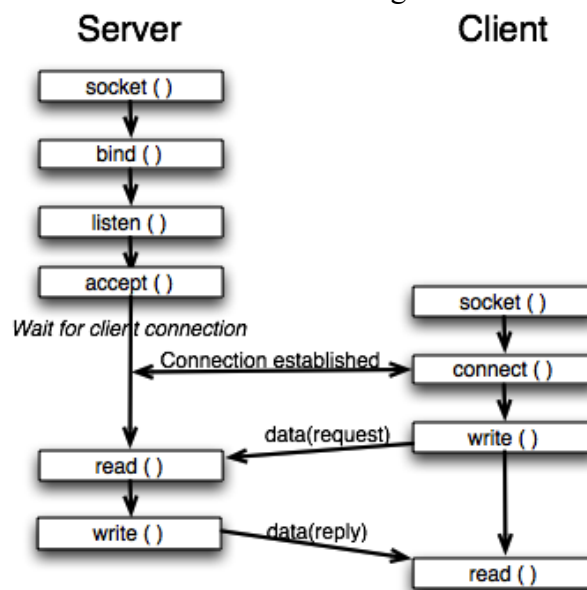

```
int connect (int s, struct sockaddr *name, int namelen);
```

Establishes a connection between the socket `s` and the other one indicated in `name`.

```
int listen (int s, int backlog);
     backlog - length of the queue for new connection requests.
```

Indicates that the *socket* `s` can receive connections.

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

Blocks the process until a remote one establishes a connection. It returns the identifier of the new socket created for data transfer on the new connection.

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

Sends a message through socket `s` to the remote socket associated. Returns the number of bytes effectively sent, or **-1** in case of error. In function *send*, the parameter *flags* can have the value MSG_OOB, meaning that data is sent out of band.

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

Receives a message from the remote socket by socket `s`. Returns the number of bytes read, **0** if the connection was cut or **-1** if the operation was stopped. In `recv` function, the `flags` parameter can have the values MSG_OOB or MSG_PEEK, respectively, meaning if you want to read data out-of-band or if you want to peek at the data without removing it from the buffer.

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

Let's close one of the directions for data transmission, depending on the value of `how`: `0` – only allow writing; `1` – only allow reading; `2` – closes both ways.

The TCP sockets can be used in blocking mode (default), where the operations of connection establishment, reading, or writing will block until data is available, or in nonblocking mode, which returns an error (EWOULDBLOCK) when they cannot be performed yet. The change of operating mode is made using the `fcntl` function:

```
fcntl(my_socket,F_SETFL,O_NONBLOCK);    // nonblocking mode
fcntl(my_socket,F_SETFL,0);             // blocking mode (by default)
```

2.1.3. Socket configuration

The socket interface supports configuring a wide range of parameters in the various protocol layers. Parameters can be read and modified respectively by the functions `getsockopt` and `setsockopt`.

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

The `setsockopt` function takes as arguments the socket descriptor (`s`), the protocol layer that will be set (`level` – e.g., SOL_SOCKET for the socket level and IPPROTO_TCP for TCP) and the identity of the parameter you want to configure (`optname`). The list of supported options for the IP level is defined in `<bits/in.h>`. The parameter type (`opt-val`) passed to the function depends on the option, being an integer type for most parameters. The function returns **0** on success.

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

The function `getsockopt` receives the same parameters and allows the values associated with the same options to be read.

Examples of parameters for UDP or TCP sockets are:
- `SO_RCVBUF` and `SO_SNDBUF` of layer `SOL_SOCKET` – specify respectively the buffer size for receiving and sending packets for TCP and UDP sockets;
- `SO_RCVTIMEO` and `SO_SNDTIMEO` of layer `SOL_SOCKET` – specify respectively the timeout for receiving and sending packets for TCP and UDP sockets;
- `SO_REUSEADDR` of layer `SOL_SOCKET` – allows several sockets to share the same port number at the same IP address;
- `TCP_NODELAY` of layer `IPPROTO_TCP` – controls the usage of Nagle's algorithm;
- `SO_LINGER` of layer `IPPROTO_TCP` – controls connection ending, preventing the socket to enter state TIME_WAIT.

For instance, to modify the sending buffer length, we would use:

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

2.1.4. IP Multicast

Any datagram socket can be associated with an IP Multicast, starting to receive all packets sent to that address. Packet transmission is performed in the same manner as for a unicast address. All settings to support IP Multicast are performed by activating various options with the setsockopt function.

2.1.4.1. Association to an IPv4 multicast address

Membership in a multicast IP address is performed using IP_ADD_MEMBERSHIP option at the IPPROTO_IP layer. The value of the IPv4 address must be class D (224.0.0.0 to 239.255.255.255). The address "224.0.0.1" is reserved, grouping all IP Multicast sockets.

```
struct ip_mreq imr;
if (!inet_aton("225.1.1.1", &imr.imr_multiaddr)) { /* conversion failed */; … }
imr.imr_interface.s_addr = htonl(INADDR_ANY);  /* Default network device */
if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *) &imr,
    sizeof(struct ip_mreq)) == -1) {
  perror("Failed association to IPv4 multicast address"); … }
```

The reverse operation is performed with the option IP_DROP_MEMBERSHIP, with the same parameters.

2.1.4.2. Association to an IPv6 multicast address

The association to an IPv6 multicast address is carried out using the option IPV6_JOIN_GROUP of layer IPPROTO_IPV6. The address must be a multicast address (ff00::0/8).

```
struct ipv6_mreq imr;
if (!inet_pton(AF_INET6, "ff18:10:33::1", &imr.ipv6mr_multiaddr)) {/*conversion failed*/}
imr.ipv6mr_interface = 0;  /* Interface 0 */
if (setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, (char *) &imr,
    sizeof(imr)) == -1) {
  perror("Failed association to IPv6 multicast address"); … }
```

The reverse operation is performed with the option IP_LEAVE_GROUP, with the same parameters.

2.1.4.3. Sharing the port number

By default, only one socket can be associated with a port number. Using the SO_REUSEADDR option of the SOL_SOCKET layer, you can share a port between multiple sockets, receiving all messages sent to this port at all sockets.

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

2.1.4.4. Defining the timeout for a read operation

A variable of type *struct timeval* is used to specify the timeout value on a read operation. The reading function (e.g., *read*, *recv*, *recvfrom*, etc.) stops after the time specified if a packet is not received or returns the data received until then if it is not enough. If nothing is received, the reading function returns -1, and the system variable errno has the value EWOULDBLOCK. The reading timeout can be canceled by defining a null timeout value.

```
struct timeval timeout;
timeout.tv_sec = 10;   // 10 seconds
timeout.tv_usec = 0;   // 0 microseconds
```

```
if (setsockopt (sock, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout, sizeof(timeout)) < 0) {
        error("setsockopt failed\n");
}
```

2.1.4.5. Defining the scope of a multicast group

For IPv4, the scope of a group is defined only when sending packets. The Time to Live (TTL) of a packet sent to an IPv4 Multicast can be controlled using the `IP_MULTICAST_TTL` option. The value 1 restricts the packet to the local network. Packets are only forwarded in multicast routers for TTL values greater than 1. In the MBone network, the scope was specified using the TTL value (<32 is restricted to the organization's network; <128 is restricted to the continent).

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

The equivalent option for IPv6 is `IPV6_MULTICAST_HOPS`, but it is less used because, in this case, the scope is defined by the address value. The IPv6 addresses have the following structure:

8	4	4	112 bits
11111111	flags	*scope*	Group Id

The flags have a set of 4 bits |0|0|0|T|, where only T is defined.

- T = 0 defines a permanent multicast address (see RFC 2373 e 2375);

- T = 1 defines a transient address vulgarly used in user applications.

The *scope* defines the range of the multicast group. The values are:

0 reserved	1 local to the node	2 link local	3 not defined
4 not defined	5 place	6 not defined	7 not defined
8 organization	9 not defined	A not defined	B not defined
C not defined	D not defined	E global	F reserved

2.1.4.6. Echo of data sent to the socket

With the option `IP_MULTICAST_LOOP`, you can control whether the data you send to the group is received in the local IPv4 socket.

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

For IPv6 the equivalent option is IPV6_MULTICAST_LOOP.

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

2.1.5. Auxiliary functions

A set of functions is used to assist the development of applications. These functions perform address translation between binary format (IPv4: `struct in_addr` and IPv6: `struct in6_addr`) and string (`char *`), get the port number associated with a socket, etc. You can find a comprehensive list of functions for IPv6 in RFC 3493.

2.1.5.1. Translation between address formats

There are two ways to identify a machine on the network:
- by its IP address (string or binary format) (e.g. "193.136.127.217" for IPv4, equivalent to "::ffff:193.136.127.217" for IPv6; or "2001:690:2005:10:33::1" for native IPv6);
- by the domain name of the machine (e.g. "tele1.deec.fct.unl.pt ").

Two functions were defined to translate IPv4 addresses between binary format (`struct in_addr`) and the string format:

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

The function `inet_aton` converts from the format string ("a"scii) to binary ("n"umber), returning **0** if it is not a valid address. The `inet_ntoa` function creates a temporary string with the address passed in the argument.

Later, two new functions that support IPv4 and IPv6 addresses and allow the conversion between binary format and the format string were added:

```
#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);
```

The function `inet_pton` converts from format string ("p"ath) to binary, returning **0** if it is not a valid address. The parameter `af` defines the type of address (AF_INET or AF_INET6). The `dst` parameter must point to a variable of type `struct in_addr` or `struct in6_addr`. The `inet_ntop` function creates a string with the contents of `src` in an array of characters passed in the argument `dst` of length `size`. The array must have a dimension bigger or equal to `INET_ADDRSTRLEN` or `INET6_ADDRSTRLEN` (for IPv4 and IPv6, respectively), two constants declared in *<netinet/in.h>*. Returns **NULL** on error, or `dst` when it accomplishes the conversion.

The translation of a machine's name or an IP address to binary format can also be achieved using the functions `gethostbyname` or `gethostbyname2`:

```
struct hostent *gethostbyname(char *hostname);              // only for IPv4
struct hostent *gethostbyname2(char *hostname, int af);     // IPv4 or Iv6
```

The program below shows a code excerpt that completes the structure `sockaddr_in` (IPv4), given the name or address of a machine and port number.

```
#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);
```

2.1.5.2. Get the local IPv4 or IPv6 address

You can obtain the local machine's IPv4 or IPv6 addresses using the functions above and the function `gethostname`, which reads the local machine's name. This function fills the buffer name received as an argument, returning **0** on success. This method fails when there isn't an entry in the DNS service or the file '`/etc/hosts`' with the domain name in the request (IPv4 or IPv6).

