First Experiences with BlueZ

Sukey Nakasima-López[1], Francisco Reyna-Beltrán[2], Arnoldo Díaz-Ramírez[2], and Carlos T. Calafate[3]

[1] Graduate School, Cetys Universidad
Calzada Cetys s/n, Mexicali, Mexico
sukey.nakasima.lopez@gmail.com
[2] Department of Compuer Systems, Instituto Tecnológico de Mexicali
Ave. Tecnológico s/n, Mexicali, Mexico
freyna, adiaz@itmexicali.edu.mx
[3] Department of Computer Engineering, Polytechnic University of Valencia (UPV)
Camino de Vera s/n, Valencia, Spain
calafate@disca.upv.es

Abstract. The low cost and low battery consumption of *Bluetooth* devices, allied with a plethora of novel functionalities, has promoted the widespread adoption of this technology. In this paper we enter into the *Bluetooth* Technology, that like an emergent technology, it is conceived to be the best option for the wireless communication, combined to the ample range of possibilities of development of applications, positioning itself like the technology of the future for the movable devices. Our contribution consists of impelling the development of mobile applications under this technology and shares our experiences. We have chosen *BlueZ*, a protocol stack with License *GPL* for *Linux*, and using *C*, *C++* and *Qt* as programming languages. We hope to promote its use and show its tools for application development at basic level for new *BlueZ/C++* users.

Keywords: Bluetooth, BlueZ.

1 Introduction

The necessity of communication nowadays is a prevailed subject, for businesses or for personal reasons, it is always necessary to be in constant communication. In an enviroment where the technological devices become indispensables in our lifes, because of little effort that these require, the facility of its uses and in some cases its low costs, it is difficult to imagine us without a cellphone, palmtop, laptop or any other device that allows us to be in contact with the exterior world. At this life-rhythm, it is elementary to be able to free us of cables and use wireless connections of short-range in order to facilitate the demand of connectivity between the devices, so *Bluetooth* is the simplest option.

Bluetooth is a wireless standar available in the whole world, it connects mobile telephones to each other, laptops, MP3 players and a lot of other devices. This

Advances in Computer Science and Artificial Intelligence.
Research in Computing Science 39, 2008, pp. 97-114

Received 14/06/08
Accepted 01/08/08
Final version 11/09/08

technology provides great efficiency and saving of costs for home and businesses users; allowing the replacement of cable, the facility to share files, wireless synchronization and connectivity to internet, also, thanks to its great acceptance, a *Bluetooth* device can be connected with almost any other compatible device in its proximities eliminating the borders anywhere of the world.

In other way, for the best advantage of all these qualities already mentioned about *Bluetooth*, it is elemental have the necessaries tools that help to the efficient development of applications for this technology. In this sense, *BlueZ* is an alternative to consider, starting from the fact that it is free-software and included in the Linux core since the version 2.4. Unfortunately almost does not exist documentation about *BlueZ* or about the APIs to development applications with *BlueZ*.

In this article we describe briefly the features of the *Bluetooth* technology and we will show the *BlueZ* protocols stack by Linux, thus examples of its uses, so that any user who begins to developing under this standard counts by a minimal reference and understands better its operation.

2 Bluetooth technology

The *IEEE 802.15.1* standard [8], also known as *Bluetooth*, is an open standard for the wireless connectivity that allows the data and voice transference between the communication devices and PC's giving facility to the users for create *Wireless Personal Area Networks (WPANs)* and *Ad Hoc networks*, impelling a greater integration of the *Bluetooth* technology to MANET networks (*Mobile Ad Hoc Network*).

2.1 Bluetooth features

Bluetooth operates in a free license band, enlarging the possibilities of its use, this industrial, scientific and medical band (ISM) is between 2.4 and 2.485 Ghz, using an extended spectrum, frequency hops, full-duplex signal in a nominal rate of 1600 hops/sec. Actually there are three available "classes", the *Bluetooth* devices have a rank operation from 3 - 300 feet (1-100m), depending the class of the device and adapted for the needs of the user.

In order to reduce to the minimum any interference, the *Bluetooth* technology makes use of a capacity denominated adapted frequency hops (AFH); which was designed to detect other devices in the spectrum and to avoid the frequencies that are in use, at this way the signal hops between 79 frequencies in 1 Mhz intervals, generating a high degree of immunity to interferences, respecting to the voice and data transferences, *Bluetooth* supports up to 3 synchronous channels of voice of 64kbls each one and asynchronous data up to 723,2 kb/s asymmetric or 433.9 kb/s symetric [3].

