**Specification of Bouncing Balls**

**Written by Young Ho Son**

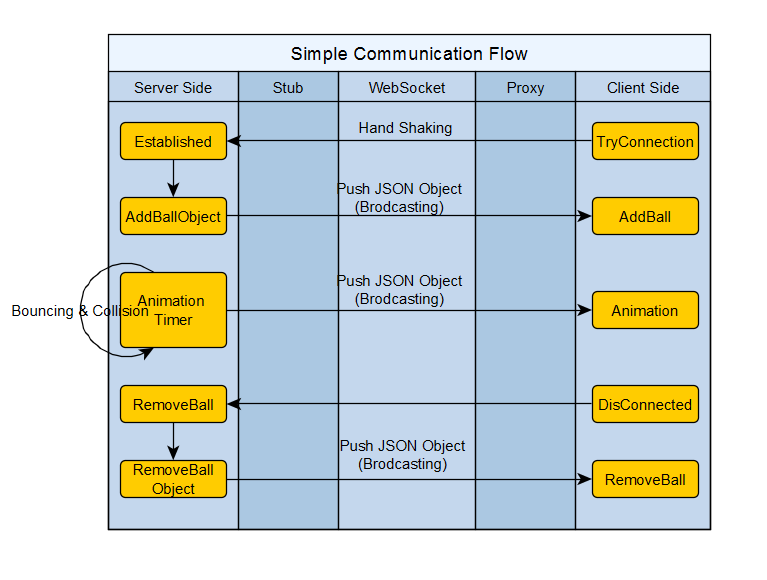
**Scope of Sample Project by myself**

* Each browser client that connects adds one ball to the simulator. All browser clients can see all balls in the simulator.
* The balls bounce off of each other, and around a viewport in the window.
* The physics logic should happen in the server-side C# code, and this logic should be authoritative (clients should not be able affect other clients by altering their JavaScript).
* The code should be documented and unit tested.
* Your web-client code only needs to support modern browsers (Chrome / Firefox).
* The physics do not have to be realistic with no gravity.

**Frameworks**

* .NET Framework 4.5
* ASP.NET SignalR 2.2
* JQuery 1.12, CSS
* XUnit, Moq
* JSON
* Visual Studio 2015 : Nuget Package Manager will install above components automatically.

**Communication Flow**



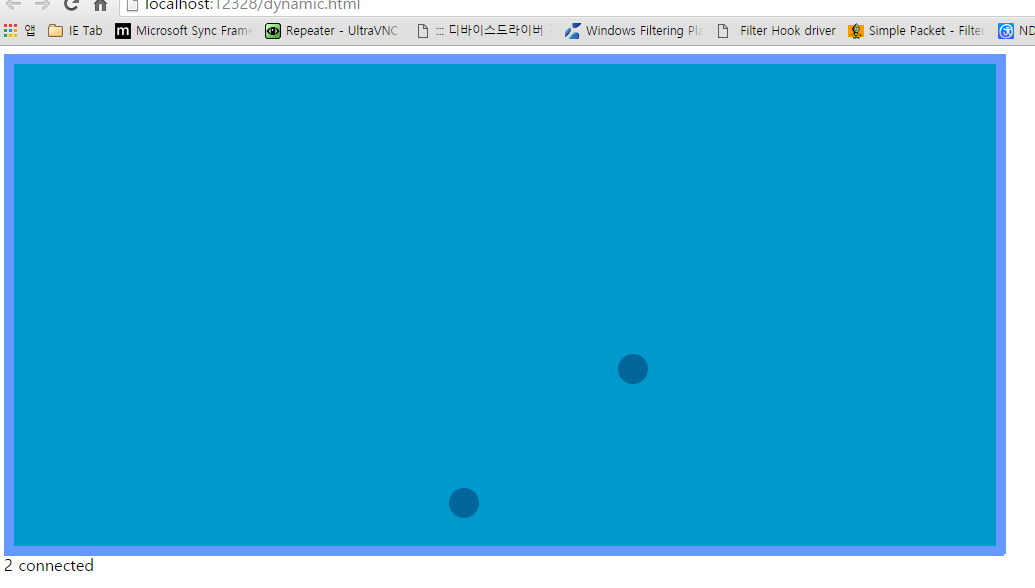
**Limitation**

* The bouncing behavior is implemented based on trigonometry. Sometimes there is wrong calculated values and not perfect for this time.
* The unit test is insufficient for this time.
* The communication flow and protocol are not optimized for this time.

**Fundamental**

SignalR is a new Microsoft library that lets the server-side code push content to its connected clients.

