

10/21/2012

SPANG

API WALKTHROUGH

Version 1.0 | Spang

Table of Contents

Low Level.....	2
Opening a Connection.....	2
Using the Connection	2
Protocols	2
Example usage	3
Sending Data	3
Receiving data	4
Complete Code Usage example	4
High level.....	5
IClient vs IServer	5
Asynchronous connections.....	5
Event based	6
Sending Messages.....	6
Receiving messages.....	7
Handling disconnections	7
Complete Code Usage Example	7

Low Level

Opening a Connection

The low level API makes use of the standard implementations for *IConnection*, *IConnector* and *IConnectionListener*. To start sending messages between two remote devices the first thing that has to be done is to open a connection between the two.

One side of the connection listens for incoming connection request... (At the time of this writing only implemented in c#)

```
//Listens to incoming connections and return a connection object when a
client connects.
IConnectionListener.ReciveConnection(port : int) : IConnection.
```

...while the other side of the connection connects to the listening side.

```
//Tries to connect to the remote host and returns a connection if successful.
IConnector.Connect(host: string, port : int) : IConnection
```

Using the Connection

Now that we have a connection, we can use it to send and recive messages.

Protocols

When data is being send the user must decide what protocol to use for sending.

All protocols are extensions of the UDP-Protocol. A more in depth look at protocols can be found in the Protocols.pdf file.

There are four different low level protocols:



It's important to note that an IConnection can send diffrent messages using diffrent protocols. So there is no need to open a new connection for each protocol that can be used.

Example usage

Let's say we have this awesome networked multiplayer game. Several times per second all the players send positional information to the server. This is higly mutable information that is likely to change every frame. It would make sence to use the Ordered protocol here since if we miss a message or two that is no big deal. And we want to discard out of date messages since old positional information is irrelevant.

To contrast this, the player also has the ability to shoot. This is a message that only happens on rare occasions and the other players would like know when this happens. So here it would make sense to use the Reliable protocol.

Geometry is procedurally animated using a several messages long noisetexture. This is generated on the server and the server sends it to all its clients. Here we would be best to use Reliable Ordered since this is a message that must be in order and that must make it to the end destination.

Sending Data

```
//Sends data using user defined default protocol.  
IConnection.send(data:byte[]) : void  
  
//Sends data using provided protocol.  
IConnection.send(data:byte[], protocol:Protocol) : void
```

Receiving data

```
//Protocols are managed transparently here so the receiving end doesn't need  
//to worry.  
IConnection.Receive() : byte[]
```

Complete Code Usage example

For simplicity the code below does not do any exception handling.

C# Code:

```
class MyServerProgram  
{  
    public static void main(String... args) throws Exception {  
  
        IConnectionListener listener = new ConnectionListener(); //Default  
        implementation.  
  
        //Listens until a connection is requested on port 31337. (This call is  
        blocking)  
        IConnection connection = listener.ReceiveConnection(31337);  
  
        //Sends "Hello" in UTF-8 format.  
        connection.Send(Encoding.UTF8.GetBytes("Hello"));  
  
        //Recives a response from the client.  
        byte[] recievedResponse = connection.Receive();  
  
        //Outputs response.  
        Console.WriteLine(Encoding.UTF8.GetString(recievedResponse),  
        Protocol.Reliable);  
    }  
}
```

Java Code:

```

class MyClientProgram
{
    public static void main(String... args) throws Exception {
        IConnector connector = new Connector(); //Default implementation.
        //Tries to connect to the server host "somehost" on the port 31337
        IConnection connection = connector.Connect(new
        InetSocketAddress(InetAddress("somehost", 31337)));

        //Receives a message from the server-
        byte[] receivedMessage = connection.receive();

        System.out.println(new String(receivedMessage , "UTF-8"));

        //Sends a "Nice to meet ya" message.
        connection.send("Nice to meet ya".getBytes("UTF-8"),
        Protocol.Reliable);

        //Terminates the connection.
        connection.close();
    }
}

```

High level

The high level API makes use of implementations of *IServer*, *IClient*, *IConnector*, *IConnectionListener* as well as classes that serialize data. (Implementations of *Serializer<T>* and *SerializeManager*)

At this time of writing *IServer* and its implementation exists only in the c# codebase.

IClient vs IServer

IServer is capable having multiple connections at the same time. A server can never initiate a connection but is instead always listening for incoming connections.

IClient can only have one connection at any given time. It cannot receive incoming connections, but is instead the connecting party. To establish even the most basic connection; one side needs an *IServer* while the other side needs an *IClient*.

When a connection is established, both the *IServer* and *IClient* can send messages and receive messages from each other.