2.2 Bluetooth profiles

So that device can use the *Bluetooth* wireless technology, it must know how to interpret the *Bluetooth profiles*, as shown in Figure 1 that describes the different possible applications. These profiles are guides that specify the adjustment of stack parameters as well as the required features and procedures so that the devices equipped with *Bluetooth* technology communicate to each other. Thanks to the exclusive concept of "profiles", it is not necessary to install controllers in the *Bluetooth* devices. All the profiles supported by *Bluetooth* are defined in its web site [9].

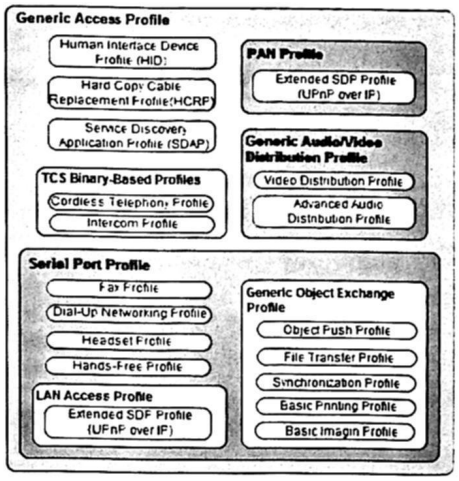

Fig. 1. *Bluetooth profile structure.*

2.3 Network topology

Piconets y scatternets. The *Bluetooth* wireless links are formed in the context of a Piconet. Piconet is a group of two to eight devices that occupy the same physical channel (unique for each piconet), consisting of a single master device and one or more slaves, where slave devices are synchronized to the same clock and a specific pattern of frequency hops provided by the master device. Besides, a device can belong at the same time as more of a Piconet, creating what is called a Scatternet, as seen in Figure 2.

Operational procedures. *Bluetooth* uses a search procedure (*Inquiry*) to discover or to be discovered by other near devices, as well as to discover the services that these nodes offer. Followed to this, the paging procedure is used to contact between both devices.

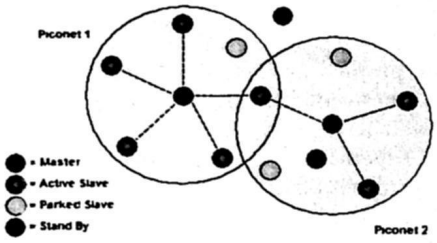

Fig. 2. *Bluetooth scatternet diagram.*

2.4 Bluetooth specification

The core specification. The core specification define all the layers of the *Bluetooth protocols stack* (Figure 3), which is structured in four layers with associated protocols defined by the specifications or *Bluetooth* profiles. The three inferior layers radio, baseband and Link Manager are grouped in a subsystem denominated *Bluetooth Controller (Module)*, while the *L2CAP* layer, services layer and the superior layers are known as *Host Bluetooth*. This grouping of the core layers requires a physical communication interface between the *Bluetooth* controller and the *Host Bluetooth*, this interface is known as *HCI (Host Controller Interface)*. The *Bluetooth* specification makes possible the compatibility between different *Bluetooth* systems through of the definition of protocol messages that interchange it between the equivalent layers. Also it determines a common interface between the controllers and *Bluetooth* Hosts to make compatible the different subsystems.

Protocols stack. The *Bluetooth protocol stack* is divided in two zones, each one is implemented in different processor:

The *Bluetooth Module (hardware)* is the responsible of the tasks related to the information transferences through radio frequency interface. The *Host Bluetooth (software)* is the responsible of the part related to the layers superiors of connection and application.

On the layer of specific *Bluetooth* protocols, each manufacturer can implement his propietary protocols layer of application. by this way, the open specification of *Bluetooth* expands the number of applications that can be benefited from their capacities. Even though, the *Bluetooth* specification demands that, in spite of the existence of different propietary application protocol stacks, the interoperability must exist between devices that implement different stacks.

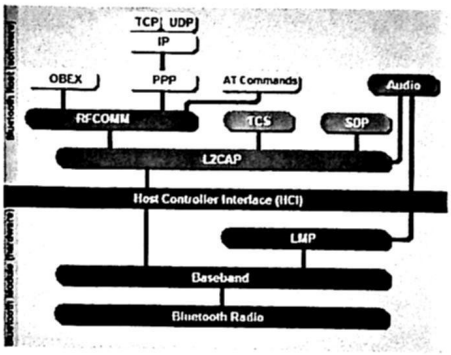

Fig. 3. *Bluetooth Protocols Stack.*

3 BlueZ

In the following paragraphs the main characteristics of *BlueZ* and their tools will be commented. There is special emphasis in the pand daemon beacuse it is the base of the development.