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

You can obtain the local IPv4 address from the name of the network device (e.g. "eth0") using the `ioctl` function.

```
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;
}
```

However, this function does not support IPv6 addresses. In this case, one possible solution is to invoke the `ifconfig` command (which returns all system interfaces) and apply filters to the resulting chain of characters in order to isolate the first global address of the list, creating a temporary file. The following function returns a string with a global IPv6 address in the buffer `buf` from name `dev` of the network device (e.g. "*eth0*").

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 carateres 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"); // Delete temporary file

  if (n <= 0) return FALSE;
  if (n >= 256) return FALSE;
  char *p= strchr(buf, '/');
  if (p == NULL) return FALSE;
  *p= '\0';
  return TRUE;  // Returns the IPv6 address in 'buf'
}
```

2.1.5.3. Get the port number associated with a socket

The `getsockname` function returns a structure that includes all the information about the socket, including the port number, IP address, and socket type.

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

Below a program where you get the port number associated with an IPv4 socket `s` is presented.

```
struct sockaddr_in addr;
int len= sizeof(addr);
...
if (getsockname(s, (struct sockaddr *)&addr, &len)) {
    perror("Error getting the socket name");  … }
if (addr.sin_family != AF_INET) { /* It is not IPv4 */ … }
printf("The socket has the port #%d\n", ntohs(addr.sin_port));
```

The equivalent code for IPv6 would be:

```
struct sockaddr_in6 addr;
int len= sizeof(addr);
...
if (getsockname(s, (struct sockaddr *)&addr, &len)) {
     perror("Error getting the socket name");  … }
if (addr.sin6_family != AF_INET6) { /* It is not IPv6 */ … }
printf("The socket has the port %d\n", ntohs(addr.sin6_port));
```

2.1.5.3. Waiting on multiple sockets in parallel

Most of the primitives described earlier to accept new calls and to receive data on a TCP or UDP socket block. To make applications that receive data from multiple sockets, keyboard, and mouse events, the select function was created, which allows parallel waiting to receive data from several file descriptors. Like almost all types of interaction can be described by a file descriptor, the function is used for almost all applications. The exceptions are the multi-tasking applications where there may be multiple active tasks in parallel, each treating a different socket.

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

This function takes as argument three arrays of bits, which indicate which file descriptors (associated with Input / Output protocols) it is waiting to receive data (readfds - input mask), where it is expecting to have space to continue writing (writefds - writing mask) and where it wants to receive signaling errors (exceptfds - exceptions mask). The field width must be completed with the highest descriptor value in the mask (plus one). This function blocks until it receives one of the events or until the expiration of the maximum waiting time (defined in timeout). Returns the number of events enabled, **0** if the timer has expired, or **-1** on error. The active events are identified by a bit in the masks passed in arguments. In timeout, the function returns the time that was lacking for the timeout to expire when the event was received. To deal with bit masks of the type fd_set, the following four functions are given:

```
FD_ZERO (fd_set *fdset)          // Sets all bits to 0.
FD_SET (int fd, fd_set *fdset)   // Turns on the bit corresponding to fd.
FD_CLR (int fd, fd_set *fdset)   // Turns off the bit corresponding to fd.
FD_ISSET (int fd, fd_set *fdset) // Tests if the bit corresponding to fd is set.
```

The following code illustrates the use of the select function to wait for two seconds on two socket's file descriptors in parallel:

```
struct timeval tv;
fd_set rmask;           // read mask
int sd1, sd2,           // socket file descriptors
    n, max_d;
FD_ZERO(&rmask);
FD_SET(sd1, &rmask);            // Set socket sd1
FD_SET(sd2, &rmask);            // Set socket sd2
max_d= max(sd1, sd2)+1;         // It could be all file descriptors, getdtablesize();
tv.tv_sec= 2;                   // seconds
tv.tv_usec= 0;                  // microseconds
n= select (max_d, &rmask, NULL, NULL, &tv);
if (n < 0) {
    perror ("Interruption of select");  // errno == EINTR if it was interrupted
} else if (n == 0) {
    fprintf(stderr, "Timeout\n"); …
} else {
    if (FD_ISSET(sd1, &rmask)) {// Data available for reading at sd1
        …
    }
    if (FD_ISSET(sd2, &rmask)) { // Data available for reading at sd2
        …
    }
}
```

The select function is at the base of systems that support event-based pseudo-parallelism and is at the core of some programming environments (e.g. Gnome/Gtk+ main cycle). In these environments, the function is used indirectly because Gtk+ allows registering functions to handle

events such as reading, writing, or handling exceptions in the main loop of the graphics library.

2.1.5.4. Obtaining the current time and calculating time intervals

There are several functions for getting the time (`time`, `ftime`, `gettimeofday`, etc.). The `gettimeofday` function returns the time with a precision of milliseconds. To calculate the time difference, we simply calculate the differences between the fields (`tv_sec` - seconds) and (`tv_usec` - microseconds) of two times, combining them.

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

2.1.5.5. Other functions

Most of the functions previously presented modify the value of the variable `errno` after returning an error. To write the contents of the error in the command line, you can use the `perror` function that takes as argument a string that concatenates before the description of the last error detected.

Another set of functions deals with sequences of arbitrary bytes and format conversion of binary integers:

Call	Description
`bcmp(void*s1,void*s2, int n)`	Compares the sequence of bytes, returning 0 if equal
`bcopy(void*s1,void*s2, int n)`	Copy n bytes from s1 to s2 (non-overlapped s1 and s2)
`memmove(void*s2,void*s1, int n)`	Copy n bytes from s1 to s2 (overlapped s1 and s2)
`bzero(void *base, int n)`	Fills with zeros n bytes starting from base
`long htonl(long val)`	Converts byte order of 32-bit integer from host to network
`short htons(short val)`	Converts byte order of 16-bit integer from host to network
`long ntohl(long val)`	Converts byte order of 32-bit integer from network to host
`short ntohs(short val)`	Converts byte order of 16-bit integer from network to host

The last four functions (defined in `<netinet/in.h>`) aim to allow portability of code to machines that use a representation of integers with a different ordering of bytes from the order specified for packets and arguments of the socket library routines. When an integer (s)hort (16 bit) or (l)ong (32 bits) is passed as an argument to a function of this socket library, it must be converted from format (h)ost (machine) to the (n)etwork format. When a parameter is received, the inverse conversion must be used: (n) to (h).

The 'man' command can be used to obtain more information on the command set presented.

2.1.6. Data structures

The names of the sockets are defined as specializations of the structure:

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

In the case of AF_INET domain sockets used on the Internet (IPv4), the `struct sockaddr_in` type is used, with the same number of bytes as the generic type. Because the C language does not support defining inheritance relationships between structures, explicit type changes (`struct sockaddr *`) are necessary to avoid warnings during compilation.

```
<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 */
};
```

For AF_INET6 domain sockets (IPv6), it is used the type `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 */
};
```

The structure `struct hostent` is returned by functions `gethostname` and `gethostname2` with a list of addresses associated with a host/DNS domain name.

```
<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
```

The structure `struct ip_mreq` is used during the association to IPv4 multicast addresses.

```
<bits/in.h>:
struct ip_mreq {
    struct in_addr  imr_multiaddr; // IPv4 multicast address
    struct in_addr  imr_interface; // IP unicast address of the network interface
};
```

`struct ipv6_mreq` is used during the association to IPv6 multicast addresses.

```
<netinet/in.h>:
struct ipv6_mreq {
    struct in6_addr    ipv6mr_multiaddr;  // IPv6 multicast address
    unsigned int       ipv6mr_interface;  // number of the interface (0 if there is only one)
};
```

The structure `struct time_val` is used in functions *select* and *gettimeofday*.

```
<sys/time.h >:
struct time_val {
    long tv_sec;       // Seconds
    long tv_usec;      // Microseconds
};
```

2.1.7. Concurrency based on threads

To increase parallelism in an application, it is possible to run several concurrent functions (if the processor has multiple cores) in C/C++. There are two alternative mechanisms: POSIX *threads*, similar to those used in Java; and concurrent processes. This section introduces the *pthread library*,

which implements the threads in Linux. The following section introduces the subprocesses.

2.1.7.1. thread creation

To increase parallelism in an application, it is possible to launch several concurrent tasks using the `pthread_create` function. Each *thread* is identified by an identifier of type `pthread_t` (in Ubuntu 18.08, and later, it is set to *unsigned long int*, but may vary in other implementations), returned in the first parameter (*thread*) of the function. The *attr* parameter allows you to configure thread parameters (scheduling policy, priority, stack size, etc.) – when using NULL, you set the values by default. The function run is specified in *thread_routine*, and the *arg* parameter is passed as the *thread_routine* function's argument.

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

The signature of a function running on a *thread* receives in *ptr* the value entered in *arg*, in the *pthread_create* call and returns a generic pointer. If you don't need to return anything, you can return NULL. Otherwise, you should return a pointer to dynamic memory (created with *malloc*) or to a global variable.

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

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

In the *thread_routine* function, you can terminate the *thread* at any point in the code using the `pthread_exit` function, indicating the returned value.

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

When calculated data needs to be collected on a thread, the `pthread_join` function is recommended. This function locks itself while waiting for the task with the *th* identifier to finish, collecting the value returned by the function in the *thread_return* argument. If no values are returned, you do not need to invoke this function.

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

It is possible to stop a *thread* using the `pthread_cancel` function, but it is not recommended. It is preferable to check in the *thread* code for a global variable value, which allows it to exit cleanly (without interrupting the execution of functions, leaving the global state incoherent).

To compile code that uses POSIX threads, you must use the `-lpthread` option in the compile and link command.

As with the threads used in Java, all share the same variables. If synchronizing access to non-shareable resources or coordinating the order of execution of threads is needed, additional mechanisms, presented in the next section, may be necessary.

2.1.7.2. Thread synchronization

In addition to waiting for the end of a *thread*, there are other mechanisms to control competition between *threads*. In this document, only the mutex locks are presented, which control exclusive access to code sections. For a quick introduction to other mechanisms, it is recommended to read external documents[1].

A mutex lock is created with the PTHREAD_MUTEX_INITIALIZER macro and saved in a

---

[1] You may find additional information about thread synchronization mechanisms at POSIX Threads Libraries (YoLinux.com) and Multithreaded programming (POSIX pthreads tutorial).

variable of type `pthread_mutex_t`. Any block of code that can only be accessed by one *thread* must be preceded by blocking the mutex lock with the *pthread_mutex_lock* function and unlocking it at the end with the *pthread_mutex_unlock* function, as shown in the following example.

