

William Taylor | Advanced games programming part 1 | November 14, 2014

Networking Tutorial

By B00235610

# Basic Theory

So what is a network? Well a network is simply an operation where data is sent between computers. Games can use this to great affect by storing game data on a server and then sending it to multiple players or clients so one game state can be shared over multiple machines.

There are 2 types of protocol you will need to learn and these are:

**TCP**

TCP or transmission control protocol is a connection based protocol whereby you establish a connection between some computers and then you write bytes to that connection like a file stream. Most games will use TCP for chat rooms as its safe and reliable way of sending data or packets from one computer to another in a synchronized fashion.

**UDP**

UDP or user datagram protocol is connectionless protocol meaning no connection is maintained between the computers. You send data through an available port and it bounces till it finds its correct destination which is specified through a port number and a hostname. UDP is used most of the time for games online as it’s much faster than TCP but it does have problems but I will leave a link to where you can read more about these protocols.

**Link to a decent article on these two protocols as well as some other things.**

<http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/>

# Implementing networking in my application

In this section I am going to detail how I achieved my pong multiplayer game. The tutorial I will show you however will be a much simpler program which can help you see how to implement a basic network between two computers whereby one computer sends the other a message. This example that I will provide allows plenty of room to maneuver to play around with the tutorial whereas my actual application is huge and cannot be so easily altered. So I will first detail how I got networking in the actual project and then walk you through setting up a simple networking application next.

**Determining the networks structure**

First of all I needed to decide how the network would be structured this was going to be fairly simple as my demo was a game of pong. Player one would be the server and would hold a copy of the games state which would be then shared with player 2 which would be the client.

The relationship would be a little like the diagram below.

Server send updated game state to client

Player 2 – Client receives server state and applies the correct state

Player 1 – Server holds the ball position and player positions

Client sends player 2’s location

While I could implement this with either UDP or TCP with no ill effect due to the fact it’s such a simple game and simple network choosing either would be fine. I decided to use UDP as it’s what the majority of games use and it means I learn how to use the most used protocol in networking at least when it comes to the majority of games.

**Choosing the correct networking library**

There are many networking libraries you can choose from these include:

* SDL\_NET
* Boost:::ASIO
* POCO
* Winsock
* BSD
* DirectPlay

However I decided to use SDL\_NET for a couple of reasons. First it’s a library which is independent of SDL whereas in the others it is a subsystem which can be annoying and the others which aren’t a sub system of an existing library are instead platform dependent. While BSD is none of those it is also a lot more complicated to use than SDLnet which can lead to less manageable code. SDLNet also has good documentation which again makes it very easy to use.

**Creating a reusable class structure**

So obviously the next part comes in, creating the classes which will make constructing network communication easy between player 1 and player 2. I decided to have 3 main classes to make this happen.

* **A client class**
* **A server class**
* **A packet class**

**Client class**

The client class simply has some basic data members you would normally see, a socket, a packet and an IP address to the server we are sending to. The programmer is meant to use this class at the part of the application which is designed to be the client so in my case when player 2 is selected. It allows the user to decide where the packet is going, what port it will arrive on and most importantly the data to be sent.

**Packet class**

The packet class simply controls the SDLNet packet object and deals with converting the data once it has been received by the server or when data needs to be formatted so it can be sent from the client. Both the client and server classes use this object for getting and sending data from the two objects

**Server class**

In comparison to the previous classes the server class is a little more complicated. The server class will open up a port for incoming messages. Once opened it looks for connect and disconnect commands for multiple clients. The server then sends data to all connected clients when the user wishes to send the updated game state. It’s designed so that a single game state can be sent to multiple clients and then handle messages about updates from many clients. In my case player 1 is the server in the game and handles the global game state and player 2 is the one and only client.

# Sending packets and how the string is formatted

So as you probably know a packet is just a set number of bytes that is transmitted from a port. This is understandably done with an array of chars as 1 char = 1 byte. The following is a small section where I go over how I formatted what is essentially a C style string to hold the global game state. Obviously this format changes depending on weather you are the server or the client as the client should be responsible for sending updates whereas the server is responsible for transmitting the entire game state rather than client updates which would be much smaller.

**The server’s packet data format**

Player one would control the server class and send the state in a string which would look like this. Once the packet had been sent it will wait to hear back from player 2 so it can update player 2’s position in the global state. Once again when player 2 or the client has responded it will send back the next update of the game state.

**P1:000,000|P2:000,000|B:000,000**

Player 1 ID X position Y position Player 2 ID X position Y position Ball ID X position Y position

*MAX SIZE OF PACKET 50 BYTES*

**The client’s packet data format**

Player 2 would then read this string once it had arrived and then set the player positions and the ball positions. Once done and the input events had been handled Player 2 will then send a string to Player 1 or the server which would look like this.

**P2:000,000**

Player 2 ID X position Y position

*MAX SIZE OF PACKET 18 BYTES*

**Creating input on screen to get required data**

As a final note obviously I didn’t have time to do matchmaking for multiple users looking for a game. So instead I decided to let the user break the application if they wish. They can select the computer they want to play against by inputting their opponent’s computers hostname which is displayed on the multiplayer screen in case the user doesn’t know it. I however didn’t allow them to choose the port which would have been a major improvement.

# The Tutorial

So as I said in the previous part this is where I will cover the tutorial code that I have provided which will actually teach you how to set up networking in your application using SDLNet a simple networking library. Instead of giving an in depth review of the code in this document I have decided to comment each line of code which details how the application works. The only thing I have to cover is how to set the project and how to run it.

**Setting up the project**

So to setup the code you will need to open the project in ***Visual Studio 2013***. The file you should try to understand is the ***lab\_main.cpp*** which contains the code for the networking. You will notice that this project also contains ***lab\_main.h*** which contains a list of simple functions to help remove the clutter so you can focus on the networking code.

**Running the project**

I have bundled in the executable just in case you can’t compile the project for some really weird reason. Once it starts you will see a message box come up asking which program you want to run the client or the server. If you select server a blank console will come up which will show the last message send by the client. If you start up the client you will again receive a console which will ask for input. The moment you enter a word it will be sent to the server to be displayed.

Something to note the title bar states what version of the application you a running just in case you get mixed up. Also to quit both applications just enter “quit” as the message and both the client and the server will shut down.

# Additional Exercises

Here are some additional exercises for you to try which should be easy to implemented providing you do it by extending the lab provided.

**Key Logger**

Using SDL2 which is already in the solution, try sending key state data to the server program and get it to display an active record off all keys pressed by the client.

**2 way network**

Try and get the server to send back a response to the client without creating another server like object on the client side of the program.

# Cross platform play

So my full part one project was actually pong which not only had basic multiplayer but was also cross platform and allowed one user to play the game on Linux and the other to play it on Windows. While the tutorial does not cover this area of the demo due to the fact it would be ridiculously long, I would like to make some quick notes for you so you know where to look when porting your game to other platforms.

**Cross platform libraries**

The first tip is to avoid using native API calls wherever possible, Windows is particularly bad for this. This is something you need to take note the moment you start your project because the more you use platform independent code the less portable it becomes. So here is my advice look for open source libraries which compiles across multiple platforms. Never go to the standard Windows or Linux API’s unless it’s absolutely necessary. Use frameworks such as SDL2 and SMFL which are very popular and have seen great enhancements and are actually used to port applications to other systems.

**Graphics & Audio**

If you are using DirectX abandon it and join the OpenGL community as you will not be able to be cross platform without it. Furthermore OpenGL has embedded versions of its which can even make you cross platform on mobile devices as well thanks to the invention of OpenGL ES. Likewise this also applies to Audio API’s so libraries like DirectSound or DirectShow these should be scrapped as well and replaced with libraries such as IrrKlang and BASS. I didn’t use OpenGL rendering for my project as SDL2 has a nice simple renderer for rendering simple 2D games which is very effective I would advise if you are making a simple 2D game you should use the built in renderer as well.

**Unix/Linux filenames**

One quick note here. When you are on Windows “Text.txt” and “TEXT.tXt” will be the same on Linux they will not be. Filenames in Linux will be case sensitive so be sure to take note about this. Make sure you provide effective exception handling when loading resources from disk as it will help you identify the files that are causing the problems when you port your game. Alternatively make filenames lowercase and make sure that the string you provide as the location is indeed correct.

**Best distribution for newbies**

While there are many distributions on Linux I would recommend you go with Linux Mint the most recent version as it’s the one I got setup in with very little effort and due to its popularity there are lots of forms where you can ask questions should you get stuck.