3.1 BlueZ definition and features

BlueZ [7] is the *Official Linux Bluetooth Protocol Stack*; it was developed initially by *Qualcomm* and at present is released under the *General Public License (GPL) [4]* that means that can be copied, studied, modified and redistributed freely.

BlueZ is part of the Linux official kernel since the version 2.4, therefore, included in any modern distribution of Linux; it will be not necessary to install something. Some of the main characteristics of *BlueZ* are:

- Flexible, efficient and modular architecture.
- Support for multiple *BlueZ* devices.
- Multitask data processing.
- Hardware abstraction.
- Standard socket interface to all layers.
- Multi-platform: x86 (single and multi-processor), SUN, SPARC, ARM, PowerPC, Motorolla, DragonBall.
- Operation in all the Linux distributions: RedHat, Debian, Suse, etc.
- Great quantity of supported devices (PCMCIA, UART, USB).
- Supports *L2CAP, SDP, RFCOMM* and *SCO.*
- Availability of a *Bluetooth* emulator and devices of configuration and test.
- Support for the following profiles of use: GAP, DUN, LAN, SPP, PAN, Headset, OBEX (FTP), OBEX (OPP).
- Mailing lists of participating, with developers anywhere in the world contributing to the support and programming with *BlueZ.*

The main disadvantage of *BlueZ* is that there is not enough documentation. This can be solved directly studying the source code, but this takes a lot of time and could be complicated for the new users.

3.2 BlueZ packages

BlueZ is distributed in a set of packages, although the core depends on the distribution of kernel Linux that we are using. For the previous versions to version 2.4 that do not include the *Bluetooth* functionality exists patches [6]. Besides the support of the core, the packages that can be used based on the final needs of the users are:

bluez-libs: Necessary libraries for the development of applications and the rest of *BlueZ* packages and applications that link dynamically to the libraries.
bluez-utils: Control applications for the *Bluetooth* devices. Necessary to make inquiry or general communications.
bluez-sdp: It contains the libraries, tools and the *SDP* server (sdpd) that conform all the *SDP* functionality .
bluez-pan: Programs, daemons and scripts necessaries for the profiles DUN, LAN and BNEP-PAN.
bluez-hcidump: Useful orders to debug and to study the general operation of the devices *Bluetooth* using *HCI*.
bluez-hciemu: It contains the emulator. It allows the programmers to test their code without a real *Bluetooth* device.
bluez-bluefw: It contains the firmware of several kind of *Bluetooth* devices.

All these packages can be downloaded from the official *BlueZ* site [7] in the section of downloads.

3.3 BlueZ tools

As already commented, the documentation of this protocol stack is non-existent and therefore the knowledge about this API has been realized from the own source code. Fortunately, the *BlueZ* core comes accompanied by a set of tools that allows to execute the *Bluetooth* functions implemented in the protocol stack from a shell or console orders. The first step to determine which are the functions that interest to us is studying of these tools:

hciconfig: Configure local *Bluetooth* devices.
hcid: *HCI* interface daemon.
hcitool: Link manager with other *Bluetooth* devices, detection of remote devices and name resolutions among others fuctions.
hcidump: Local sniffer for the *HCI* traffic, either incoming or outgoing, by the *Bluetooth* device installed in the system.
l2ping: Send request "echo request (ping)" in *L2CAP* level.

sdptool: *SDP* manager, dicovering of *Bluetooth* services in remote devices.
sdpd: Daemon of the service discover protocol *SDP*. It manages to provide access to the local *Bluetooth* services.
rfcomm: Manager of connections rfcomm.
pand: Manager of PAN (*Personal Area Network*) connections.

For greater information about command-functions that provide us these tools, we can review the manuals that linux provides to us in the console. For example, look the fuctions that sdptool privide to us:

```
#: man sdptool
```

The command-functions provided by these tools are relatively very simple to use, the manual provides information of each command, as well as of its syntax.

4 Sockets

Sockets [2] are a fundamental tool in the communication between devices, throughout the next examples we will use sockets in repeated occasions to make connections. For such reason is good idea to know a little bit of sockets implementation before getting to program with them.