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

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

It is recalled that mutex locks should be used carefully to avoid *deadlocks*. Tasks might be blocked while waiting for mutex locks that will not be released because of cross-dependencies.

2.1.8. Concurrency based on subprocesses

2.1.8.1. Subprocess creation

To increase the parallelism in an application, you can create multiple concurrent processes using the `fork` function. Unlike the tasks used in Java, **each process has a private copy of the variables**, making it necessary to use external channels (described in section 2.1.8.2) to synchronize the various processes. Each process is identified by an integer (*pid* - process id), which can be consulted on the command line with the instruction (`ps axu`).

When the `fork` function is invoked, the initial process is split into two, with exactly the same variables and the same files, pipes, and sockets opened. The value returned tells you whether it is the original process (returns the *pid* of the subprocess created) or the child process (returns **0**). In case there is no memory for the creation of subprocess returns **-1**.

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

The child processes run in parallel with the parent process but only die completely after the parent process invokes a variant of the function `wait` (usually `wait3`, or `wait4` when the father is blocked waiting for the end of a subprocess). Before that, they get into a *zombie* state.

```
#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);
```

Function `wait3` blocks by default unless the `options` parameter equals `WNOHANG`. In this case, returns -1 if no subprocess zombie is waiting. After the termination of a child process, a `SIGCHLD` signal is generated in the parent process. You can prevent the parent process from blocking until processing this signal, and indirectly, detecting faults in subprocesses. The functions return the `status` parameter, which allows us to detect if the process ended normally invoking `_exit`, or ended with an error (which generates a signal associated with an exception). The example of Section 3.5 illustrates how one can accomplish this functionality.

2.1.8.2. Inter-process synchronization

In addition to the sockets, it is possible to use various other types of synchronization mechanisms among local processes. When using subprocesses, it is common to use *pipes* or local sockets to communicate between the parent process and the child processes. A pipe is an **unidirectional channel** local to a machine similar to a TCP socket - all that is written in the

descriptor `p[1]` is sent to the descriptor `p[0]`. The `pipe` function creates two file descriptors (equivalent to sockets). If the pipe is created before invoking the `fork` operation, it is known for both processes. To maintain an open channel for communication between a parent process and the child process, they only have to close one end (each) and communicate between them through the channel created using the instructions `read` and `write`. If you intend to have **bidirectional communication**, you can use `socketpair` to create a pair of sockets.

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

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

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

In addition to being used for reporting asynchronous events (dead subprocess, data out of band, etc.), the signals may also be used in the communication between processes. Two signals are reserved for that purpose (SIGUSR1 and SIGUSR2), which can be generated using the function `kill`. The use of signals is illustrated in the example in Section 3.5.

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

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

Signal reception is carried via a *callback* function (e.g., `handler`) associated with a signal using the `signal` function. At a signal, the operating system interrupts the application and runs the callback function code, returning to the point where it stops. Blocking read calls are interrupted, returning an error (-1) and setting `errno==EINTR`.

```
#include <signal.h>

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

2.1.9. Read and write binary files

You can read and write binary files using file descriptors (`int`) or "`FILE *`" descriptors. In the example provided below for copying files two FILE descriptors are used: one for reading (`f_in`) and one for writing (`f_out`). A file is opened using the `fopen` function, which takes as its argument the open mode: reading "`r`", writing at the beginning of the file "`w`", writing at the end of file "`a`", or in one of several modes of reading and writing ("`r+`", "`w+`" or "`+`"). Reading is performed with the function `fread`, which specifies the size of each element and the number of elements to be read and returns the number of elements read. In the example, the aim is to read byte by byte, so the size of each element is 1. The value returned by `fread` function can be: **> 0**, indicating that read data successfully; **= 0**, indicating that it has reached the end of the file; **< 0**, indicating an error in accessing the file. The writing is performed with the function `fwrite`, which also identifies the size of the elements and the number of elements to be written with the same meaning. This function returns the number of elements written, or **-1** if the write operation failed. The file descriptors should be closed with `fclose` operation to ensure that all content written is effectively copied into the file system – otherwise, data may be lost.

```
#include <stdio.h>

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

```
  if ((f_in= fopen(from_filename, "r")) == NULL) {
    perror("error opening file for reading");
    return FALSE;
  }
  if ((f_out= fopen(to_filename, "w")) == NULL) {
    perror("error opening file for writing");
    fclose(f_in);
    return FALSE;
  }
  do {
    n= fread(buf, 1, SND_BUFLEN, f_in);
    if (n>0) {  // Read n bytes
      // n bytes were read to buffer 'buf'
      if ((m= fwrite(buf, 1, n, f_out)) != n) {
        // Failed writing
        break;
      } // else: else: If n==0; reached the end of file; If n==-1 occurred a reading error
  } while (n>0);
  fclose(f_in);
  fclose(f_out);
  return (n==0) || (n==m);
}
```

2.1.10. Timers outside the graphical environment

There are several alternatives to implement timers in a program in C/C++ outside a graphical environment that already supports this feature. An alternative is to use the `timeout` parameter of function `select`, described on page 12. Another alternative is to use the `alarm` function to schedule a SIGALRM signal generation with an accuracy of seconds. The `alarm` function is used both to set a timer (if `seconds>0`) and to disarm it (if `seconds==0`).

2.2. Applications with a Gtk+/Gnome GUI

In an application in text mode (no GUI) the programmer writes the main routine (`main`) controlling the sequence of actions in the program. In an application with a graphical interface, the application is built from a GUI and a set of functions that handle the graphical events, and if any, other asynchronous events. In this case, the `main` function starts with the various objects ending with an invocation to function `gtk_main()`, which starts a thread that runs an infinite loop waiting for graphics interactions, timers, or events from any other file descriptor. Internally, the Gtk+ implements this using the `select` function.

2.2.1. Graphical user interface editor Glade-3

The graphical user interface (GUI) editor **Glade-3** (command `glade-3` (Fedora) or `glade` (Ubuntu)) can be used to draw the GUI for applications on Gnome using multiple programming languages (C, C + +, Python, etc.). This application is included in the Fedora or Ubuntu distributions of Linux. There are also other Linux distributions, and more recently, it has been ported to other operating systems.

Unlike the approach used in the previous version (Glade-2), Glade-3 does not generate code 'C'; the graphic editor is limited to generating an XML file that is loaded when the application starts using an object `GtkBuilder` of the Gtk+ library. The code 'C' is then completely generated by the user. This document proposes a methodology to implement this code.

Developing a new project begins with defining the graphical interface using *glade*. When a new project is opened, an integrated environment appears, represented below, which is composed of three main elements: in the center at the bottom, there is the window that is being developed, and at the top, a set of buttons that allows access to four palettes of components that can be used

in the design of windows; On the left, there is a tree view of the defined components (e.g. windows), and on the right there is a window for editing the properties of the selected object. In addition, it includes the usual save/open buttons for interface specification files in XML files with the '.glade' extension. The figure shows the configuration window used in the demonstration program demoserv presented in section 3.6.1 (page 37).
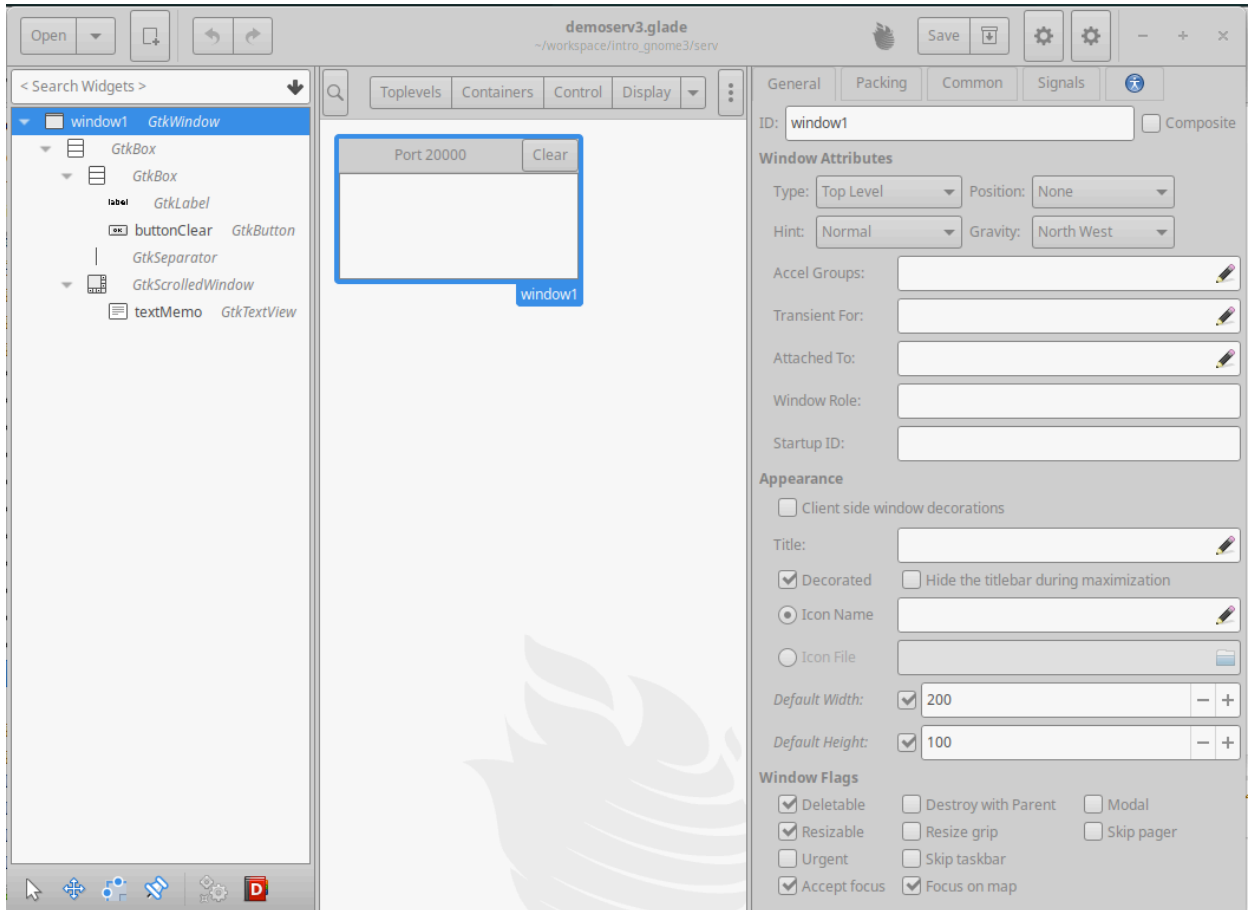
The graphic components organized into four groups are visible by selecting the buttons in the central window. You can find comprehensive information about all components (functions, callbacks, etc.) using the DevHelp app directly from Glade-3 by selecting the icon. Only a subset of these components will be used for the applications developed in the course, which is summarized below[2].

