Writing a WebSocket server in C#

This article is in need of a technical review.

Introduction

If you would like to use the WebSocket API, it is useful if you have a server. In this article I will show you how to write one in C#. You can do it in any server-side language, but to keep things simple and more understandable, I chose Microsoft's language.

This server conforms to RFC 6455 so it will only handle connections from Chrome version 16, Firefox 11, IE 10 and over.

First steps

WebSocket's communicate over a TCP (Transmission Control Protocol) connection, luckily C# has a TcpListener class which does as the name suggests. It is in the *System.Net.Sockets* namespace.

It is a good idea to use the `using` keyword to write less. It means you do not have to retype the namespace if you use classes from it.

TcpListener

Constructor:

```
1 TcpListener(System.Net.IPAddress localaddr, int port)
```

You set here, where the server will be reachable.

To easily give the expected type to the first parameter, use the `Parse` static method of `IPAddress`.

Methods:

- Start()
- System.Net.Sockets.TcpClient AcceptTcpClient()
  Waits for a Tcp connection, accepts it and returns it as a TcpClient object.

Here's how to use what we have learnt:

```
using System.Net.Sockets;
using System.Net;
using System;

class Server {
    public static void Main() {
        TcpListener server = new TcpListener(IPAddress.Parse("127.0.0.1"), 80);

        server.Start();
        Console.WriteLine("Server has started on 127.0.0.1:80.{0}Waiting for a con

        TcpClient client = server.AcceptTcpClient();

        Console.WriteLine("A client connected.");
    }
}
```

TcpClient

Methods:

- `System.Net.Sockets.NetworkStream GetStream()`
  Gets the stream which is the communication channel. Both sides of the channel have reading and writing capability.

Properties:

- `int Available`
  This is the Number of bytes of data that has been sent. the Value is zero until *NetworkStream.DataAvailable* is *false*.

NetworkStream

Methods:

```
Write(Byte[] buffer, int offset, int size)
```

Writes bytes from buffer, offset and size determine length of message.

```
Read(Byte[] buffer, int offset, int size)
```

Reads bytes to *buffer, offset and size determine the length of the message*

Let us extend our example.

```
TcpClient client = server.AcceptTcpClient();

Console.WriteLine("A client connected.");

NetworkStream stream = client.GetStream();

//enter to an infinite cycle to be able to handle every change in stream
while (true) {
    while (!stream.DataAvailable);

    Byte[] bytes = new Byte[client.Available];

    stream.Read(bytes, 0, bytes.Length);
}
```

Handshaking

When a client connects to a server, it sends a GET request to upgrade the connection to a WebSocket from a simple HTTP request. This is known as handshaking.

> *This code has a bug. Let's say `client.Available` returns 2 because only the GE is available so far. The regex would fail even though the received data is perfectly valid.*

```
using System.Text;
using System.Text.RegularExpressions;


```

```
Byte[] bytes = new Byte[client.Available];

stream.Read(bytes, 0, bytes.Length);

//translate bytes of request to string
String data = Encoding.UTF8.GetString(bytes);

if (new Regex("^GET").IsMatch(data)) {

} else {

}
```

Creating the response is easier than understanding why you must do it this way.

You must,

1. Obtain the value of *Sec-WebSocket-Key* request header without any leading and trailing whitespace
2. Concatenate it with "258EAFA5-E914-47DA-95CA-C5AB0DC85B11"
3. Compute SHA-1 and Base64 code of it
4. Write it back as value of *Sec-WebSocket-Accept* response header as part of a HTTP response.

```
if (new Regex("^GET").IsMatch(data)) {
    Byte[] response = Encoding.UTF8.GetBytes("HTTP/1.1 101 Switching Protocols" +
        + "Connection: Upgrade" + Environment.NewLine
        + "Upgrade: websocket" + Environment.NewLine
        + "Sec-WebSocket-Accept: " + Convert.ToBase64String (
            SHA1.Create().ComputeHash (
                Encoding.UTF8.GetBytes (
                    new Regex("Sec-WebSocket-Key: (.*)").Match(data).Groups[1].Val
                )
            )
        ) + Environment.NewLine
        + Environment.NewLine);

    stream.Write(response, 0, response.Length);
}
```

Decoding messages

After a successful handshake client can send messages to the server, but now these are encoded.

If we send "MDN", we get these bytes:

129	131	61	84	35	6	112	16	109

- 129:

FIN (Is this the whole message?)	RSV1	RSV2	RSV3	Opcode
1	0	0	0	0x1=0001

FIN: You can send your message in frames, but now keep things simple.
Opcode *0x1* means this is a text. Full list of Opcodes

- 131:

If the second byte minus 128 is between 0 and 125, this is the length of message. If it is 126, the following 2 bytes (16-bit unsigned integer), if 127, the following 8 bytes (64-bit unsigned integer) are the length.

> *I can take 128, because the first bit is always 1.*

- 61, 84, 35 and 6 are the bytes of key to decode. Changes every time.

- The remaining encoded bytes are the message.

Decoding algorithm

decoded byte = encoded byte XOR (position of encoded byte Mod 4)th byte of key

Example in C#:

```
1 Byte[] decoded = new Byte[3];
2 Byte[] encoded = new Byte[3] {112, 16, 109};
3 Byte[] key = Byte[4] {61, 84, 35, 6};
4
5 for (int i = 0; i < encoded.Length; i++) {
6     decoded[i] = (Byte)(encoded[i] ^ key[i % 4]);
7 }
```

Related

- Writing WebSocket servers