The function `socket()` returns a socket descriptor, which we will be able to use soon for calls to the system. If it returns -1 , an error has taken place.

```
int socket(int Dominio,int Tipo,int Protocolo);
```

- Domain: It defines the property to a group of socket that we want to use, that is, you can use AF_INET (for protocols ARPA of Internet), AF_UNIX (protocols that allow internal communication of the system) and AF_BLUETOOTH (protocol for the communication between devices that support this technology).
- Type: It refers about to the class of socket that we want to use, is this of datagrams UDP or data stream TCP. We will use SOCK_SEQPACKET (is used to indicate a socket with reliable datagram-oriented semantics where packets are delivered in the order sent).
- Protocol: It indicates the protocol that will allow us the information transference (BTPROTO_L2CAP).

Once obtained the socket descriptor, will be necessary to associate it with a port, for this reason, we will make use of the function `bind()` [4].

```
int bind(int fd, struct sockaddr *my_addr,int addrlen);
```

Already established the relation between socket and port we can to make a connection through the function `connect()`. This function is used to connect to a port defined in a IP address.

```
int connect(int fd, struct sockaddr *serv_addr,
int addrlen);
```

Now to be able to obtain that our device remains awaiting for some other device can be connected to, it will have to use the function `listen()`, which has main function remain awaiting the incoming connections.

[4] For more information about the functions parameters, see the reference [2].

```
int listen(int fd,int backlog);
```

Already having a function that is awaiting for a connection, when finally some device is connected, is necessary to accept this connection, for it we will use the function `accept()`.

```
int accept(int fd, void *addr, int *addrlen);
```

We already have all the previous elements, you will be able to use of the functions `send()`, `recv()`, `write()` and `read()` for the exchange of information through sockets descriptors .

For the development of the next applications, we have made use of the protocol *L2CAP*, that allows the interchange of packages with or without connection-oriented, but as we mentioned previously are diverse protocols of which you can make use according to your needs.

5 Bluetooth applications examples

The development is located for Linux platforms, doing use of the *BlueZ* protocol stack *BlueZ* that comes included in the linux kernel. For which, it is elementary that the core of the application to be developed entirely in language *C++* programming, since it is the used one in *BlueZ*, whereas the graphical interface is recommended to be realized in *C++* language using the library *Qt* de Trolltech [1].

As previously mentioned (Section 3), the *BlueZ* protocol supports different transport protocols as *RFCOMM, L2CAP, SCO*, etc., which use a programming structure (interface) based on *sockets* (Section 4) to be able to communicate, Unlike those protocols, *HCI* made it easier to use thanks to functions and specific-use commands that it provides us. In the subsequent examples the protocols *L2CAP, SDP* and *HCI* are used, showing the difference between its uses and capabilities.

This chapter starts from a simple scaning application to a client/server using a *SDP* service. Logically programming in *C++* for *BlueZ* has a much broader scope of which we will present, but, by the propose of our work some examples at basic level for new *Bluetooth-BlueZ* and *C++* users are shown.

5.1 Searching nearby devices (scan)

The scan is a primary function at the time of schedule with *Bluetooth* devices, because it provides us information of the nearby devices with which we can interact. this tool is defined by default as a *BlueZ* tool (hcitool scan), also is recommended to understand its operation on the code, since it is highly required in most applications that we develop. A way to get this code is exploring the *BlueZ* libraries [5], where not only the scan function is, we can find all the functions defined by the API (Section 3.3); is necessary to identify the .c file with the

[5] It is recommended not to modify the original libs, you can download them from www.bluez.org/downloads and make with them your tests.

functions, isolate the code and make our tests to help us to understand its functioning.

The following example [5] will search for nearby *Bluetooth* devices, providing us its name and *Bluetooth* address. This program will use *HCI*, in subsequent examples were used *L2CAP* and *SDP*.

