Practice class 1

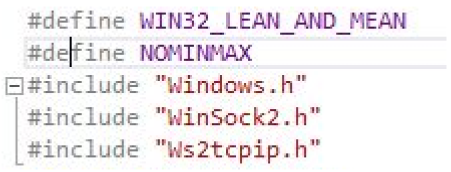
Berkeley Sockets

Berkeley sockets is an application programming interface (API) to work with sockets, and it is used with the aim of inter-process communication (IPC), either to allow communication among processes within the same machine or different machines connected to a network. Actually, Berkeley sockets is a broadly standardized sockets API, and it is natively available in many platforms. However, its interface is not completely uniform over different operating systems. There are a few subtle differences in the way they expose its functionality through the use of its data structures and functions. In these practice classes we will use Windows, and more specifically, the Visual C++ IDE. Therefore, the explanations and examples described in the following classes will refer to the API provided by this environment.

Table of contents

API headers

To have available all data structures and functions in the sockets API in the following sections, we need to include the following header files:



```
#define WIN32_LEAN_AND_MEAN
#define NOMINMAX
#include "Windows.h"
#include "WinSock2.h"
#include "Ws2tcpip.h"
```

Initialization and clean up

In windows, to initialize and clean up the library internal structures, we need to call a couple of functions explicitly.

NOTE: This is one of the different aspects between Windows and other platforms. For instance, other operating systems such as Linux or Mac OS X, the library is active by default and there is no special need to call initialization or clean up functions.

In Windows, however, the following function needs to be called to initialize the sockets library, typically at the beginning of our program.

```
WSADATA wsaData;
int iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
if (iResult != NO_ERROR)
{
 // Log and handle error
 return false;
}
```

To perform the clean up, we need to call the following function:

```
int iResult = WSACleanup();
```

Sockets creation

To create sockets, the library defines the following function:

```
SOCKET socket(int af, int type, int protocol);
```

As we can see, the function returns an object of the opaque type SOCKET. If the function fails for whichever reason, and the socket could not be created, the returned value equals the constant *INVALID_SOCKET* (defined in the included headers).

We will usually use this function to create two different kinds of sockets: UDP sockets and TCP sockets.

To create UDP sockets:

```
SOCKET s = socket(AF_INET, SOCK_DGRAM, 0);
```

To create TCP sockets:

```
SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
```

In these calls, the argument *AF_INET* refers to the family of addresses IPv4. If, at some point, the use of sockets belonging to the IPv6 address family were needed, we should pass the value *AF_INET6*.

The second argument, can either have the values SOCK_DGRAM or SOCK_STREAM, depending on the type of socket we want to create (UDP or TCP respectively).

NOTE: With *SOCK_DGRAM* (for UDP sockets) discrete datagrams (that is, independent packets of data) will be send through the network. With *SOCK_STREAM* (for TCP sockets) the data sent through the network will be packed into a contiguous stream of data that needs to be separated by the application code.

To stop using a socket, we use the closesocket function (it is usually invoked when we don't need using the socket anymore, or at the end of the program):

```
int closesocket(SOCKET s);
```

Socket Address

To connect different processes to each other, some of them need to have an associated address. This is logical because, for instance, if we think about the postal service, we cannot send a letter without knowing the destination address. In this case, at least the receiver needs to have an associated address.

To represent addresses, the library defines the following data structure:

```
struct sockaddr_in{
        short sin_family; // Address family
        unsigned short sin_port; // Port
        struct in_addr sin_addr; // IP Address
        char sin_zero[8]; // Not used
};
```

By giving the value *AF_INET* to the field *sin_family*, we will use the IPv4 address family:

```
struct sockaddr_in bindAddr;
bindAddr.sin_family = AF_INET; // IPv4
```

The field *sin_port* will contain the port through which the connection will be done. Note that the bytes of this number need to be lay out in a specific order (network order). To that aim, the function htons (host to network short) does this: it takes a short integer and re-arranges its bytes to match network order (also big endian).

```
bindAddr.sin_port = htons(port); // Port
```

The remaining field, sin_addr, contains the actual IP address. We will usually use this field in two different ways:

1. Specifying a destination host IP (could be the same machine, or localhost).
2. Specifying a local IP to listen incoming data.