All interfaces are drawn from a basic component (usually
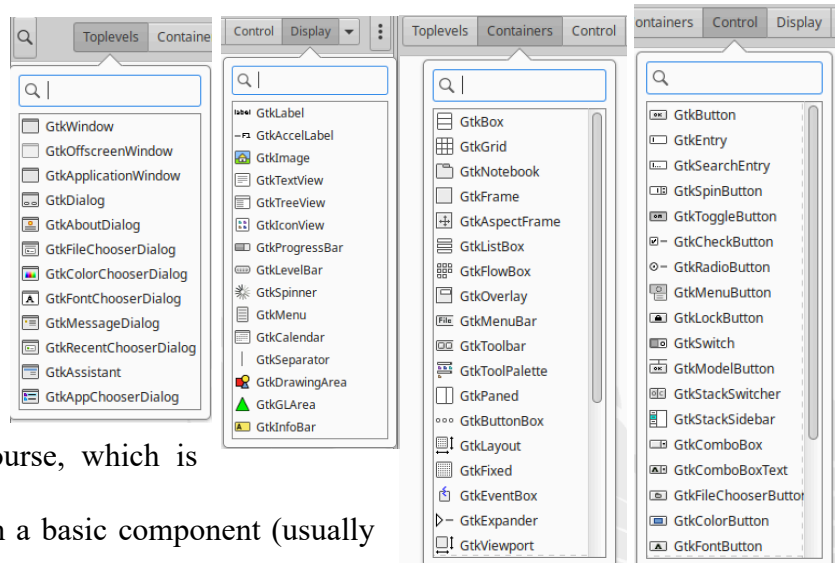
`GtkWindow` or `GtkDialog`). All examples provided in this document were developed using `GtkWindow` (▢).

By default, a `GtkWindow` only supports one graphical component therein. To have more than one component, it is necessary to use one or more structural components that subdivide the window into several boxes. Some of these components (▤ ⊞ ▯) allow, respectively, the division into columns or rows of the window, the window split into a matrix, and the division into folders. Others allow arbitrary positions in the window but do not allow them to automatically handle window scaling.

Once subdivided the window, other graphical components may be placed in each box. The graphical components used in the examples of this document (page `Control and Display`) were:

▤	`GtkTextView`	Editable box with multiple lines, usually used within a `GtkScrolledWindow`	▦
label	`GtkLabel`	Titles	
▭	`GtkEntry`	Editable text box with a single line	
OK	`GtkButton`	Button	
ON	`GtkToggleButton`	Button with state	

It was also used a graphical component to perform more complex tables:

▤ `GtkTreeView` (also with `GtkScrolledWindow`) – Viewer of data in trees or tables, where data is stored in a list of types GtkListStore ▤ , defined in folder `Miscellaneous`. `GtkCellRenderer` type derivative objects are used to interconnect the data on the list with the viewer objects, keeping the displayed data consistent with the data in the database, as shown in the figure on the demonstration program `democli` presented in section 3.6.2 (page 41). You can find a video with a tutorial at https://www.youtube.com/watch?v=nJloKxELPgI.

▼ ▤ treeview1	GtkTreeView	
⚡ treeview-selection1	GtkTreeSelection	(i
▼ ▤ treeviewcolumn1	GtkTreeViewColumn	
▤ cellrenderertext1	GtkCellRendererText	
▼ ▤ treeviewcolumn2	GtkTreeViewColumn	
▤ cellrenderertext2	GtkCellRendererText	

Like other integrated editors, you can edit the initial values of the components' properties using the graphics window at the right. The case shown illustrates some properties of the object window1 of type `GtkButton`. This window has five pages. The first page, **General**, contains the definition of the name of the object graph (`ID:`), and several other properties specific to each object (e.g. The initial window size at `Default width` and `Default Height`). For a `GtkEntry`, we can define whether it is editable, if the text is visible, what the initial text is, the maximum length of the text, etc. The page **Common** controls graphical aspects like the length and width of the box. The page **Packing** controls the relative position of the component when it is in a row or a column. In the example, the component "`buttonClear`" is in the second position of the second GTK box; we can change the relative position by changing the value of "`Position:`".

The page **Signals** allows you to assign functions to the events supported by the graphical components. For each component various functions can be associated to different events (termed "**Signals**"). The "`Handler`" field has the name of the function, which must be created in the

code developed. The "User data" allows passing an argument in the function invocation with a reference to a graphic component. In the example, a pointer is passed to the window "window1", but it could be any other graphical component. The function prototype can be obtained by consulting the *DevHelp* ([D]). The signal ⚠ is associated with

events that are not supported by all versions of Gnome or are obsolete.

Glade-3 creates an XML file with the specification of graphical components with the extension ".glade" (e.g. `example.glade`). The C code developed will have to start the graphical window, include all event processing functions specified in the XML file, and have a `main` function that runs the main loop of Gnome. To facilitate interaction with graphical objects, it is recommended that you create a file `gui.h`, which defines a structure with pointers to all the graphic elements that will be accessed in the program (e.g., `GtkEntry` with input or output parameters, `GtkTreeView` with tables, etc.), a function to initialize the window, and the declaration of all other functions used in the GUI.

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

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

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

The function `init_app` should be specific to each interface. It must initialize all pointers of the structure created earlier after loading the XML specification. These pointers can only be obtained during the window creation using the function `gtk_builder_get_object` and the name of the graphic object defined with *glade*. Explicit type conversion (e.g. GTK_ENTRY) is used, which, in addition to casting the type, also checks if the referenced graphic object is compatible.

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

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

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

        /* connect signals, passing our TutorialTextEditor struct as user data */
```

```
        gtk_builder_connect_signals (builder, window);

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

     // Do other initializations …

        return TRUE;
}
```

Finally, the `main` function initiates the structure (after pre-allocating the memory) and shows the window. It starts a thread in the `gtk_main` function, which performs the main loop where the graphical system processes the events. This thread is only released after running the `gtk_main_quit` function, which ends the program.

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

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

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

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

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

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

The developed code is compiled using the command `pkg-config` to get the folders pathnames and symbols specific to each distribution. To compile the file `main.c` to generate the binary `app` the command would be:

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

In Ubuntu, Glade-3 is available through the command line "`glade [project name].glade`" or once configured via the Eclipse integrated environment. Several documentation files (including the manual and FAQ) are available in *DevHelp*.

Two examples of programs developed using Glade-3 are presented starting on page 35, in sections 3.6 and 3.7.

2.2.2. Auxiliary functions

To develop an application in Gtk+/Gnome, several auxiliary functions for accessing graphical objects are necessary. Additionally, there are functions to handle the active socket descriptors in the main loop and to work with strings, lists, etc. The description of this set of functions is available through the application DevHelp. Two examples of programs that use some of the features are also provided. This section overviews some functions that handle poorly documented aspects of this library.

2.2.2.1. Gnome data types (glib) – lists (GList)

Gnome redefines a set of basic types (`int`, `bool`, etc.) to equivalent types with a name

prefixed with (g): `gint`, `gboolean`, etc. Additionally, Gnome defines several structured types, including the **type `GList`, which defines a list**. A list begins with a NULL pointer (for `GList`) and can be manipulated with the following functions:

```
GList *list= NULL;                               // The list begins with NULL pointer
GList* g_list_append(GList *list, gpointer data);    // Appends 'data' to the list
GList* g_list_insert(GList *list, gpointer data, gint position);
                                 // Adds 'data' to the list at 'position'
GList* g_list_remove(GList *list, gconstpointer data);          // Removes element
void g_list_free (GList *list);      // Frees the list; does not free the elements' memory
guint g_list_length (GList *list);         // list length
GList* g_list_first (GList *list);         // Returns the first element of the list
GList* g_list_lastGList *list);            // Returns the last element of the list
GList *g_list_previous(list);              // Returns the element before or NULL
GList *g_list_next(list);                  // Returns the element after or NULL
GList* g_list_nth(GList *list, guint n);   // Returns the n-th member of the list

// The elements' data is obtained using the field data using a cast: (Type)pt->data
```

2.2.2.2. Functions for manipulating strings

The Gnome library duplicates many of the functions of <string.h>, for example, the function `g_strdup`. If you need to create complex strings from multiple elements, you can use the function `g_strdup_printf` to allocate space and format a string with a syntax like the `printf` function.

2.2.2.3. Accessing graphical objects

Access to each graphic object is accomplished via specific interface functions. For example, for the text within the text box `entryIP` mentioned previously, we would use the following function:

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

The writing operation would be:

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

2.2.2.4. Ending an application

A program ends when the function `gtk_main_quit()` is invoked, ending the main cycle.

By default, closing all program windows does not end the program. To ensure this happens, you must associate a function to the event "`delete_event`" in the main window that returns FALSE. The content of the event handler function can be:

```
gboolean
on_window1_delete_event                  (GtkWidget        *widget,
                                          GdkEvent         *event,
                                          gpointer          user_data)
{
  /* Close all application specific data */ …
  gtk_main_quit();
  return FALSE;
}
```

2.2.2.5. External events – sockets and pipes

Gnome allows registering event handling functions to read, write, or exceptions of sockets in the main cycle through a channel descriptor (type `GIOChannel`). The functions are registered with the function `g_io_add_watch`, indicating the type of event you want. An example of association of a function `callback_data` to read events (`G_IO_IN`), writing (`G_IO_OUT`) and exceptions (`G_IO_NVAL`, `G_IO_ERR`) to a socket descriptor `sock` would be:

```
GIOChannel *chan= NULL;      // socket's channel descriptor
guint chan_id;               // channel number
…
if ( (chan= g_io_channel_unix_new(sock)) == NULL) {
```

```
      g_print("Failed IO channel creation\n"); …
}
if (! (chan_id= g_io_add_watch(chan,
       G_IO_IN|G_IO_OUT|G_IO_NVAL|G_IO_ERR, /* after read, write or error events */
       callback_dados /* callback function */,
       NULL /* parameter passed to the callback function */)) ) {
    g_print("Failed activation of data reception \n"); …
}
```

The function `callback_data` should have the following structure:

```
gboolean callback_dados (GIOChannel *source, GIOCondition condition, gpointer data)
{
  if (condition == G_IO_IN ) {
    /* Receives the data …*/
    return TRUE; /* the callback function remains active */
  }else if (condition == G_IO_OUT ) {
    /* There is space in the socket to continue writing data …*/
    return TRUE; /* the callback function remains active */
  } else if ((condition == G_IO_NVAL) || (condition == G_IO_ERR)) {
    /* Handles error … */
    return FALSE; /* Stops receiving events */
  }
}
```

You can turn off the callback function associated with an event using the function `g_source_remove` using the channel number as an argument.