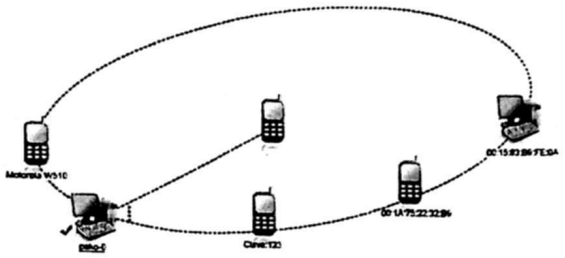

Fig. 4. *Scanning nearby devices.*

Note: *The explanation of the examples code will be given at the end of each.*

scan.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>
int main(int argc, char **argv)
{
 inquiry_info *ii = NULL;
 int max_rsp, num_rsp;
 int dev_id, sock, len, flags;
 int i;
 char addr[19] = { 0 };
 char name[248] = { 0 };

 dev_id = hci_get_route(NULL);
 sock = hci_open_dev( dev_id );

 if (dev_id < 0 || sock < 0) {
 perror("opening socket");
 exit(1);
 }

 len = 8;
 max_rsp = 255;
 flags = IREQ_CACHE_FLUSH;
 ii = (inquiry_info*)malloc(max_rsp * sizeof(
 inquiry_info));
 num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL,
 &ii, flags);

 if( num_rsp < 0 ) perror("hci_inquiry");
```

```
for (i = 0; i < num_rsp; i++) {
ba2str(&(ii+i)->bdaddr, addr);
memset(name, 0, sizeof(name));

if (hci_read_remote_name(sock, &(ii+i)->bdaddr,
sizeof(name), name, 0) < 0)
strcpy(name, "[unknown]");
printf("%s %s\n", addr, name);
}

free( ii );
close( sock );
return 0;
}
```

To compile the program is necessary to link the *Bluetooth* library for use its functions.

```
# gcc -o scan scan.c -lbluetooth
```

Running the program...

```
# ./scan
```

There are differents predefined structures for schedule *C++* with *BlueZ*, these structures will be required for some functions and we must know its purpose. One of the most often used is bdaddr_t; which is referred to store and manipulate the *Bluetooth* devices addresses.

```
typedef struct {
 uint8_t b[6];
} __attribute__((packed)) bdaddr_t;
```

These addresses can be converted between strings and bdaddr_t structures or the opposite through the following functions:

```
int str2ba( const char *str, bdaddr_t *ba );
int ba2str( const bdaddr_t *ba, char *str );
```

When we schedule, is possible to have multiple *Bluetooth* devices in our computer, therefore, it is required to specify which *Bluetooth* adapter we are going to use for allocating system resources. These adapters are identified by a number starting from 0.

If you already know the *Bluetooth* local adapter (*Bluetooth address*), the following function returns the resource number of the *Bluetooth* adapter address passed in as a parameter.

```
int dev_id = hci_devid( "00:32:56:27:6B:9A" );
```

But, if only we have a *Bluetooth* adapter or no matter which we use, the following function (Null) retuns the resource number of the first *Bluetooth* adapter available.

```
int dev_id = hci_get_route( NULL);
```

Once the *Bluetooth* adapter is defined, is required to open a socket using *hci_ open_ dev*, this function opens a socket connection to the microcontroller (for controlling it) on the specified local *Bluetooth* adapter. "Attention",the socket is not a connection to a remote *Bluetooth* device.

```
socket = hci_open_dev( int dev_id );
```

If there are errors when opening the socket, the function returns -1 and sets errno [6], if there are no problems, returns a handle to the socket.

When the socket is already open, the scan (*inquiry*) starts using the function:

```
int hci_inquiry(int dev_id, int len, int max_rsp,
```

[6] Most of the functions return -1 in case of error.

```
const uint8_t *lap, inquiry_info **ii, long flags);
```

This function does not even use the socket, we use the resource number of the *Bluetooth* adapter (int dev_id), the duration of the inquiry (len * 1.28sec.), the maximum number of responses (max_rsp), a structure for storing the info of inquiry (inquiry_info ** ii) and flags for indicate whether or not to use previously discovered device information or to start a fresh (IREQ_CACHE_FLUSH: cache flushed, 0: results of previous inquiries may be returned). If there is not an error, the devs parameter are stored in an predefined array of inquiry_info structures.

It is often easier to identify devices by its friendly name (nickname) than by its direction, hence, the function hci_read_remote_name provides us the remote device name in a char (name). Is necessary to indicate the socket and the *Bluetooth* device address.

```
int hci_read_remote_name (int socket, const bdaddr_t
 * ba, int len, char * name, int timeout);
```

finally free the memory used by * ii and close the socket.