Destination IP address (typically for clients)

On the other hand, if we are defining the IP address of a destination host we want to connect to, we will use the function *inet_pton*, which converts a formatted string into the appropriate IP address library's internal data format:

```
struct sockaddr_in remoteAddr;
remoteAddr.sin_family = AF_INET; // IPv4
remoteAddr.sin_port = htons(port); // Port
const char *remoteAddrStr = "127.0.0.1"; // Not so remote... :-P
inet_pton(AF_INET, remoteAddrStr, &remoteAddr.sin_addr);
```

IP address for bind() (typically for servers)

To specify a generic local IP address (to any local networking device) where remote hosts will be able to send data, we will typically use the following value:

```
bindAddr.sin_addr.S_un.S_addr = INADDR_ANY;
```

Here, *INADDR_ANY* means that this IP address matches any IP address available in the machine. Note that a machine could have several network interfaces, so we could specify a specific device by giving the specific it specific IP. However, in many cases, a machine will only have a single network interface, and using *INADDR_ANY* is convenient for simplicity, and to avoid hard-coding the local address.

Bind

So far we know how to specify IP addresses. Now, we need to tell the operating system that a certain socket is bound to a specific local address. This operation is called *binding*. To bind a socket to a specific IP address and port we use the following function:

```
int bind(SOCKET s, const struct sockaddr * address, int address_len);
```

In this call, *address* is a pointer to an IP address (as described in the previous section) we want to bind the socket to (recall that the machine could have several network interfaces, each one with its own IP address). It is worth clarifying that this address is not related to the address of any potential remote host. It is rather the address we enable in our machine to receive data from remote hosts.

NOTE: At this point, we can see that *bind* is an operation only necessary when configuring a socket to publicly receive incoming data or connections from remote hosts. If we are setting up a socket that does not expect receiving data from arbitrary hosts, and will initiate the communication with remote hosts, calling *bind* it is not necessary at all.

If we have a look at the data structure pointed to by *address* (*sockaddr*), we can see that the type does not match the data type we used in the previous section in order to define IP addresses (*sockaddr_in*). The sockets API exposes its functionality in the C language. Unlike C++, C has more limited polymorphism capabilities (inheritance, virtual methods…). The reason for using a different data type for the parameter *address* is to use a generic type (as if it was a base class in C++) that allows passing different types of addresses (not only IPv4). To pass our defined address (of type sockaddr_in) to the *bind* function, we simply need to cast the pointer type:

```
struct sockaddr_in bindAddr;
...
int res = bind(s, (const struct sockaddr *)&bindAddr, sizeof(bindAddr));
```

address_len needs to contain the size of the concrete data type of the parameter *address* (in our case, *sizeof(sockaddr_in)*).

NOTE: Sometimes, if a socket was not properly closed, the operating system might think that the previously used IP address and port are still in use. In this case, calling *bind* with the same IP address and port will return *SOCKET_ERROR* by default, indicating that it is not possible binding the given socket to the specified pair address/port. To avoid this, before invoking *bind*, the socket can be configured to force reusing the given address/port in the function *bind*:

```
int enable = 1;
res = setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (const char*)&enable, sizeof(int));
if (res == SOCKET_ERROR) {
  // Log and handle error
}
```

UDP sockets

To work with UDP sockets, we can start sending and receiving datagrams just after creating the socket. Only if a socket needs to receive datagrams from arbitrary hosts, the socket have to be previously bound (function *bind*). But if the socket is intended to initiate communication with other hosts, the socket is automatically bound on the first data transmission.

sendto

To send datagrams through a UDP socket, we use the *sendto* function:

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

Where *buf* is a data buffer with data to be sent, and *len* is the size of this data in bytes. *Flags*, which can be 0 at the moment, are a few extra options to take into account when transmitting data. By last, *to* and *tolen* are a pointer to the address where the datagram has to be sent, and the size in bytes of the address data structure.

recvfrom

To receive data through UDP sockets, we use the function *recvfrom*:

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

Where *buf* is a data buffer where the received data will be stored, and *len* is the maximum number of bytes to receive (typically the size of the buffer). *Flags*, at the moment can have value 0, and they can specify some extra options to take into account in the operation. *From*, in this case, is a pointer to an address structure where the remote host's address will be stored, telling us where the datagram comes from, and *fromlen* is a pointer to an integer with the size in bytes of the address data type.

NOTE: it may seem that *fromlen* is an output parameter. However, the pointed integer needs to have the size of the address data type (*sizeof(sockaddr_in)*) from the beginning.

Using the UDP protocol, datagrams are sent immediately just after invoking sendto as an atomic chunk of data. If the destination receives the datagram, it receives it completely.

NOTE: However, UDP does not provide reliability mechanisms, so there is not guarantee that sent packets will be received by the destination host. There is also no guarantee that the order in which packets are received is the same as the order in which they were sent.

NOTE: By default, the functions *sendto* and *recvfrom* are blocking functions, this, if the socket is not ready to send data or there is not data to receive the current thread will block on these calls until the operation can be finished.

