Lesson 1 - Socket Programming
An Introduction to Sockets

Summary

We are going to introduce some of the functions and data structures you will come across when programming with sockets.

New Concepts

Sockets, Stream Sockets, Datagram Sockets

Brief Overview of Networking

A socket is a mechanism for allowing communication between processes, be it programs running on the same machine or different computers connected on a network. More specifically, Internet sockets provide a programming interface to the network protocol stack that is managed by the operating system. Using this API, a programmer can quickly initialise a socket and send messages without having to worry about issues such as packet framing or transmission control.

There are a number of different types of sockets available, but we are only really interested in two specific Internet sockets. These are:

- Stream sockets
- Datagram sockets

What differentiates these two socket types is the transport protocol used for data transmission. A stream socket uses the Transmission Control Protocol (TCP) for sending messages. TCP provides an ordered and reliable connection between two hosts. This means that for every message sent, TCP guarantees that the message will arrive at the host in the correct order. This is achieved at the transport layer so the programmer does not have to worry about this, it is all done for you.

A datagram socket uses the User Datagram Protocol (UDP) for sending messages. UDP is a much simpler protocol as it does not provide any of the delivery guarantees that TCP does. Messages, called datagrams, can be sent to another host without requiring any prior communication or a connection having been established. As such, using UDP can lead to lost messages or messages being received out of order. It is assumed that the application can tolerate an occasional lost message or that the application will handle the issue of retransmission.

There are advantages and disadvantages to using either protocol and it will be highly dependant of the application context. For example, when transferring a file you want to ensure that, upon receipt, the file has not become corrupted. TCP will handle all the error checking and guarantee that it will arrive as you sent it. On the other hand, imagine you are sending 1000 messages detailing player position data every second in a computer game. The application will be able to tolerate missing messages here so UDP would be more suitable.

Addressing

Using Internet Sockets, we can identify a host using an IP address and a port number. The IP address uniquely identifies a machine while the port number identifies the application we want to contact at that machine. There are a range of well known port numbers, such as port 80 for HTTP, in the range 0 - 1023. When choosing port numbers, anything above the well know port number range should be fine, but you cannot guarantee that another application will not be already using it.

Programming with Sockets

Now we are going to introduce the Windows Sockets API.

Note on Error Handling

Within this section we will not be discussing error handling in too much depth. All functions used have a wide range of different error messages associated with them. Discussing each error for each function is beyond the scope of these tutorials. Error values for each function can easily be found in the API.

Windows Specific Functions

All the methods for declaring and using sockets are available in two header files:

```
#include <winsock2.h>
#include <ws2tcpip.h>
```

Windows Sockets header file

You will also need to link the `ws2_32.lib` library in your project settings.

The first thing we have to do before being able to declare or use a socket is make a call to the `WSAStartup()` method.

```
int WSAStartup(
   WORD      version, // highest version of winsock
   WSADATA   *data    // pointer to WSADATA struct
);
```

WSAStartup() function declaration

This method must be called first before any other calls involving the sockets library. This method simply allows the programmer to define the version of the Windows sockets specification to be used and stores the result in the `WSADATA` struct.

Complimentary to `WSAStartup()`, we also have `WSACleanup()` which should be called at the end of your program when you no longer require the use of the socket library.

```
int WSACleanup(void);
```

WSACleanup() function declaration

You will also find `WSAGetLastError()` useful when you need to perform some error checking or debug your code:

```
int WSAGetLastError(void);
```

WSAGetLastError() function declaration

This method takes no parameters and returns the error status for the last socket operation that failed. The range of error codes that can be returned is too extensive to discuss here. You find a detailed list of all return codes and values in the Sockets documentation.

Addressing Information

Having initialised the API we now need to set up the address we will be listening on (if we are the server) or the one we will be sending to (if we are the client). The function `getaddrinfo()` is going to help us out here.

```
int getaddrinfo(
   const char            *name,    // host name or IP
   const char            *service, // service name or port
   const struct addrinfo *hints,   // socket info
   struct addrinfo       *result   // result struct
);
```

getaddrinfo() function declaration

We either provide `NULL` to the name parameter if we are creating our server and in the case of the client we want to provide a host name string or IP address. The service parameter takes either a string service name (e.g. "http" translates to port 80) or the port number. We then have two `addrinfo` structs. We need to populate the `hints` struct with some some option settings before calling the function. The `result` struct stores the results after the function has been executed.

```
typedef struct addrinfo {
   int              ai_flags;      // socket options
   int              ai_family;     // address family
   int              ai_socktype;   // socket type
   int              ai_protocol;   // protocol type
   size_t           ai_addrlen;    // ai_addr length
   char             *ai_canonname; // canonical name
   struct sockaddr *ai_addr;       // pointer to sockaddr struct
   struct addrinfo *ai_next;       // next addrinfo struct
};
```

addrinfo struct typedef

There are quite a few fields here so we'll look a little closer at the important ones:

- `ai_flags` - Here we can set some flags to change the socket options. There are a number of options available that you can find in the API. The only one we will need for now is the `AI_PASSIVE` flag. This indicates that the socket we will be creating will be used in a call to the function `bind()`. We must bind to a socket if we plan on listening for messages, i.e. we are creating a server program.
- `ai_family` - This is the address family we will be using. For IPv4 we would use the flag `AF_INET` and for IPv6 the flag is `AI_INET6`. We do not have to specify the address family if we want to be able to use both versions. The flag for this is `AF_UNSPEC`.
- `ai_socktype` - Here's where we can specify the socket type, stream (`SOCK_STREAM`) or datagram (`SOCK_DGRAM`).