```
/* Removes the socket from Gtk's main loop */
g_source_remove(chan_id);
/* Frees the channel */
g_io_channel_shutdown(chan, TRUE, NULL);// Ends the connection after draining the buffer
g_io_channel_unref(chan);          // Decrements the number of channel users;
                                   // The memory is freed after reaching zero references
```

2.2.2.6. External events – timers

Gnome allows arming timers that invoke a function periodically. A timer is started using the function `g_timeout_add`:

```
guint t_id;
t_id= g_timeout_add(atraso,   /* Number of miliseconds */
             callback_timer,/* callback function */
             NULL); /* Argument of the callback function */
```

The timer callback function must have the following signature:

```
gboolean callback_timer (gpointer data)
{
  // data – parameter defined in g_timeout_add
    return FALSE;  // cancels the timer
or  return TRUE;   // keeps calling the timer periodically
}
```

The timer can be canceled using the function `g_source_remove` with the value of `t_id` as the parameter.

2.2.3. Using threads in applications with GUI

To access glib library functions from concurrent threads, it is necessary to use the parallelism management mechanisms provided by the GDK (Gnome Dev. Kit), described in https://gtk.developpez.com/doc/en/gdk/gdk-Threads.html, which allows running the graphical environment with a dedicated thread and coordinating access to the graphical environment.

The first method[3] is based on acquiring a lock before accessing the graphical interface when the invocation is made from a different thread or after receiving an asynchronous event. In the

---

[3] For more information see https://gtk.developpez.com/doc/en/gdk/gdk-Threads.html

case of synchronous actions triggered from the graphical interface (e.g., pressing a button), these already occur in the context of the graphical thread. Any attempt to acquire the lock leads to the application being blocked (as it was acquired before processing the button).

It is necessary to initialize the graphics thread, calling `gdk_threads_init` in the `main` function.

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

All access to the glib and Gtk+3 library functions must be made after blocking the glib lock with the `gdk_threads_enter` function, and the lock must be unlocked with the `gdk_threads_leave` function after the operation.

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

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

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

This lock-based method was recently considered deprecated. Instead, a method based on registering a function called by the glib thread to invoke the methods of the glib and Gtk+3 libraries and update the graphical interface was proposed. In this way, these callback functions are called in the context of the graphics thread. To register a graphical callback function, the `g_idle_add` function is used.

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

To be able to pass more than one argument to the callback function, it is necessary to define a structure with all the necessary data, allocate and initialize a variable of that structure, and pass the pointer in the data parameter. In the callback function (worker, in the following example), you must retrieve the initial pointer and run the graphic code. Ultimately, the value the function returns determines whether the callback remains active (TRUE) or disconnected (FALSE).

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

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

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

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

The GUI update is triggered after activating the callback and waiting for the `worker` function to be invoked – with the disadvantage of losing control over the order in which the callback functions are called.

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

In this work, we will use the approach based on acquiring the lock before accessing the functions of the Gtk+3 library.

2.2.4. Using subprocesses in applications with GUI

Subprocesses can be used in applications with GUI, as long as **only one process** writes in the

graphical interface. After the `fork` operation, all processes share the same connection to the X graphical server, which only accept commands sequentially numbered. The solution to achieve this integration is to keep the parent process as responsible for graphic interaction, using the IPC mechanisms to support communication between the parent process, and the child subprocesses. The pipes and AF_UNIX sockets are described by file descriptors, so callbacks can be associated.

The following code illustrates the reception of data from the subprocess. You can use the `ptr` parameter to identify the subprocess sending (in this example the *process id* is passed as argument when registering the callback). One can also obtain the descriptor of the pipe, with the code represented.

```
/* Regists socket */
gboolean callback_pipe(GIOChannel *chan, GIOCondition cond, gpointer ptr) {
  int pid= (int)ptr;          // PID of the process sending data
  int p= g_io_channel_unix_get_fd(chan); // Get the file descriptor of the pipe
  // See the structure of the callback_data function in pages Erro! Marcador não definido. and
       Erro! Marcador não definido..
  // You cancel the callback with 'return FALSE;'// Keep it alive with 'return TRUE;'
}
```

The code of the child process is called in the subprocess starting function, which follows the structure shown above, in sections 2.1.7 and 2.1.8, and is exemplified below in example 3.4. A major change occurs in the code of the parent process, which can not be blocked. Thus, you should register the callback data processing of the subprocess.

```
gboolean start_subprocesso( … ) {
 int n;              // subprocess pid
 int p[2];           // pipe descriptor (or AF_UNIX socket)
 guint chan_id;      // Gtk+ channel number
 GIOChannel *chan;   // Gtk+ channel (device) descriptor

 pipe(p);
 n= fork(); // Starts the subprocess
 ...
 if (n == 0) {
     /******* SON's process *****************************************************/
     ...    // writes to p[1]
     _exit(0); // Ends the subprocess
     /******* End of SON's process *********************************************/
 }
 /******* FATHERS's process ****************************************************/
 fprintf(stderr, "Son process started with pid %d\n", n);
 close(p[1]);    // Father uses p[0]
 if (!put_socket_in_mainloop(p[0], (void *)n/*sets the parameter equal to pid*/, &chan_id,
     &chan, callback_pipe)) {
   Log("failed registration of pipe's callback in Gtk+ loop\n");
   return FALSE;
 }
 ...
}
```

2.3. Eclipse integrated environment for C/C++

The Eclipse development environment simplifies the development of C/C++ applications at the editing and debugging levels. The indexing engine allows you to see where a function or variable is declared, greatly facilitating the visualization of existing code. The integrated debugger allows you to display the values of a variable by positioning the mouse over it or to insert breakpoints. The environment is especially suited for C and C++.

In case Eclipse does not recognize `glade-3` as the editor of ".*glade*" files, to support integrated edition, you can add the application "*/usr/bin/glade*" (for Ubuntu) to the extension ".*glade*" in the menus *"Window"; "Preferences"; "General"; "Editors"; "File Associations"*.

One problem that sometimes occurs in large projects is a lack of memory. By default, eclipse starts with 256 MBytes of memory, but it can be increased by defining a command file to start

eclipse with the following contents:

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

To run an application in the debugger is necessary to add the application to the application menu. From the menu "*Run,*" choosing "*Debug ...*" opens a window where a list of "*Configurations*" appears on the left. In this list, you should create a new local application, "C/C++ Local Application." In the "*Main*" window, you should pick up the project and the application executable; in the "*Debugger*" you must choose "*GDB Debugger*" that runs the application `gdb`.
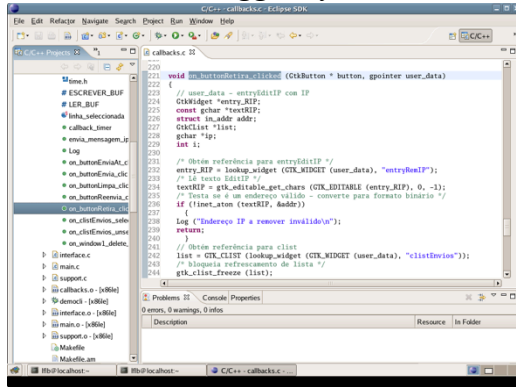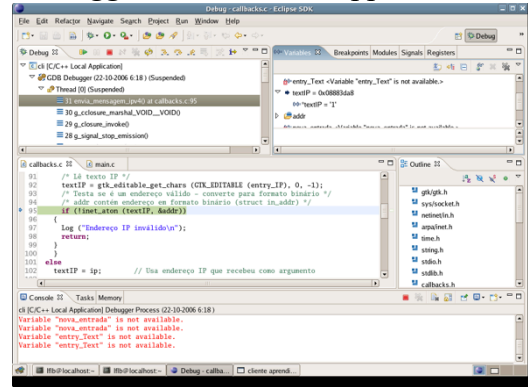
(a) File edition   (b) Debugging

The two images above illustrate the graphical aspect of Eclipse in edit mode and Debug mode. In the first case (a), we have access to the file list and the list of functions, variables, and inclusions per file. The "*Problems*" section lists hyperlinks to environmental problems identified in the code. In the debugging window (b), we can view the call stack of functions (*Debug*) and the values of variables after reaching a breakpoint (*breakpoint*).

2.4. Linux configuration to run dual-stack multicast applications

The latest Linux systems support IPv6 at the NetworkManager service, commonly associated with the network icon in the upper right corner of the screen. To enable IPv6 support, simply access the application's graphical network configuration. Within the application, you must configure the network device you are using (usually the 'eth0', but it can also be 'ens33', 'wlan0', etc. – you can use the command 'ifconfig' to see the devices list), and enter the property editing, selecting the option to support IPv6, configuring the IPv6 address statically in the case of being in an IPv4 network only. It is suggested that an address be used in the range 2001:690:1fff:bb::X/120, where X may have a value between 1 and ffff. If you want to test a real IPv6 network, you can see the page https://wiki.ubuntu.com/IPv6 to learn how to configure tunnels for the IPv6 Internet.

The system's firewall can block the application's operation. In this case, you can temporarily disable the firewall with the command *"iptables –F"* (IPv4) and *"ip6tables –F"* (IPv6) or add rules for the application's ports to be accepted.

3. APPLICATION EXAMPLES

In this section, five examples of client-server applications using sockets are provided. Sections 3.1 and 3.2 describe applications with UDP sockets implemented without a GUI, respectively, for IPv4 and IPv6. Section 3.3 describes an application with TCP sockets without a GUI. Section 3.4 describes an application with POSIX threads. Section 3.5 describes an

application with several subprocesses. Section 3.6 describes the development of a graphical interface and its integration using sockets UDP. Finally, section 3.7 describes the modifications to the example in section 3.6 for integrating TCP sockets. All the source code presented in this section is provided as an eclipse project.

3.1. Client and Server for IPv4 UDP Multicast in text mode

The application programming in text mode boils down to the transcript of the code using a text editor. In this example, the server starts at port 20000, joins the multicast IPv4 address "225.1.1.1", and is blocked waiting to receive a message. The message can be received either via IP Multicast or Unicast address of the local machine.

The code of "servv4.c" is:

```c
#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 port= 0; /*  The port is defined by the system */
  int  reuse= 1;
  /* Multicast */
  struct ip_mreq imr;
  char loop = 1;
  /* reception */
  struct sockaddr_in fromaddr;
  int fromlen= sizeof(fromaddr);

  /* Creates the socket. */
  sock = socket(AF_INET, SOCK_DGRAM, 0);
  if (sock < 0) {
    perror("Error opening datagram socket");
    exit(1);
  }
  if (argc == 2) { /* A port number was received as a parameter */
    port= (short int)atoi(argv[1]);
  }
  /* Makes the port number sharable for this IP – allows having more than one servers with the
      same port in the same machine */
  if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *) &reuse,
      sizeof(reuse)) < 0) {
    perror("Failed setsockopt SO_REUSEADDR");
    exit(-1);
  }
  /* Associates the socket to a port */
  name.sin_family = AF_INET;
  name.sin_addr.s_addr = htonl(INADDR_ANY);  // Local IP (0.0.0.0)
  name.sin_port = htons(port);
  if (bind(sock, (struct sockaddr *)&name, sizeof(name)) < 0) {
    perror("Binding of datagram socket failed");
    exit(1);
  }
  /* Multicast configuration */
  if (!inet_aton("225.1.1.1", &imr.imr_multiaddr)) {
    perror("Failed conversion of multicast address");
    exit(1);
  }
  imr.imr_interface.s_addr = htonl(INADDR_ANY);  /* Default network device */
  /* Join group */
  if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *) &imr,
      sizeof(struct ip_mreq)) == -1) {
    perror("Failed association to IP multicast group");
    abort();
```

```
    }
    /* Configure socket to receive echo of multicast packets sent */
    setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));

    /* Finds out the socket's port number defined by the system */
    length = sizeof(name);
    if (getsockname(sock, (struct sockaddr *)&name, &length)) {
      perror("Error getting the port number");
      exit(1);
    }
    printf("The socket at address IP Multicast 225.1.1.1 has port %d\n", htons(name.sin_port));
    /* Read a message from the socket */
    if (recvfrom(sock, buf, 1024, 0/* By default blocks*/, (struct sockaddr *)&fromaddr, &fromlen)
       < 0)
      perror("Error receiving data packet");
    printf("Received from %s:%d -->%s\n", inet_ntoa(fromaddr.sin_addr),
      ntohs(fromaddr.sin_port), buf);
    printf("The server is going down\n");
    close(sock);
}
```

The client's code is comparatively simpler. The client merely creates a socket, sets the destination's IP and port, and sends the message. Note that in addition to setting the TTL sent in the packet, nothing varies when sending a packet to an IPv4 Multicast or Unicast address.

The code of "cliv4.c" is:

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

/* Here a datagram is sent to a receiver defined in the command line */

#define DATA "Hello world!" /* Constant message */

main (int argc, char *argv[])
{
    int sock;
    struct sockaddr_in name;
    struct hostent *hp;
    u_char ttl= 1; /* send only to the local network */

    /* Creates the socket */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
      perror("Error opening datagram socket");
      exit(1);
    }
    /* Prepares name, of the destination socket. Gethostbyname returns a structure that includes
        the destination IP address, accepting "pc-1" or "10.1.55.1". With the second class of
        addresses inet_aton function could also be used. The port is obtained from the command line
        */
    if (argc<=2) {
      fprintf(stderr, "Usage: %s ip port\n", argv[0]);
      exit(2);
    }
    hp = gethostbyname(argv[1]);
    if (hp == 0) {
      fprintf(stderr, "%s: unknown address\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]));  /* converts to network format */
    /* Configures socket to only send multicast packets to the local network */
    if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, (char *) &ttl,
        sizeof(ttl)) < 0) {
      perror("Failed setsockopt IP_MULTICAST_TTL");
    }
    /* Sends packet */
    if (sendto(sock, DATA, strlen(DATA)+1 /* so send '\0'*/, 0,
```

```
        (struct sockaddr *)&name, sizeof(name)) < 0)
      perror("Error sending datagram");
    close(sock);
}
```

Finally, create a file named "Makefile" to automate the creation of executables. The main goal is described in the first line (`all: cli serv`) - creating two executables. The following two lines indicate which files are used to create each executable (`cli: cli.c` − `cli` is created from `cli.c`) and the command line to create the executable (`gcc -g -o cli cli.c`) preceded by a tab.

The file "`Makefile`" has the following content:

```
all: cliv4 servv4

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

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

3.2. Client and Server for IPv6 UDP Multicast in text mode

The application implementation for IPv6 is very similar to that for IPv4, except for the use of functions specific to IPv6. Using the addresses **"::ffff:a.b.c.d"** this application can communicate with the application developed in section 3.1, i.e., it works in dual-stack mode. In this example, the server starts at port 20000 and joins the IP Multicast "ff18:10:33::1" if both parameters are omitted and it is blocked waiting to receive a message. The message can be received either via the IP Multicast or Unicast addresses of the local machine.

The code for "`servv6.c`" is:

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

/* Returns one string with the IPv6 address */
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";   // default address
  struct ipv6_mreq   imr;
  char               loop = 1;
  /* reception */
  char               buf[1024];
  struct sockaddr_in6 fromaddr;
  int                fromlen= sizeof(fromaddr);

  /* Creates the socket. */
  sock = socket(AF_INET6, SOCK_DGRAM, 0);
  if (sock < 0) {
    perror("Error opening ipv6 datagram socket");
    exit(1);
  }
  if (argc >= 2) {
```

```
    /* The port number can be introduced has the first parameter */
    porto= (short int)atoi(argv[1]);
  }
  if (argc == 3) {
    /* The second parameter is the IPv6 multicast address */
    addr_mult= argv[2];
  }
  /* Makes the port number sharable for this IP - allows having more than one servers with the
     same port in the same machine */
  if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *) &reuse,
      sizeof(reuse)) < 0) {
    perror("Failed setsockopt SO_REUSEADDR");
    exit(-1);
  }
  /* Associates the socket to a port */
  name.sin6_family = AF_INET6;
  name.sin6_flowinfo= 0;
  name.sin6_port = htons(porto);
  name.sin6_addr = in6addr_any;  /* local IPv6 (::) */
  if (bind(sock, (struct sockaddr *)&name, sizeof(name)) < 0) {
    perror("Binding of datagram socket failed");
    exit(1);
  }
  /* Multicast configuration */
  if (!inet_pton(AF_INET6, addr_mult, &imr.ipv6mr_multiaddr)) {
    perror("Failed conversion of multicast address");
    exit(1);
  }
  imr.ipv6mr_interface = 0;  /* Interface 0 */
  /* Join the group */
  if (setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, (char *) &imr,
      sizeof(imr)) == -1) {
    perror("Failed association to IP multicast group");
    abort();
  }
  /* Configure socket to receive echo of multicast packets sent */
  setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop, sizeof(loop));

  /* Finds out the socket's port number defined by the system */
  length = sizeof(name);
  if (getsockname(sock, (struct sockaddr *)&name, &length)) {
    perror("Error getting the port number");
    exit(1);
  }
  printf("The socket at address IP Multicast %s has port %d\n",
    addr_ipv6(&imr.ipv6mr_multiaddr), htons(name.sin6_port));
  /* Read a message from the socket */
  if (recvfrom(sock, buf, 1024, 0/*By default blocks*/, (struct sockaddr *)&fromaddr, &fromlen)
      < 0)
    perror("Error receiving datagram packet");
  printf("Received from %s#%d -->%s\n", addr_ipv6(&fromaddr.sin6_addr),
      ntohs(fromaddr.sin6_port), buf);
  printf("The server is going down\n");
  close(sock);
}
```

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

O código do "cliv6.c" é:

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

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

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

char *addr_ipv6(struct in6_addr *addr) {
```

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

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

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

The creation of the file Makefile is left for exercise.

As an **additional exercise**, you may limit the time waiting for packets to 10 seconds.

3.3. Client and Server for IPv6 TCP in text mode

Application programming with TCP sockets is a bit more complicated because it is connection-oriented. The difference between this and the client's previous example is in connection establishment and termination. All other initializations are identical. This example illustrates how you can send a message with two components – sending one component at a time.

The code for the client "clitcpv6.c" is:

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

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

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

```
}

/*
 * This program creates a socket and initiates a connection
 * with the socket given in the command line. One message
 * is sent over the connection and then the socket is
 * closed, ending the connection.
 */
main (int argc, char *argv[])
{
     int                     sock, msg_len;
     struct sockaddr_in6     server;
     struct hostent *hp, *gethostbyname();

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

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

The server code is much more complex because multiple parallel connections will coexist on the server. In this example, the server binds to port 20000 and prepares to receive calls (with the function `listen`). From there, the server is blocked in a cycle waiting to receive calls (with the function `accept`), writing the content of the first message received from each connection and then closing it. Using threads, subprocesses, or the `select` function, it would have been able to receive new links and data concurrently from multiple connections.