```
free( ii );
close( socket );
return 0;
```

5.2 Basic client-server

The example above applies only the scanning *HCI* function, now we will see the incorporation of functions *L2CAP* to demonstrate how to establish an *L2CAP* channel and transmit a string of data like a server-client application.

server.c

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>
int main(int argc, char **argv)
{
 struct sockaddr_l2 loc_addr={0}, rem_addr={0};
 char buf[1024] = {0};
 int s, client, bytes_read;
 socklen_t opt = sizeof(rem_addr);

 // allocate socket
 s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_
 L2CAP);

 // setting parameters
 loc_addr.l2_family = AF_BLUETOOTH;
 loc_addr.l2_bdaddr = *BDADDR_ANY;
 loc_addr.l2_psm = htobs(0x1001);

 // bind socket to port 0x1001 of the first
 //available bluetooth adapter
 bind(s, (struct sockaddr *)&loc_addr, sizeof
 (loc_addr));

 // put socket into listening mode
 listen(s, 1);

 // accept one connection
```

```
client = accept(s, (struct sockaddr *)&rem_addr,
&opt);

ba2str( &rem_addr.l2_bdaddr, buf );
fprintf(stderr, "accepted connection from %s\n"
, buf);
memset(buf, 0, sizeof(buf));

// read data from the client
bytes_read = read(client, buf, sizeof(buf));

if( bytes_read > 0 ) {
printf("received [%s]\n", buf);
write(client, "hello client!", 15);
}
close(client);// close connection
close(s);
}
```

client.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>
int main(int argc, char **argv)
{
 struct sockaddr_l2 addr = { 0 };
 int s, status,bytes_read;
 char server[18], buf[1024] = { 0 };

 if(argc < 2) {
 fprintf(stderr, "usage: %s <bt_addr>\n",
 argv[0]);
 return 1;
 }

 //asignate the server address to server
 strncpy(server, argv[1], 18);

 // allocate a socket
 s = socket(AF_BLUETOOTH, SOCK_SEQPACKET,
 BTPROTO_L2CAP);

 // set the parameters for connection
 addr.l2_family = AF_BLUETOOTH;
 addr.l2_psm = htobs(0x1001);
 str2ba( server, &addr.l2_bdaddr );

 // put socket into listening mode
 listen(s, 1);

 // connect to server
 status = connect(s, (struct sockaddr *)&addr,
 sizeof(addr));

 // if connect succefully
 if( 0 == status ) {

 // send a message to server
 status = write(s, "hello server!", 15);

 // read data from the client
 bytes_read = read(s, buf, sizeof(buf));
 if( bytes_read > 0 ) {
```

```
printf("received [%s]\n", buf);
}
}

if( status < 0 ) perror("Error:");
close(s);
return 0;
}
```

For connecting we have to open a *L2CAP* socket;

```
s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
```

The first parameter should still be AF_BLUETOOTH, but the next two parameters should be SOCK_SEQPACKET and BTPROTO_L2CAP, respectively.

L2CAP sockets use the struct sockaddr l2 addressing structure:

```
struct sockaddr_l2 {
 sa_family_t l2_family;
 unsigned short l2_psm;
 bdaddr_t l2_bdaddr;
};
```

The first field, l2 family, should always be AF_BLUETOOTH, l2_bdaddr denotes the address of either a server to connect to, a local adapter and port number to listen on, or the information of a newly connected client, depending on context. and the l2_psm field specifies the *L2CAP* port number to use.

Running the client program we must indicate the server address to connect, for example:

```
#: ./client 00:0C:78:31:FC:8C
```

If we don't know the server address, we could find it using:

```
#: sdptool scan
```

5.3 Registering and searching a service

When a *Bluetooth* device offers a service (application) acts as a server, making it necessary to specify the type of service being offered and register it; thus the remote devices (clients) searching for this service could identify the service and request it to a correct device (server).

The Service Discovery Protocol (SDP) defines the way by which a client can discover the services availables on *Bluetooth* devices, as well as their attributes (of services). In the following examples *SDP* tools are used for register and search *SDP* services.

The first program registers a service named called bluefriend (the name does not matter), while the second program will search for this service.

The SDP daemon. Every *Bluetooth* device typically runs an *SDP* server that answers queries from other *Bluetooth* devices. In *BlueZ*, the implementation of the *SDP* server is called sdpd, and is usually started by the system boot scripts. Sdpd handles all incoming *SDP* search requests.

Registering a service. Registering a service with sdpd involves describing the service to advertise, connected to sdpd, instructing sdpd on what to advertise, and then disconnecting. For make it easier the service has the *UUID* 0xABCD.

registering.c

