# Methodology

## Overview of the system

### What

I wanted the game engine to communicate to an interface, which was an external application, and for that interface to communicate to another application.

### How

The method I chose to implement was the client server architecture. Since the unity is built in mono, which itself uses .net, I could easily use the functionality built in. There are other architectures that I could have used but I believe that this would provide the widest range of functionality. The extra functionality comes from the ability for different languages to communicate with the servers. For example a client written in Java can connect to a C# server.

I originally intended to use the windows communication foundation to communicate between applications.

The server holds a number of methods/functions that the client can access. When the client wants to run the function all it has to do is call the method. The connection will communicate to the server and the server will return data.

Of course the user will have to host the server on a machine.

Mono is a version of .net that can be used on multiple platforms; .net is only useable on the windows operating system. While mono is based on .Net, it doesn’t implement everything that .Net has.

Another feature that should be implemented would be a way for the user to run the server. The developer aimed to create an executable file that would be run, which would run the server until the user chose to close it down.

### What I really did

While I originally aimed to use WCF tool to create the server I ran into problems implementing it. Due to the version of .Net that mono is built on and the Unity game engine I found it extremely difficult to develop. Therefore I opted to use ASP.Net. WCF is the newer version of ASMX, aimed to have all the features and more.

The difficulties arouse from hosting the server and generating the appropriate files from that server. When hosting the server, the developer can run a command that will generate code for the client. When trying this, the developer ran into issues with connecting to the server. The command would not generate any code but rather an empty file, as it could not connect.

The user has to host the asmx server on a machine in order for the game engine to communicate with this. As stated above, the user would have to turn on the server before the interface would work. Originally the developer wanted to create an executable file that the user could run. But this proved to be difficult to implement, therefore the user has to manually run it from command line.

Setting up a second server has proven difficult to achieve as of yet. Therefore all external applications are built into the interface as of just now. This will be fixed soon.

## Prototype 1

### Design

Prototype One will contain a bot that will use a simple wander behaviour. The bot will move about the environment randomly. There is no goal for the bot, it just moves completely randomly around the environment. This behaviour will be built into the game engine. There will be no external applications at this point. This is the way developers in the industry do it.

### Implementation

Wander behaviour is a simple steering behaviour. The bot will constantly move forward at a constant velocity. A random number between certain ranges will be selected. This will represent how much the bot should rotate. By restricting the amount the bot can rotate, a more believable behaviour can be obtained.

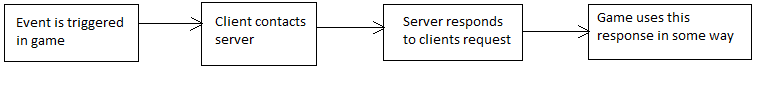
Developing this was straightforward. The bot generates a random number between -10 and 10. This serves are the amount of degrees for the bot to rotate by. This range of numbers was selected as it allowed the bot move smoother than if it could 360º. The bot moves forward at a constant speed. By combining this and the rotation, it allows the bot to move around the environment.

## Prototype 2

### Design

The goal for Prototype Two is to test if it is possible to connect the game running in unity to the interface. This will be driven by the player/user.

When the game is running it must be able to communicate with the interface. Therefore the client will be the game and the server will be the interface. The client must contact the server and receive data back. This is driven by the player, this is due to fears of overloading the network if it is done automatically.



### Implementation

The developer wrote a simple function that acts as an accumulator on the server. This will serve as the message between the client and the server. The client will send a message to the server, the server will then reply with the accumulated value. The client being the game, the server being the interface. This accumulated value will act as the amount of times that the function has been called by the client.

The reason it keeps an accumulated value rather than a piece of text was because the developer wanted to check how variables are stored on the server. The returned value is printed off in the console of the game and the value is printed off in the server’s console. This is to clarify that both have the same value.

To trigger the function call to the server the player has to click the button on screen.

## Prototype 3

### Design

This prototype is a combination of the two previous prototypes.

The game contains a bot that will wander about the environment, but the interface will be generating the random values for the bots rotation.

At every update of the game the game will call the server and ask for a random number between a certain range. This is similar to prototype one but it uses the interface rather than being built into the game.

The purpose of this prototype is to test the server. The server needs to be able to handle requests from the client at a high frequency. The game engine runs at 30 frames per second, therefore the server needs to be able to handle this.

### Implementation

The server contains a function that will return a value between the given parameters. This is called and the returned value acts as the amount of degrees for the bot to rotate by. This function is called every update by the game engine.

## Prototype 4

### Design

This prototype will feature an external application that the server will use to obtain data from. In this prototypes case it will be a neural network that the server will connect to.

At every frame of the game the game will contact the interface, which will in turn contact the neural network. The game will contain a simple bot that can see objects at three points around itself. The game passes information about what the bot see’s into the interface. The interface then passes on the data to the neural network. It is up to the neural network to decide what action should be done. It is this action that is passed back into the game where it is carried out.

But having the neural network connect isn’t enough. It has to be able to learn. Therefore it must contain a method for learning, in this case is by using a genetic algorithm.

By using a genetic algorithm, it allows the neural network to evolve and learn. This will allow the bot to learn how to act in the environment.

The neural network will be of the simple feed forward mutli-layered architecture. The data from the game will act as the input to the neural network. This will be the details about what the bot can see. This gets passed through the neural network and the output will be the action that should be undertaken. In this case the output will control how much the bot should rotate and the speed at what they move forward at.

During testing, the game will run for a predetermined amount of time. During this time the bot runs the neural network and its fitness is evaluated. After this time is up, the next neural network is tested. This continues until all the networks have been evaluated. Once this is complete the genetic algorithm will create children of the current generation to test. This keeps on happening until the user stops this process.

### Implementation

The key part of this prototype is setting up a delay. Therefore no updates will call until a certain amount of time has passed. If it tries to run when everything is not setup or if it tries to run during it is generating a new population, serious problems could occur. Therefore after certain functions are called there will be a delay until it can do anything.

One thing that became apparent it was needed was wiping the server after each run. As long as the server was active it contained data about all previous runs. Such as the genetic algorithms population would add onto previous runs populations. Therefore the developer wrote a function that wipes the server of all previous data, this function is called at the start of every run.

The genetic algorithm must have a fitness function in order for learning to occur. As stated in the design that the fitness will be dependent on following the player. Therefore the fitness will be if the middle sensor detects the player then the fitness will increase.

Connecting multiple servers together proved to be a problem. Therefore currently the neural network is built into the interface. This is just a temporary, once the developer finds and fixes the bug that is stopping this, it will be moved to a separate server.

## Final Product

### Design

This will be the final product.

It will feature two levels.

One showing off a cooperative bot helping the player

Another will show the competitive side of the bot, going up against the player.

### Implementation

## Things to test

Stress test the network

That learning can be achieved

That the interface is well documented.

Different computer on lan.