The code of the server, "`servtcpv6.c`", is:

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

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

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

 /* Create socket on which to read. */
 sock = socket(AF_INET6, SOCK_STREAM, 0);
 if (sock < 0) {
     perror("opening stream socket");
```

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

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

3.4. Program with threads in text mode

This example illustrates the creation and termination of POSIX threads (*pthreads*) to run concurrent activities.

The program code of "demopthread.c" is:

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

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

  // ptr was initialized in pthread_create argument
  int *state= (int *)ptr;
  /* function code */
  fprintf (stderr, "thread started\n");
```

34

```
  *state= 2;              // Changing the state
  fprintf (stderr, "thread is sleeping for 5 seconds\n");
  sleep (5);              // Sleep 2 seconds
  fprintf (stderr, "thread stopped\n");

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

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

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

3.5. Program with subprocesses in text mode

This example illustrates the creation and termination of a sub-process and using a pipe to send a message with two elements from the child process to the parent process. Function `reaper` analyses why the child process ends.

The program code of "`demofork.c`" is:

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

void reaper(int sig)          // callback to handle SIGCHLD
{
  sigset_t set, oldset;
  pid_t pid;
  union wait status;

  // blocks other SIGCHLD signals
  sigemptyset(&set);
  sigaddset(&set, SIGCHLD);
  sigprocmask(SIG_BLOCK, &set, &oldset);
  //
  fprintf(stderr, "reaper\n");
  while ((pid= wait3(&status, WNOHANG, 0)) > 0) {    // While there are zombie childs
    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;
```

```
  }
  // Reinstall signal handling
  signal(SIGCHLD, reaper);
  //Unblock SIGCHLD signal
  sigemptyset(&set);
  sigaddset(&set, SIGCHLD);
  sigprocmask(SIG_UNBLOCK, &set, &oldset);
  //
  fprintf(stderr, "reaper ended\n");
}

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

  signal(SIGCHLD, reaper);                        // prepares callback for SIGCHLD signal
  if (pipe(p)) { perror(pipe); exit(-1); }        // Create a pair of local pipes
  n= fork();                                      // Creates one subprocess
  if (n == -1) {
    perror("fork failed");
    exit(-1);
  }
  if (n == 0) {
/*************************************************************************/
// Code of the child process
    char *msg= "Hello";
    msg_len= strlen(msg)+1;
    fprintf(stderr, "child (pid = %d)\n", (int)getpid());
    close(p[0]);             // p[0] is used by the parent
    sleep(2);                // sleeps 2 seconds
    // Sends a message to the parent
    write (p[1], &msg_len, sizeof(msg_len)); // send the length
    write (p[1], msg, msg_len);              // send the msg
    close(p[1]);
    fprintf(stderr, "the child process died\n");
    exit(1);                 // Ends the child process
/*************************************************************************/
  }
  // Code of the parent process
  fprintf(stderr, "parent: child process started with pid %d\n", n);
  close(p[1]);               // p[1] is used by the child
  do {
      m = read (p[0], &msg_len, sizeof(msg_len));
  } while ((m == -1) && (errno == EINTR));   // Repeats if it was interrupted by a signal
  buf= (char *)malloc(msg_len);
  do {
      m = read (p[0], buf, msg_len); // Waits for a message from the child
  } while ((m == -1) && (errno == EINTR));   // Repeats if it was interrupted by a signal
  fprintf (stderr, "Child process %d sent %d/%d bytes:'%s'\n", n, m, msg_len, buf);
}
```

3.6. IPv4 UDP client and server with a graphical interface

This distributed application consists of two executables created from two files Glade-3. The server creates a UDP socket and associates it with port 20000, waiting for messages from a client. The content of incoming messages is written in a `GtkTextView` box, which can be cleaned up using a button. The client allows you to send the contents of a text box (`GtkEntry`) to a remote socket. Apart from the immediate sending, it supports deferred sending after a configurable number of milliseconds. The client registers all submissions and may resend the message to a previous IP address.

3.6.1. Server

The server must be programmed in a different directory from the client. The first part of programming the server is drawing the GUI using Glade-3. You must create the project "demoserv3.glade".

The server is developed from a GtkWindow window (window1), subdivided into three window lines using a GtkBox, and the first line in two columns with a GtkBox (hbox1). Then, you enter a GtkLabel with "port 20000", a button (button1), and a GtkTextView (text memo) with a scrollbar, which should be renamed according to the component tree shown on the right.

The development of the code begins with the creation of the file "gui.h", which defines the structure pointing to graphical components and declares the global functions used.
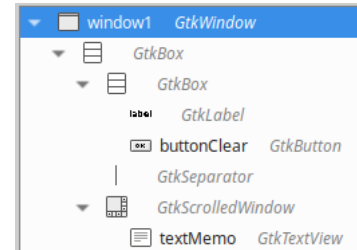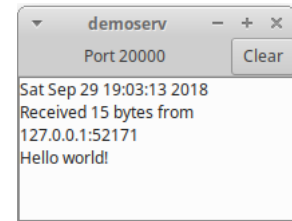
```c
#include <gtk/gtk.h>

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

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

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

The initialization function creates the window. On error, a window opens with the error using the function error_message defined in module gui_g3.c, provided with the assignment.

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

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

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

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

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

```
        return TRUE;
}
```

In the next step, we program the functions processing graphical events. A callback will be associated with the signal "**delete**" of the main window "**window1**" to stop the program when the window is closed:

```
gboolean on_window1_delete_event        (GtkWidget      *widget,
                                         GdkEvent       *event,
                                         gpointer        user_data)
{
  gtk_main_quit();      // Stop the Gtk+ main loop
  return FALSE;         // Closes the program
}
```

A function will be associated with the event "**clicked**" of the button "**button1**" in order to clear the contents of the box "textMemo." It must be added the following code:

```
void on_buttonClear_clicked             (GtkButton      *button,
                                         gpointer        user_data)
{     /* Clears the textMemo */
      GtkTextBuffer *textbuf;
      GtkTextIter tbegin, tend;

/* Gets a reference to the data model */
      textbuf = GTK_TEXT_BUFFER (gtk_text_view_get_buffer(main_window->text_view));
/* defines 2 limits and clears the window */
      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);
}
```

To facilitate the writing of messages, we created the auxiliary function Log:

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

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

To facilitate the development of applications that communicate using sockets, a module is provided in sock.c and sock.h with a set of helper functions to handle sockets. The functions and variables available in this module are defined in the file "sock.h." You can check the files to see the details of those functions.

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

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

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

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

```
gboolean is_local_ip(const char *ip_str);

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

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

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

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

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

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

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

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

The creation of the socket and registration of the callback function associated is done in the main function, ensuring that from the moment you start the executable is ready to receive messages. The text of file main.c is then:

```
#define GLADE_FILE    "demoserv3.glade"

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

int main (int argc, char *argv[]) {
    /* allocate the memory needed by our ServWindowElements struct */
    main_window = g_slice_new (ServWindowElements);

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

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

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

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

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

    return 0;
}
```

To end the server programming, we need to program the callback function that receives data from the socket. The reading routine uses the macro **READ_BUF** to read field by field from the received message. The macro reads n bytes from a buffer to the address var and increments the read pointer pt. The event handler of the socket has the following code (in file callbacks.c):

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

  if (condition == G_IO_IN)
  {
    /* Read data from the socket */
    n = read_data_ipv4 (sock, buf, MAX_MESSAGE_LENGTH, &ip, &porto);
    if (n <= 0)
    {
     Log ("Read from socket failed\n");
     return TRUE;              // Keeps waiting for more data
    } else {  // n > 0
     time_t tbuf;
     short unsigned int m;
     char *pt;
     /* Writes date and sender of the packet */
     time (&tbuf);            // Gets current date
     sprintf (write_buf, "%sReceived %d bytes from %s:%hu\n",
              ctime (&tbuf), n, inet_ntoa (ip), porto);
     Log (write_buf);
     /* Read the message fields */
     pt = buf;
     READ_BUF (pt, &m, sizeof(m));   // Reads short (2 bytes) to m and moves pointer
                                     // pt points to the 3rd byte
     m = ntohs (m);   // Converts the number to host format
     if (m != n - 2)
     {
       sprintf (write_buf, "Invalid 'length' field (%d != %d)\n", m, n - 2);
       Log (write_buf);
       return TRUE;   // Keeps waiting for more data
     }
     /* Writes data to the memo box - assumes that it ends with '\0' */
     Log (pt);// pt points to the first byte of the string
     Log ("\n");
     return TRUE;     // Keeps waiting for more data
    }
  }
  else if ((condition == G_IO_NVAL) || (condition == G_IO_ERR))
  {
    Log ("Detected socket error\n");
    remove_socket_from_mainloop (sock, chan_id, chan);
    chan = NULL;
    close_socket (sock);
    sock = -1;
    /* Stops the application */
    gtk_main_quit ();
    return FALSE;     // Removes socket's callback from main cycle
  } else {
    assert (0);       // Must never reach this line - aborts application with a core dump
    return FALSE;     // Removes socket's callback from main cycle
  }
}
```

Note that the message consists of two octets with the length and the content of a string. It is only missing declaring the value of the maximum message length (MAX_MESSAGE_LENGTH)

and the signature of the functions and global variables created in the file definitions "callbacks.h":

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

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

// Maximum message length
#define MAX_MESSAGE_LENGTH    5000

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

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

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

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

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

To automate the building of the program a `Makefile` file was created, with instructions to compile all modules and the dependencies list:

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

all: $(APP_NAME)

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

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

To compile the application just run the command "`make`". The executable must be run in the same directory where the file *democli3.glade* is, or the file name must include the full path.

3.6.2. Client

The client is created in a new folder. The first part of the programming is again the design of the GUI with Glade-3, in the project "`democli3.glade`".

The GUI is made from a `GtkWindow` (`window1`), which is divided into four lines (`box1`). The first line is divided into five columns, respectively: a `GtkLabel` (initialized with `"IP"`), a `GtkEntry` (`entryIP` with the text `"127.0.0.1"`), a `GtkLabel` (initialized to `"Text"`), a `GtkEntry` (`entryText` with text `"Hello world!"`) and a `GtkButton` (`buttonSend` with the label `"Send"`).
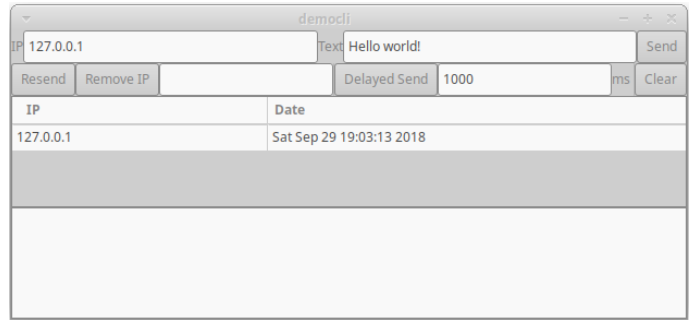
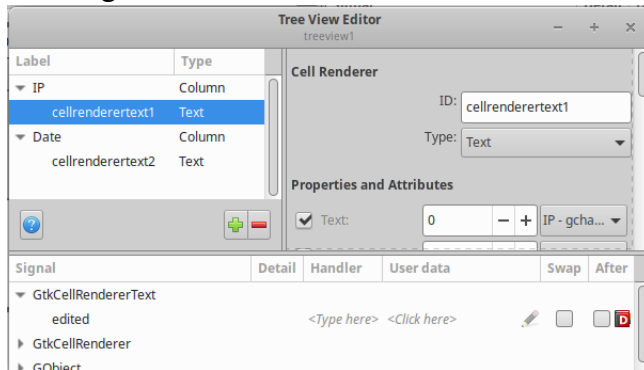The second row is divided into nine columns, respectively: a `GtkButton` (`buttonResend` with the label `"Resend"`), a `GtkButton` (`buttonRemove` with the label `"Remove IP"`), a `GtkEntry` (`entryRemIP`); a vertical separator (`separator1`), a `GtkButton` (`buttonDelayedSend` with the label `"Delayed Send"`), a `GtkEntry` (`entrymSec` with the text `"1000"`), a `GtkLabel` (`label3` with `"ms"`); `separator2`; and a `GtkButton` (`buttonClear` with the label `"Clear"`). The penultimate line contains a `GtkTreeView` (`TreeView1`) with two columns, and the last line contains a `GtkTextView` (`textview1`), both within a `GtkScrolledWindow`. For the illustrated aspect, you need to set the height and width of each window element using the editing window properties.

The table data is stored in `liststore1` and created together with the `GtkTextView`. It is necessary to edit `liststore1` to set two fields of type `gchararray` (i.e., string) respectively with the names `IP` and `Date`. Then you need to edit `treeview1` by pressing the right mouse button and selecting **Edit ...**, as shown below at the left. Comes up then the Tree View editor, which will be used to add and configure the two columns (with "**add column**"). At a later stage, each column must be associated with text viewers with "**add child text**." It is this step, represented at the left, that defines what data will be presented in each column.

In this example, it is defined that the IP "Text" column displays the "`IP-gchararray`" object in position 0 of `liststore1`. Therefore, any change made in the list is automatically updated in the graphics user interface.

The development of C code starts by creating the file "`gui.h`" with the definition of the structure pointing to the graphical components and the declaration of global functions used.

