4.2. RFCOMM sockets

As with Python, establishing and using RFCOMM connections boils down to the same socket programming techniques we already know how to use for TCP/IP programming. The only difference is that the socket addressing structures are different, and we use different functions for byte ordering of multibyte integers. Example 4-2 and Example 4-3 show how to establish a connection using an RFCOMM socket, transfer some data, and disconnect. For simplicity, the client is hard-coded to connect to ``01:23:45:67:89:AB".

Example 4-2. rfcomm-server.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc 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_STREAM, BTPROTO_RFCOMM);

    // bind socket to port 1 of the first available
    // local bluetooth adapter
    loc_addr.rc_family = AF_BLUETOOTH;
    loc_addr.rc_bdaddr = *BDADDR_ANY;
    loc_addr.rc_channel = (uint8_t) 1;
    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.rc_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);
    }

    // close connection
    close(client);
    close(s);
    return 0;
}
```

Example 4-3. rfcomm-client.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc addr = { 0 };
    int s, status;
    char dest[18] = "01:23:45:67:89:AB";

    // allocate a socket
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

    // set the connection parameters (who to connect to)
    addr.rc_family = AF_BLUETOOTH;
    addr.rc_channel = (uint8_t) 1;
    str2ba( dest, &addr.rc_bdaddr );

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

    // send a message
    if( status == 0 ) {
        status = write(s, "hello!", 6);
    }

    if( status < 0 ) perror("uh oh");

    close(s);
    return 0;
}
```

Most of this should look familiar to the experienced network programmer. As with Internet programming, first allocate a socket with the `socket` system call. Instead of `AF_INET`, use `AF_BLUETOOTH`, and instead of `IPPROTO_TCP`, use `BTPROTO_RFCOMM`. Since RFCOMM provides the same delivery semantics as TCP, `SOCK_STREAM` can still be used for the socket type.

4.2.1. Addressing structures

To establish an RFCOMM connection with another Bluetooth device, incoming or outgoing, create and fill out a `struct sockaddr_rc` addressing structure. Like the `struct sockaddr_in` that is used in TCP/IP, the addressing structure specifies the details of an outgoing connection or listening socket.

```
struct sockaddr_rc {
        sa_family_t     rc_family;
        bdaddr_t        rc_bdaddr;
        uint8_t         rc_channel;
};
```

The `rc_family` field specifies the addressing family of the socket, and will always be `AF_BLUETOOTH`. For an outgoing connection, `rc_bdaddr` and `rc_channel` specify the Bluetooth address and port number to connect to, respectively. For a listening socket, `rc_bdaddr` specifies the local Bluetooth adapter to use, and is typically set to `BDADDR_ANY` to indicate that any local Bluetooth adapter is acceptable. For listening sockets,

`rc_channel` specifies the port number to listen on.

4.2.2. A note on byte ordering

Since Bluetooth deals with the transfer of data from one machine to another, the use of a consistent byte ordering for multi-byte data types is crucial. Unlike network byte ordering, which uses a big-endian format, Bluetooth byte ordering is little-endian, where the least significant bytes are transmitted first. BlueZ provides four convenience functions to convert between host and Bluetooth byte orderings.

```
unsigned short int htobs( unsigned short int num );
unsigned short int btohs( unsigned short int num );
unsigned int htobl( unsigned int num );
unsigned int btohl( unsigned int num );
```

Like their network order counterparts, these functions convert 16 and 32 bit unsigned integers to Bluetooth byte order and back. They are used when filling in the socket addressing structures, communicating with the Bluetooth microcontroller, and when performing low level operations on transport protocol sockets.

4.2.3. Dynamically assigned port numbers

For Linux kernel versions 2.6.7 and greater, dynamically binding to an RFCOMM or L2CAP port is simple. The `rc_channel` field of the socket addressing structure used to bind the socket is simply set to 0, and the kernel binds the socket to the first available port. Unfortunately, for earlier versions of the Linux kernel, the only way to bind to the first available port number is to try binding to *every* possible port and stopping when `bind` doesn't fail. The following function illustrates how to do this for RFCOMM sockets.

```
int dynamic_bind_rc(int sock, struct sockaddr_rc *sockaddr, uint8_t *port)
{
    int err;
    for( *port = 1; *port <= 31; *port++ ) {
        sockaddr->rc_channel = *port;
        err = bind(sock, (struct sockaddr *)sockaddr, sizeof(sockaddr));
        if( ! err || errno == EINVAL ) break;
    }
    if( port == 31 ) {
        err = -1;
        errno = EINVAL;
    }
    return err;
}
```

The process for L2CAP sockets is similar, but tries odd-numbered ports 4097-32767 (0x1001 - 0x7FFF) instead of ports 1-30.

4.2.4. RFCOMM summary

Advanced TCP options that are often set with `setsockopt`, such as receive windows and the Nagle algorithm, don't make sense in Bluetooth, and can't be used with RFCOMM sockets. Aside from this, the byte ordering, and socket addressing structure differences, programming RFCOMM sockets is virtually identical to programming TCP sockets. To accept incoming connections with a socket, use `bind` to reserve operating system resource, `listen` to put it in listening mode, and `accept` to block and accept an incoming connection. Creating an outgoing connection is also simple and merely involves a call to `connect`. Once a connection has

been established, the standard calls to `read`, `write`, `send`, and `recv` can be used for data transfer.

Prev
Bluetooth programming in C with BlueZ

Home
Up

Next
L2CAP sockets