```
#include <bluetooth/bluetooth.h>
#include <bluetooth/sdp.h>
#include <bluetooth/sdp_lib.h>
sdp_session_t *register_service()
{
 uint32_t service_uuid_int[] = { 0, 0, 0, 0xABCD };
 const char *service_name = "Bluefriend";
 const char *service_dsc = "An Matching Aplication";
 const char *service_prov = "BF Server";

 uuid_t root_uuid,l2cap_uuid, svc_uuid;
 sdp_list_t *l2cap_list = 0, *root_list = 0,
 *proto_list = 0, *access_proto_list = 0;

 sdp_record_t *record = sdp_record_alloc();

 // set the general service ID
 sdp_uuid128_create(&svc_uuid, &service_uuid_int);
 sdp_set_service_id( record, svc_uuid );

 // make the service record publicly browsable
 sdp_uuid16_create(&root_uuid, PUBLIC_BROWSE_GROUP);
 root_list = sdp_list_append(0, &root_uuid);
 sdp_set_browse_groups( record, root_list );

 // set l2cap information
 sdp_uuid16_create(&l2cap_uuid, L2CAP_UUID);
 l2cap_list = sdp_list_append( 0, &l2cap_uuid );
 proto_list = sdp_list_append( 0, l2cap_list );

 // attach protocol information to service record
 access_proto_list = sdp_list_append(0, proto_list);
 sdp_set_access_protos(record, access_proto_list);

 // set the name, provider, and description
 sdp_set_info_attr(record, service_name, service_
 prov, service_dsc);

 int err = 0; sdp_session_t *session = 0;

 // connect to the local SDP server (BDADDR_LOCAL),
 //register the service record, and disconnect
 session = sdp_connect( BDADDR_ANY, BDADDR_LOCAL,
 SDP_RETRY_IF_BUSY );

 err = sdp_record_register(session, record, 0);

 // cleanup
 sdp_list_free( l2cap_list, 0 );
 sdp_list_free( root_list, 0 );
 sdp_list_free( access_proto_list, 0 );
 return session;
}
int main()
{
 sdp_session_t* session = register_service();
 sleep(5);
 sdp_close( session );
 return 0;
}
```

A way to check whether the service there has been satisfactorily registered is through the command `sdptool browse`, this command shows all services registered in the local *Bluetooth* device.

Searching a service. Once the service is registered, the next step is that a client device finds it, therefore requires a *SDP* connection to the remote device (server) to find the service with the *UUID* desired, we the remote sdp server will return a list of services founded with the specified *UUID*.

searching.c

```
#include <bluetooth/bluetooth.h>
#include <bluetooth/sdp.h>
#include <bluetooth/sdp_lib.h>
int main(int argc, char **argv)
{
 uint32_t svc_uuid_int[] = { 0, 0, 0, 0xABCD };
 uuid_t svc_uuid;
 int err, num;
 bdaddr_t target;
 sdp_list_t *response_list = NULL, *search_list,
 *attrid_list;
 sdp_session_t *session = 0;
 str2ba( "00:0C:78:31:FC:6B", &target );

 // connect to the SDP server on the remote machine
 session = sdp_connect( BDADDR_ANY, &target,
 SDP_RETRY_IF_BUSY );

 // specify the UUID we're searching for
 sdp_uuid128_create( &svc_uuid, &svc_uuid_int );
 search_list = sdp_list_append( NULL, &svc_uuid );

 // specify that we want a list of all the matching
 // applications' attributes
 uint32_t range = 0x0000ffff;
 attrid_list = sdp_list_append( NULL, &range );

 // get a list of service records (UUID 0xABCD)
 err = sdp_service_search_attr_req(session, search
 _list, SDP_ATTR_REQ_RANGE, attrid_list,
 &response_list);

 sdp_list_t *r = response_list;
 num = sdp_list_len(r);

 //if there isn't an error and there are
 //services found
 if((err==0) && (num> 0)){
 printf("\nService(s) found in %s.\n", target);

 // go through each of the service records
 for (; r; r = r->next ) {
 sdp_record_t *rec = (sdp_record_t*)
 r->data;
 printf("found service record 0x%x\n",
 rec->handle);
 sdp_record_free( rec );
 }
 sdp_close(session);
}
```

5.4 Using all the examples

Finally there is a program using the tools displayed in the previous examples, consists of 3 steps:

1. As a first step the device server records the service (registering.c)
2. Server application (server_blue.c) to listen the clients requests for the service and then exchange information.
3. Client application (client_blue.c) to search for nearby devices and search the service in each one, if the service is found, the client shares information with the server, otherwise continues with the search.

As the following examples are a compilation of previous (practically the same code), the code will not be shown again, only the little changes applied that are explained below.

server_blue.c is practically the same as server.c, the difference is that server_blue.c has a service registered, and besides, now we want to exchange a structure data instead strings, we only have to apply the following code lines in server.c.