My Bouncing Balls that I made, for example, would need to update the new position of balls on all connected clients as it moves and bounces around and bounces off each other.

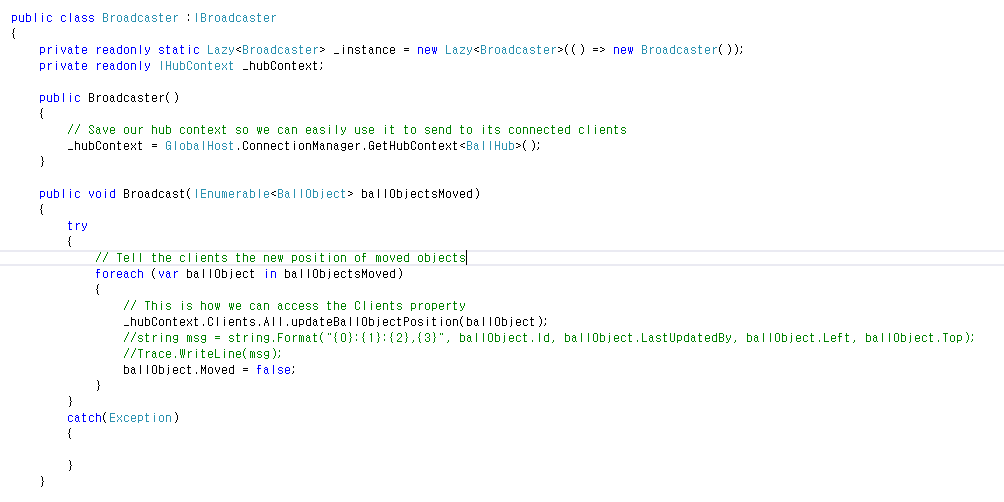


To communicate, SignalR uses the new WebSocket transport where available, and falls back to older transports where necessary. A SignalR connection starts as HTTP, and is then promoted to a WebSocket connection if it is available. WebSocket is the ideal transport for SignalR, since it makes the most efficient use of server memory, has the lowest latency, and has the most underlying features (such as full duplex communic ation between client and server), but it also has the most stringent requirements: WebSocket requires the server to be using Windows Server 2012 or Windows 8, and .NET Framework 4.5. If these requirements are not met, SignalR will attempt to use other transports to make its connections .

Let’s look at the server code first. On the server, I implemented a Hub, which is the medium that the client JavaScript uses to signal the server:



This hub contains the methods that the clients will call directly. Hub instances are transient; SignalR creates a new instance of your Hub class each time it needs to handle a Hub operation such as when a client connects, disconnects, or makes a method call to the server. This is why I needed to make my ServerController a singleton, so the same one is used each time. Since multiple threads will be running the same instance of ServerController code, the ServerController class has to be thread-safe. Lazy initialization is used for the \_instance field, not for performance reasons but to ensure that the instance creation is thread-safe. Once a client is connected, it may call any public method in the hub, the parameters passed and values returned are serialized using JSON. If you need to write initialization code when a client first connects to your hub, you can override the OnConnected, OnDisconnected and OnReconnected methods in your hub. Let’s look at my Broadcast method that sends the new position of the moved objects to the connected clients:



Using the IHubContext, I can invoke functions that I wrote in my client JavaScript. I can refer to updateBallObjectPosition here (which is a function in my client JavaScript) because Clients.All is dynamic, which means the expression will be evaluated at runtime. When this method call executes, SignalR will send the method name and the parameter value to the client, and if the client has a method named updateBallObjectPosition, that method will be called and the parameter value will be passed to it. Return types are always returned using JSON. If no matching method is found on the client, no error is raised, so be careful! The \_hubContext is initialized only once like this:



There are two reasons why you want to get the context just once: getting the context is an expensive operation, and getting it once ensures that the intended order of messages sent to clients is preserved. If you are signaling your client from within your hub class, you can directly reference Clients from there instead. If you need to, making methods in your hub asynchronous is easy. Just make them return Task (or Task<T>), and mark the method with the async keyword. The client JavaScript code is the same for both synchronous/asynchronous method invocations to the server. You should make all methods asynchronous if they are long-running or involve waiting (like a database lookup or a web service call), otherwise you can block the connection when it uses the WebSocket transport. When a hub method executes synchronously and the transport is WebSocket, subsequent invocations of methods on the hub from the same client are blocked until the hub method completes.

Let’s look at the client JavaScript code now. First, I get a reference to the proxy for the BallHub class and put it in a variable. $.connection refers to the SignalR proxies. The proxy name is the name that was set by the [HubName] attribute, or the name of the class by default:



Now I can write a function that can be called from the server:

