Introduction to Socket Programming for UDP communication

G. Aiello, M. Ahmadi, E. Zoller, M. Eugster, I. Susic, and G. Rauter*

***Bio Inspired RObotics in Medicine Lab**, University of Basel, Basel*

January 2017

E-mail: gregorio.aiello@unibas.ch

Abstract

This brief tutorial will cover an introduction to Socket programming and UDP communication as well that I have found online (Socket Programming). First I will explain what is a socket and why we need it, then I will go through the client-server model, and finally the UDP socket is presented with examples related to my project (i.e. UDP communication between C++ and C#).

Socket

A socket is the mechanism that most popular operating systems provide to give programs access to the network. It allows messages to be sent and received between applications (unrelated processes) on different networked machines. The sockets mechanism has been created to be independent of any specific type of network. IPv4, however, is by far the most dominant network and the most popular use of sockets.

UDP

The UDP communication provides service to application layer (read about TCP/IP architecture in case you don't know what the application layer is) by using the service provided by network layer, the quality of this protocol lies in hiding the physical layer, in such a way the processing complexities and the different architectures are hidden below the transport layer.

Addressing

When a packet arrives to a network layer, how does it know which application he needs to go to? there are $2^{16} = 65536$ ports $(0 - 65535)$ on one machine, every port is linked to only one (1!) application but one application may use multiple ports (i.e. FTP uses ports 20-21), the list of ports is divided in three categories:

- Common ports: from 0 through 1023
- Registered ports: from 1024 through 49151
- Dynamic or Private ports: from 49152 through 65535

Steps

There are a few steps involved in using sockets:

1. Create a socket
2. Naming a socket
3. On the server, wait for an incoming connection
4. On the client, connect to the server's socket

5. Send and receive messages

6. Close the socket

Create a socket

A socket, **s**, is created with the socket system call:

```
int s = socket(domain, type, protocol)
```

Where all the parameters as well as the return value are int valued variables, in particular:

- domain, or address family: communication domain in which the socket should be created. Some of address families are **AF_INET** (IP), **AF_INET6** (IPv6), **AF_UNIX** (local channel, similar to pipes), **AF_ISO** (ISO protocols), and **AF_NS** (Xerox Network Systems protocols).

- type: type of service, this is selected according to the properties required by the application: **SOCK_STREAM** (virtual circuit service), **SOCK_DGRAM** (datagram service), **SOCK_RAW** (direct IP service).

- protocol: indicate a specific protocol to use in supporting the sockets operation. This is useful in cases where some families may have more than one protocol to support a given type of service. The return value is a file descriptor (a small integer). The analogy of creating a socket is that of buying a physical mail box (Fig. 1)

For our UDP/IP sockets, we will specify the IPv4 address family (AF_INET) and datagram service (SOCK_DGRAM). Since there's only one form of datagram service, there are no variations of the protocol, so the last argument, protocol, is zero. The code for creating a UDP socket looks like this:

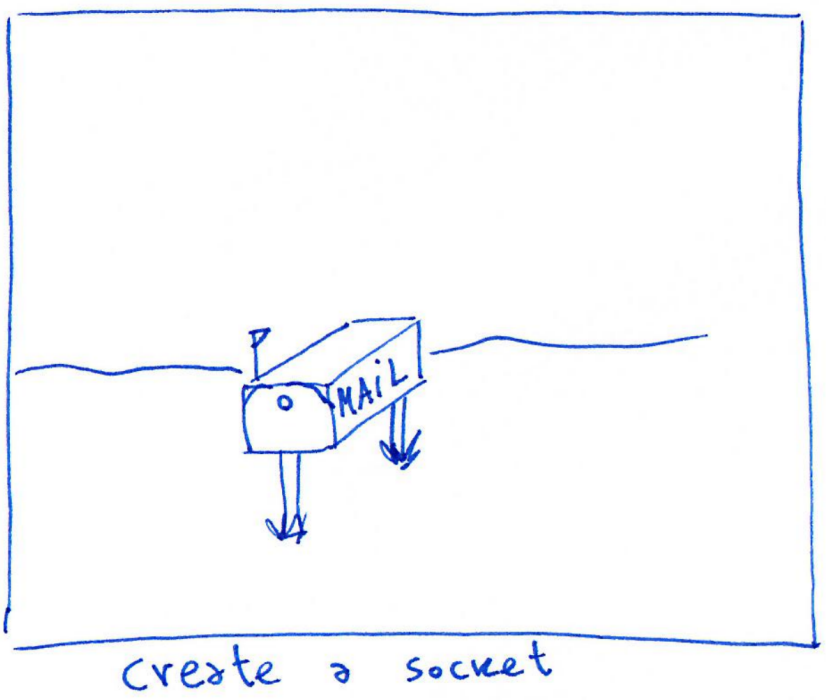

Figure 1: Analogy between socket and mail box.

```
...
if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
perror("cannot create socket");
return 0;
}
...
```

Naming a socket

When we talk about naming a socket, we are talking about assigning a transport address to the socket (a port number, yes the port numbers that we have already discussed above). In sockets, this operation is called binding an address and the **bind** system call is used to do this. The analogy is that of assigning an address to a mailbox (Fig. 2). The transport address is defined in a socket address structure. Because sockets were designed to work with various different types of communication interfaces, the interface is very general. Instead of

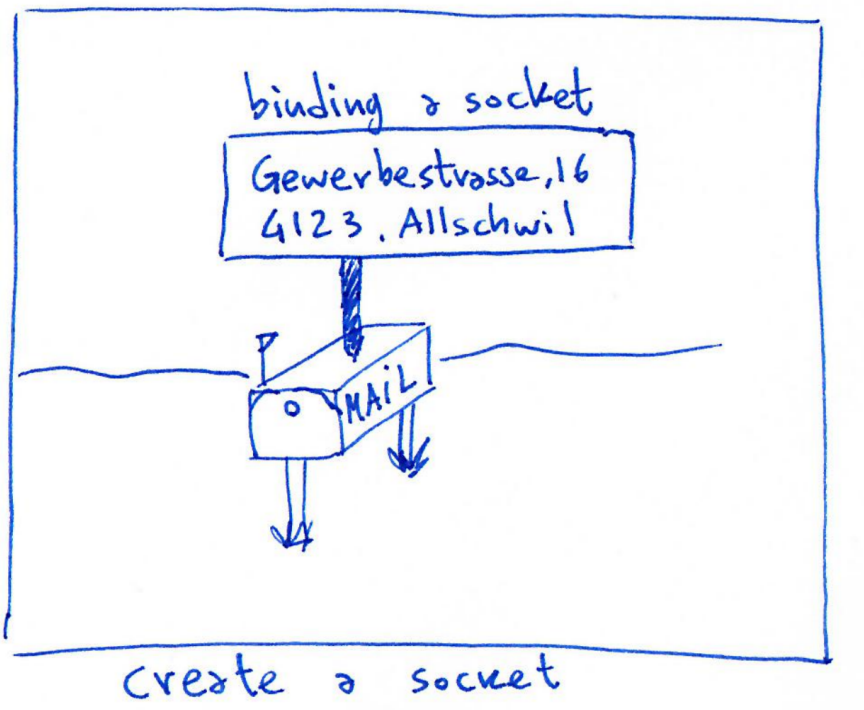

Figure 2: Analogy between bind and mail box.

accepting, say, a port number as a parameter, it takes a **sockaddr** structure whose actual format is determined on the address family (type of network) you're using. For example, if you're using UNIX domain sockets, bind actually creates a file in the file system. The system call for bind is:

```
#include <winsock2.h>
...
int bind(_In_ SOCKET s, _In_ const struct sockaddr *name, _In_ int namelen);
...
```

with:

- s: descriptor identifying the socket.
- *name: pointer to a sockaddr structure of the local address to assign to the bound socket .
- namelen: length, in bytes, of the value pointed to by the name parameter.

and in particular:

```
struct sockaddr_in {
        short            sin_family;
        u_short          sin_port;
        struct in_addr   sin_addr;
        char             sin_zero[8];
};
```

where:

- sin_family: address family we used when we set up the socket. In our case, it's AF_INET.
- sin_port: port number (the transport address). You can explicitly assign a transport address (port) or allow the operating system to assign one. If you're a client and don't need a well-known port that others can use to locate you (since they will only respond to your messages), you can just let the operating system pick any available port number by specifying port 0. If you're a server, you'll generally pick a specific number since clients will need to know a port number to which to address messages.
- sin_addr: address for this socket. This is just your machine's IP address. With IP, your machine will have one IP address for each network interface. For example, if your machine has both Wi-Fi and ethernet connections, that machine will have two addresses, one for each interface. Most of the time, we don't care to specify a specific interface and can let the operating system use whatever it wants. The special address for this is 0.0.0.0, defined by the symbolic constant INADDR_ANY.