TCP sockets

With TCP sockets, it is necessary establishing a connection between two points before starting sending and receiving data. Before proceeding to describe the API functions regarding TCP sockets, it is worth mentioning that they can be used in two different ways:

1. Listen mode (listen sockets)
2. Normal mode (connected sockets)

listen / accept

In order to receive incoming connections from remote hosts, a socket needs to enter in listen mode. After calling *bind*, the function *listen* can be invoked to configure the socket as a listen socket:

```
int listen(SOCKET s, int backlog);
```

Where *backlog* is the maximum number of simultaneous incoming connections allowed.

Once the socket is in listen mode, it can accept incoming connections by means of the *accept* function:

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

Where, similarly to the function *recvfrom*, the parameter *addr* is the address and port of the remote accepted socket, and *addrlen* is the size of the address data type in bytes. This function returns a new socket (a connected socket), already connected to the remote hosts, and it is prepared to receive and send data by means of the functions *recv* and *send*.

NOTE: By default, if no incoming connections are accepted, the function *accept* blocks the current thread either until a new incoming connection from a remote host is received or until the default timeout is reached.

connect

Any process that wants to initiate a new connection has to use the *connect* function:

```
int connect(SOCKET s, const struct sockaddr * addr, int addrlen);
```

Calling connect initiates the TCP *three-way handshake* process, sending the initial SYN packet. If the destination host has a socket listening through the appropriate port, the handshake will continue if it is accepting incoming connection with the *accept* function.

NOTE: By default, *connect* is a blocking function, and it will block the current thread until the connection is either accepted or the default timeout is reached.

send

To send data through a TCP socket, we use the function *send*:

```
int send(SOCKET s, const char * buf, int len, int flags);
```

The parameters *buf*, *len* and *flags* are analogous to those parameters in the UDP-sockets function *sendto*, specifying a data buffer, the amount of bytes in the buffer to send, and a series of flags to apply to the send operation. On success, the function *send* either returns the number of bytes sent, which can be less than the parameter *len* if there is not enough space in the internal send buffer. In case of error, it returns the constant *SOCKET_ERROR*.

recv

To receive data through a TCP socket, the function *recv* is used:

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

The parameters *buf*, *len* and *flags* are analogous to those in the UDP-sockets function *recvfrom*. On success, *recv* returns the number of bytes received, which can be less than the parameter *len* if the received datagram is smaller than the provided data buffer. The function returns *SOCKET_ERROR* in case of error.

NOTE: If the returned value is 0, being *len* > 0, it means that the remote socket has sent a *FIN* packet, notifying its disconnection.

NOTE: If the returned value is 0, begin len == 0, if means that there are data to receive. This method can be used to consult whether or not there is data to receive without blocking the current thread.

NOTE: By default, the functions *send* and *recv* are blocking functions, this, if the socket is not ready to send data or there is not data to receive the current thread will block on these calls until the operation can be finished.

Error checking

Most functions in the library return -1 in the case of error. In Windows, we can use the constant *SOCKET_ERROR* (defined as -1 in the headers) instead of the magic number -1. When a function returns *SOCKET_ERROR*, we can obtain a more specific error code using the following function:

```
int WSAGetLastError();
```

This function returns the last error code related to the sockets API in the current thread. Thus, it is important checking for errors immediately after calling any function in the library returning *SOCKET_ERROR*.

A few functions, instead of an integer number, return objects of the type *SOCKET*, like the function *socket* (to create sockets) or *accept* (to accept incoming TCP connections from remote hosts). In this case, to verify whether or not the function succeed, the returned value can be compared with the constant *INVALID_SOCKET*.

The following link describes all possible error codes returned by the function *WSAGetLastError()*. It will be useful to know some of them in particular. For instance, *WSAEWOULDBLOCK*, that we will see in the following classes.

https://msdn.microsoft.com/en-us/library/windows/desktop/ms740668(v=vs.85).aspx

Descriptive error messages...

The following function may help to reveal more descriptive error messages. It is using the previously described function *WSAGetLastError()* to obtain the last error generated by the sockets library. It also uses the function *FormatMessageW()* so that Windows can generate a string with the description of the error in text format.

```
void printWSErrorAndExit(const char *msg)
{
        wchar_t *s = NULL;
        FormatMessageW(
                FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM
                |FORMAT_MESSAGE_IGNORE_INSERTS,
                NULL,
                WSAGetLastError(),
                MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                (LPWSTR)&s,
                0,
                NULL);
        fprintf(stderr, "%s: %S\n", msg, s);
        LocalFree(s);
        system("pause");
        exit(-1);
}
```