```
typedef struct data
{
 char name[50];
 int id;
}my_data,rec;
```

we create rec and my_data to store the data received and my data.

```
struct data my_data,rec;
//setting my data
strcpy(my_data.name, "Client");
my_data.id=24;
```

we receive the client data that is stored in rec,

```
bytes_read=read(client, &rec, sizeof(my_data));
```

print the data received

```
printf("Hello my name is %s and my id is %d \n",
 rec.name, rec.id);
```

and send my_data to client:

```
write(client,&my_data,sizeof(my_data));
```

Exchanging structure is not so complicated, we only have to be careful when you create structures that match the types of data with which we are using.

client_blue.c applies the basic principle to identify nearby devices through a scan (scan.c), in each device search a service (searchin.g), and if it find it, a connection is created (client.c) to performs an action (exchange structures). It is necessary to place the code of the 3 examples in one [7]. The explain is below:

The first step is to declare the struct data my_data and rec used in server_blue.c, then we scan for nearby devices using the scan.c code

```
num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL,
 &ii, flags);
```

where, for each remote device found:

```
for (i=0; i < num_rsp; i++)
```

a *SDP* connection is made using the *Bluetooth* address provided (searching.c)

```
session = sdp_connect(BDADDR_ANY, &(ii+i)->bdaddr,

 SDP_RETRY_IF_BUSY)
```

[7] Both programs can be run on a single computer (server.c and client.c or server_blue.c and client_blue.c), just identify the role that each device plays.

for search the service with the *UUID* 0xABCD.

```
err = sdp_service_search_attr_req( session, search_list,
 SDP_ATTR_REQ_RANGE, attrid_list, &response_list);
```

if the service is found,

```
if((err==0) && (num> 0))
```

a *L2CAP* socket connection is made using the same *Bluetooth* address (server.c),

```
s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
status = connect(s, (struct sockaddr *)&address,
 sizeof(address));
```

and then exchange data structures:

```
if(write(s,&my_data,sizeof(my_data))==-1){
 perror("write");
}
bytes_read = read(s, &rec, sizeof(my_data));
```

The loop ends with this remote device and proceeded to continue with the next on the num_rsp list, until all go up and exit the program.

There are too much applications we can develop with these basic examples. It is necessary to mention that these programs are at a very basic level, for which there are many tools and opportunities offered by *C++* and *BlueZ* that aren't in this paper. The purpose of this paper is to involve new users in programming with *Bluez / C++* in a quick and easy way, forthcoming work is intended to bring this approach to one much more advanced .

6 Conclusions and future work

The proliferation of mobile communication devices at low cost and low power consumption has opened the possibility of developing applications that take advantage of new models of communication, such as the ad hoc networks. The *Bluetooth* technology is having a greater presence in the market for mobile devices. Unfortunately there is little documentation about it. This article presented the characteristics of *Bluetooth* and the *Bluetooth* protocol stack for Linux called *BlueZ*, as well as some of the most important utilities for their use. As future work is developing an application that uses *BlueZ* and programming languages C, *C++* and *Qt* that serve as reference for the development of future applications and future *BlueZ* programmers.

References

1. Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 3.* Prentice Hall in association with Trolltech Press, 2004.
2. Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version.* Prentice Hall, November 2000.
3. P. D. Garner. Mobile bluetooth networking: Technical considerations and applications. *3G Mobile Communication Technologies*, page 274, June 2003.
4. The Free Software Fundation Inc. GNU General Public License. *http://www.gnu.org/copyleft/gpl.html.*
5. Albert S. Huang and Larry Rudolph. *Bluetooth Essentials for Programmers.* Cambridge University Press, 1st edition, September 2007.

6. Linux Kernel Patches. *http://www.holtmann.org/linux/kernel.*
7. BlueZ Official Web Site. *http://www.bluez.org.*
8. The IEEE 802.15.1 Standard. *http://www.ieee802.org/15/pub/ TG1.html.*
9. Bluetooth's Official Website. *http://www.bluetooth.com.*