```
/* store the widgets which may need to be accessed in a typedef struct */
typedef struct
{
        GtkWidget               *window1;
        GtkEntry                *entryIP;
        GtkEntry                *entryText;
        GtkEntry                *entryRemIP;
        GtkEntry                *entrymSec;
        GtkTextView             *textview1;
        GtkTreeView             *treeview1;
        GtkListStore            *liststore1;
} CliWindowElements;
```

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

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

Again, the initialization function initializes the window and the structure and changes the font used in the text box `textView`:

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

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

    /* get the widgets which will be referenced in callbacks */
    window->window1 = GTK_WIDGET (gtk_builder_get_object (builder,
       "window1"));
    window->entryIP = GTK_ENTRY (gtk_builder_get_object (builder,
       "entryIP"));
    window->entryText = GTK_ENTRY (gtk_builder_get_object (builder,
       "entryText"));
    window->entryRemIP = GTK_ENTRY (gtk_builder_get_object (builder,
       "entryRemIP"));
    window->entrymSec = GTK_ENTRY (gtk_builder_get_object (builder,
       "entrymSec"));
    window->textview1 = GTK_TEXT_VIEW (gtk_builder_get_object (builder,
       "textview1"));
    window->treeview1 = GTK_TREE_VIEW (gtk_builder_get_object (builder,
       "treeview1"));
    window->liststore1 = GTK_LIST_STORE(gtk_builder_get_object (builder,
       "liststore1"));
    /* connect signals, passing our WindowElements struct as user data */
    gtk_builder_connect_signals (builder, window);

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

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

    return TRUE;
}
```

To facilitate posting in the dialog box, you will again use the `Log` function presented on page 38. Note that you need to make a slight modification because the name of the textbox on the client is "`textview1`" instead of "`textMemo`".

Given that messaging can be made from various functions, a function is created that sends the contents of `entryText` to an IP address defined by the parameter `ip`, or if the parameter is NULL, the content of `entryIP`. The function signature must be appended to the file "`callbacks.h`":

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

The macro `WRITE_BUF` is used to facilitate writing to a buffer. It is defined in file "`sock.h`" and copies `n` bytes from address `var` to `pt`, increasing `pt`. The function `send_message_ipv4` is programmed in the file "`callbacks.c`", sending the length of the

43

message before the message:

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

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

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

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

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

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

Next, the various graphical event handlers are programmed. The event "**clicked**" of button "**buttonSend**" must be associated with a function with the following code:

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

The event "**clicked**" of the button **buttonDelayedSend** must be associated with a function with the code shown below. This function sends a message with a delay equal to the number of milliseconds indicated in entrymSec using a timer. Note that the function callback_timer returns FALSE to stop the timer; if it returned TRUE would be called periodically.

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

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

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

Whenever a message is sent, a line with the IP and the transmission date is added to the table. The user can select a table row of tableview1 and send a message to the IP on that line. The function associated with the event "**clicked**" of the button buttonResend gets the selected row and sends a message:

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

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

The callback function associated with the event "**clicked**" of the button **buttonRemove**

45

was done with a helper function (`foreach_func`) called by `gtk_tree_model_foreach` for all elements of the list. This function returns a list of all lines with an address equal to the one contained in the variable `remove_ip`. This variable is pre-filled with the value of the `entryRemIP` text box. Then, all the rows in the delete list are deleted from `liststore1`.

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

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

// GUI callback function called when 'Remove IP' button is clicked
void on_buttonRemove_clicked (GtkButton * button, gpointer user_data)
{
    const gchar *textRIP;
    struct in_addr addr;

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

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

    remove_ip= NULL;
}
```

The function associated with the event "**clicked**" of button **buttonClear** clears the contents of `liststore1` and `textview1`:

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

    // Clear table with messages sent
```

```
    gtk_list_store_clear(main_window->liststore1);

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

To ensure that the application ends when the window is closed, the function on_window1_delete_event is associated to the event "**delete_event**" of window **window1**:

```
gboolean on_window1_delete_event(GtkWidget *widget, GdkEvent *event, gpointer user_data)
{
  gtk_main_quit();      // Stops the Gtk+ main cycle
  return FALSE;         // Closes the application
}
```

The "main.c" file should initialize and store the global pointer main_window to the CliWindowElements structure (similarly to that described on page 39) and start the socket. Finally, the file Makefile has a structure similar to the previous example.

3.6.3. Exercises

1) Modify the client and server code to work with IPv4 multicast addresses. **Suggestion:** Review the code presented in section 3.1.
2) Modify the client and server code to work with IPv6 unicast addresses. **Suggestion:** Review the code presented in section 3.2.
3) Modify the client and server code to work with IPv6 multicast addresses. **Suggestion:** Review the code presented in section 3.2.
4) Modify the client and server code to work with TCP sockets and IPv6 addresses, using threads to send and receive data through the sockets. **Suggestion:** Review the code presented in sections 3.3 and 3.4.

3.7. IPv4 TCP client and server with a graphical interface

This second distributed graphical application uses the same graphical interface and most of the code as the first application, presented in section 3.6. It only modifies the functions strictly necessary to support the connection-oriented service.

3.7.1. Server

In a TCP socket, messages are received in a socket created for the connection. In this way, two steps are required. In the first step, it receives the connection in the server socket (which accepts connections) and creates a data socket. In the second step, the message is received in the data socket. Thus, it becomes necessary to use a callback function for each phase.

Creating the *socket* and recording the socket handling function in the main cycle is done in the main function and is very similar to what is shown in section 3.6.1. You just change the socket initialization and the handler association, which is now responsible for receiving connections:

```
int main (int argc, char *argv[]) {
...
   /* Socket initialization */
   if ((sock = init_socket_ipv4 (SOCK_STREAM, 20000, FALSE)) == -1)
     return 1;
```

```
    listen(sock, 1);   // Set socket to receive connections
    if (!put_socket_in_mainloop (sock, main_window, &chan_id, &chan, G_IO_IN,
      callback_connection))
      return 2;
...
}
```

The `callback_connection` function handles the reception of connections in the socket `sock`, which has the code below (in the `callbacks.c` file). The event `G_IO_IN` is associated with the availability of a new connection. After creating the new data connection (`msgsock`), the data callback function `callback_data` is associated with G_IO_IN to handle the data reception. Note that multiple sockets can be created, leaving the system with an active callback for each data socket.

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

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

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

The data socket event handler function has the following code (in the `callbacks.c` file). The `g_io_channel_unix_get_fd` function is used to obtain the descriptor of the socket with data. The data is then read directly from the socket, releasing the channel when the connection ends.

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

    if (condition & G_IO_IN) {
      /* Read data from the socket */
      n = recv(s, buf, MAX_MESSAGE_LENGTH, 0);
      if (n < 0) {
          perror("read failed");
          Log ("Read from socket failed\n");
          free_gio_channel(source);
          return FALSE;                // Removes socket's callback from main cycle
      }
```

```
      else if (n == 0) {
          /* Reached end of connection */
          sprintf(write_buf, "Connection %d closed\n", s);
          Log(write_buf);
          free_gio_channel(source);
          return FALSE;                    // Removes socket's callback from main cycle
      } else {
          time_t tbuf;
          /* Writes date and sender of the packet */
          time (&tbuf);                    // Gets current date
          sprintf (write_buf, "%sReceived %d bytes from socket %d:\n", ctime (&tbuf), n, s);
          Log (write_buf);
          Log(buf);    // Write the message received
          Log("\n");
          return TRUE; // Keeps waiting for more data
      }

  } else if ((condition & G_IO_NVAL) || (condition & G_IO_ERR)) {
    Log ("Detected socket error\n");
    remove_socket_from_mainloop (sock, chan_id, chan);
    chan = NULL;
    close_socket (sock);
    sock = -1;
    /* Stops the application */
    gtk_main_quit ();
    return FALSE;    // Removes socket's callback from the main loop
  }
}
```

3.7.2. Client

As with the server, the client uses the graphical interface and all the code presented in section
3.6.2. It only modifies the send_message_ipv4 function that sends the message, which now
needs to open the connection, send the message, and close the connection. The new function code
is:

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

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

    struct sockaddr_in name;
    struct hostent *hp;

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

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

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

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

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

Note that this approach only works with a socket in blocking mode, where the write operation blocks the program until it is complete. When using a socket in non-blocking mode, it is necessary to control data sending through a callback associated with the G_IO_OUT event. The write is performed after receiving the event and until the socket stops accepting all the bytes written. Under these conditions, it is necessary to compare the number of bytes written to the socket with the number effectively sent and then resend the data that was not sent.

3.7.3. Exercises

1) Modify the client and server code to work with IPv6 unicast addresses. **Suggestion:** Review the code presented in section 3.3.
2) This example uses blocking sockets. Modify the client and server code to work with non-blocking sockets, preparing them to handle the EWOULDBLOCK error, which occurs when there is no space in the socket to write or when there is no data to read. On the client, a maximum time for data reception should be set at 10 seconds, and write operations should be performed only after the G_IO_OUT event. On the server, you must prepare the code to receive the data in various G_IO_IN evocations.
3) Modify the client and server code to work with threads and send and receive data concurrently through multiple sockets. **Suggestion:** Review the code presented in sections 3.3 and 3.4.
4) Modify the client and server code to work with subprocesses and send and receive data concurrently through multiple sockets. **Suggestion:** Review the code presented in sections 3.3 and 3.5.