Data preparation

In order to prepare the data to be in the network we will use **htonl** and **htons** references. These convert four-byte and two-byte numbers into network representations. Integers are

stored in memory and sent across the network as sequences of bytes. There are two common ways of storing these bytes: **big endian** and **little endian** notation. Little endian representation stores the least-significant bytes in low memory. Big endian representation on the other hand stores the least-significant bytes in high memory. For example the Intel x86 family uses the little endian format while the PowerPC (used by Macs before their switch to the Intel architecture) use the big endian format. Internet headers standardized on using the big endian format. To keep code portable and to keep you from worrying about this a few convenience macros have been defined:

- **htons**: host to network - short : convert a number into a 16-bit network representation. This is commonly used to store a port number into a sockaddr structure.
- **htonl**: host to network - long : convert a number into a 32-bit network representation. This is commonly used to store an IP address into a sockaddr structure.
- **ntohs**: network to host - short : convert a 16-bit number from a network representation into the local processor's format. This is commonly used to read a port number from a sockaddr structure.
- **ntohl**: network to host - long : convert a 32-bit number from a network representation into the local processor's format. This is commonly used to read an IP address from a sockaddr structure.

For processors that use the big endian format, these macros do absolutely nothing. For those that use the little endian format (most processors, these days), the macros flip the sequence of either four or two bytes. Using the macros, however, ensures that your code remains portable regardless of the architecture to which you compile. In the above code, writing htonl(INADDR_ANY) and htons(0) is somewhat pointless since all the bytes are zero anyway but it's good practice to remember to do this at all times when reading or writing network data.

Send Messages

With UDP, our sockets are connectionless hence we can send messages immediately without doing anything else. Since we do not have a connection, every message has to be addressed to its destination. The address is identified through the sockaddr structure. The function to send is **sendto()** and is defined as:

```
int sendto(
  _In_       SOCKET                 s,
  _In_ const char                   *buf,
  _In_       int                    len,
  _In_       int                    flags,
  _In_       const struct sockaddr  *to,
  _In_       int                    tolen
);
```

The first parameter, s, is the socket that was created with the socket system call and named via bind. The second parameter, *buf, provides the starting address of the message we want to send. len is the number of bytes that we want to send. The flags parameter is 0 and not useful for UDP sockets. The *to defines the destination address and port number for the message. It uses the same sockaddr_in structure that we used in bind to identify our local address. As with bind, the final parameter is simply the size, in bytes, of the address pointed to by the to parameter.

The server's address will contain the IP address of the server machine as well as the port number that corresponds to a socket listening on that port on that machine. The IP address is a four-byte (32 bit) value in network byte order (see htonl above).

Receive Messages

A server can immediately listen for messages once it has a socket since UDP communication is connectionless. We use the **recvfrom()** system call to wait for an incoming datagram on a specific transport address (IP address and port number).

The recvfrom call has the following syntax:

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

s is the descriptor identifying a bound socket, *buf is a buffer for the incoming data, len is the length, in bytes, of the buffer pointed to by the *buf parameter, flags are a set of options that modify the behavior of the function call beyond the options specified for the associated socket, *from is an optional pointer to a buffer in a sockaddr structure that will hold the source address upon return, *fromlen is an optional pointer to the size, in bytes, of the buffer pointed to by the from parameter.

The recvfrom call returns the number of bytes that were read into *buf.

Close the Socket

Since there is no concept of a connection in UDP, there is no need to call shutdown. However, the socket still uses up a file descriptor in the kernel, so we can free that up with the close system call just as we do with files.

```
...
close(fd);
...
```

with **fd** name of the socket.