These are the fields that you will want to populate for the `hints` struct. The `results` struct is a linked list as the call to the host may return multiple results. Often only one struct will be returned but you cannot guarantee that the one you want will be the first one if more than one has been found so it is recommended that you traverse the list and check for the one you require, making sure it has been initialised correctly.

Creating a Socket

Now for the `socket()` function.

```
SOCKET socket(
   int af,       // address family
   int type,     // socket type
   int protocol // protocol type
);
```

socket() function

As you can see from the parameter names, the `socket()` function takes a lot of the things that we have already specified when populating our `hints` struct for the `getaddrinfo()` function. We can use the results from the previous function call (stored in `results` struct) here. A successful call to this function will return a valid socket descriptor.

Binding a Socket

We have declared our socket but if we plan to receive messages using it then we need to bind to it using the `bind()` function. The purpose of binding is to associate a local address with a socket. The server always needs to call the bind function as it is going to listen for new connection requests and serve them. For clients it is enough to connect to the server without a prior call to bind.

```
int bind(
    SOCKET s,                       // socket to bind to
    const struct sockaddr *name, // local address info
    int namelen                     // length of *name
);
```

bind() function

Again we have most of the parameters already defined. The socket descriptor comes from our call to `socket()` and the `sockaddr` struct is stored within our `addrinfo` struct that we created with the call to the `getaddrinfo()` function. If no error has occurred, `bind()` returns zero.

Connecting to a Socket

For a client to send messages to a remote host (i.e. the server) we need to connect to that host.

```
int connect(
    SOCKET s,                       // socket to connect to
    const struct sockaddr *name, // remote address info
    int namelen                     // length of *name
);
```

connect() function

The function signature of `connect()` is exactly the same to `bind()`. The difference between the two functions is the `bind()` is used by the server and creates a local socket using local address information, while `connect()` uses the information of the remote host that we want to connect to. If no error has occurred, `connect()` returns zero.

Listening for Connections

For our server, having already specified our address information, created a socket and bound to it, we now need to listen for new connections requests from clients.

```
int listen(
    SOCKET s,   // socket to listen on
    int backlog // max. incoming queue length
);
```

listen() function

The backlog integer is used to specify the maximum length of the incoming connection queue. A client needs to connect to the server and this will only be successful when the server accepts the connection request. Here we can specify how many connections can be queued waiting to be accepted. This value will vary given the network conditions for the server. The flag `SOMAXCONN` can be used which instructs the underlying service provided (i.e. the operating system) to set the length of the queue to a some reasonable value. If a client attempts to connect to a server whose backlog queue is full, it will receive a connection refused error. If no error has occurred, `listen()` returns zero.

Accepting Connections

We have told our server to listen on a specific socket for incoming connection requests. When a connection request is received the server to needs to accept it to service the client.

```
SOCKET accept(
    SOCKET          s,       // socket the server listens on
    struct sockaddr *addr,   // incoming connection details
    int             *addrlen // length fo *addr
);
```

accept() function

Calling `accept()` will cause the first connection on the pending queue to be taken and a new socket created for it. This new socket will handle all interaction between the server and the client that requested the connection. The original socket still exists and is used to service new connection requests.

Sending and receiving

Now we are at the point where our server has accepted a client's connection request and created a new socket for it to accept and send messages.

```
int send(
    SOCKET s,         // socket to send with
    const char *buf, // data to send
    int len,          // length of *buf
    int flags         // sending options
);
```

send() function

```
int recv(
    SOCKET s,  // socket to receive from
    char *buf, // buffer to receive incoming data
    int len,   // length of *buf
    int flags  // receiving options
);
```

recv() function

The two functions are very similar. A socket is required to either send to or receive from and then data in the buffer is either sent or data is received into an empty buffer. The available flags can be found in the API and let you alter the actions of the associated function call. For example, we can set a flag to peek at the packet of the queue without actually removing it from the queue.

If we are using datagram sockets we have two different functions for sending and receiving.

```
int sendto(
    SOCKET s,                   // socket to send with
    const char *buf,            // data to send
    int len,                    // length of *buf
    int flags,                  // sending options
    const struct sockaddr *to, // address of target socket
    int tolen                   // length of *to
);
```

sendto() function

```
int recvfrom(
    SOCKET s,               // socket to receive from
    char *buf,              // buffer to receive incoming data
    int len,                // length of *buf
    int flags,              // receiving options
    struct sockaddr *from, // info of client we are receiving from
    int *fromlen            // length of *from
);
```

recvfrom() function

Again, these are both very similar to the stream socket functions but we have the addition of the `sockaddr` struct. For `sendto()` the struct contains address information for the host we want to send to. This is used as an input to the function so it has to be constructed prior to the call.

With the `recvfrom()` function, the `sockaddr` struct is used as an output and stores details about the machine that the message was sent from. As we do not create any connections for a datagram socket, we have no information about the host we are receiving from. After receiving a message the details (address, port) are stored in the struct.

Tidying up

When we are finished with a socket, we need to close it.

```
int closesocket(
    SOCKET s // socket to close
);
```

closesocket() function

We provide the socket descriptor of the socket we want to close. This ensures that the socket is closed cleanly and the memory it has used is reclaimed. If any subsequent function calls using a socket descriptor that has been closed will result in a socket error. Also remember that when you has completely finished using the Windows sockets API you want to call the `WSACleanup()` method.

Useful Links

- `http://beej.us/guide/bgnet/` - A good guide on Sockets. It's mainly focused on Socket programming in Linux but the majority of code samples and functions are the same.
- `http://tangentsoft.net/wskfaq/` - Lots of questions and answers for common problems using the Winsock library. There are also some good complete code samples available.
- `http://msdn.microsoft.com/en-us/library/ms741416(v=VS.85).aspx` - The Windows Sockets API.