Asynchronous connections

The main difference between *IServer/IClient* and *IConnection* is that *IConnection* handles incoming connection requests and incoming messages in the background by background threads. Or rather, *IServer/IClient* creates background threads using an *IConnection*.

This means that the main thread can do more things than simply being blocked by connection requests and network messages.

Event based

IServer/IClient are event based, so when important events happen such as connection, disconnection and messages are received, listeners to those events will be notified.

It is important to note that events do not happen on the thread that created the connections, but in various background threads. So code that using events from *IServer/IClient* has to be thread safe.

Creation:

```
IServer server = new Server(connectionListener : IConnectionListener,  
messageSerializer : SerializerManager);
```

The above code creates a server. The *connectionListener* is the listener that will be used to receive incoming connections. The *messageSerializer* is the serializer that will be used to serialize outgoing messages and deserialize incoming messages.

```
IClient client = new Client(connector : IConnector, messageSerializer :  
SerializerManager);
```

The above code creates a client. The *connector* is the connector that will be used when the client attempts to connect to a server. The *messageSerializer* is still the serializer that will be used to serialize outgoing messages and deserialize incoming messages.

Connecting:

```
IServer.Start(port : int);
```

The above code starts the server. It is now listening on the specified port. Now the server can handle connection requests from clients. When a new connection is received, a connection event will be triggered.

```
IClient.Connect(host : String, port : int);
```

The client attempts to connect to a server running on the specified host and port. If the connection is successful, a connection event will be raised on both the client and the server.

Sending Messages

In contrast to *IConnection*, where the send method can only send messages in the form of byte arrays, the *IServer/IClient* can send any type of message that has an associated *Serialize<T>* implementation that is registered with the *SerializeManager* in use (for more detailed explanation see *Serialization.pdf*).

```
IClient.send(message : Object, protocol : Protocol);
```

The *send* method serializes the message and sends it over the network using the given protocol.

```
IServer.send(connectionID : int, message : Object, protocol : Protocol);
```

The *send* method serializes and sends the message to the connection specified by the connection id, using the given protocol.

```
IServer.sendToAll(message : Object, protocol : Protocol);
```

The *sendToAll* method serializes and sends the message to all active connections using the given protocol.

```
IServer.sendToAllExcept(id : int, message : Object, protocol : Protocol);
```

The *sendToAllExcept* method serializes and sends the message to all active connections except the specified id using the given protocol.

Receiving messages

The receiving of messages is asynchronous, so in order to know when a message is received simply register to the received message event.

The *IServer/IClient* deserializes the incoming message and the message result is passed back an object instance that is sent with the event. (The *Serialization.pdf* explains this in more detail)

Handling disconnections

When a disconnection occurs, a disconnect event is triggered containing information about why the event occurred. For instance, the cause could be a *RemoveConnectionCrash* or a *ConnectionTimeout*.

Complete Code Usage Example

C# Code Server Side: Very Simple Chat server


```

public class MyChatServer
{
    public static void Main(string[] args)
    {
        //Create a serializerManger that can serialize objects.
        SerializeManager manager = new SerializeManager();
        //Make it capable of serializing strings.
        manager.AddSerializer(new StringSerializer());

        //Create a server.
        IServer server = new Server(new ConnectionListener(), manager);

        //Start the server on the port 1337.
        server.Start(1337);

        //Hook up a connected lambda.
        server.Connected += (s,e) =>
        {
            //Output that a connection was received.
            Console.WriteLine("A connection was received! ID : {0}", e.ConnectionID);
            //Notify all clients that a new player has entered the chat.
            s.SendToAllExcept(e.ConnectionID,
                string.Format("A new member with ID: {0} has connected", e.ConnectionID),
                Protocol.Reliable);
        };

        //Hook up a received lambda.
        server.Received += (s,e) =>
        {
            //Check if the message is a string.
            if(e.Message is string)
            {
                Console.WriteLine(e.Message);
                string message = e.Message;
                //Create a message to send to all chatmembers except the sender.
                message = e.ConnectionID + " says: " + e.Message;
                //Send the message to all chatmembers except the one that sent the message.
                s.SendToAllExcept(e.ConnectionID, e.Message, Protocol.Reliable);
            }
        };

        //Hook up a disconnected lambda.
        server.Disconnected += (s,e) =>
        {
            Console.WriteLine("A connection disconnected! ID : {0}", e.ConnectionID);
            //Notify all clients that a new player has entered the chat.
            s.SendToAllExcept(e.ConnectionID,
                string.Format("A new member with ID: {0} has disconnected",
e.ConnectionID), Protocol.Reliable);
        };

        while(true)
        {
            //This server does nothing except deal with the network. If it was more
            //Complicated the main thread would do something here instead of simply sleeping.
            Thread.Sleep(100000);
        }
    }
}

```

Java Code: Very simple chat client.

```
public class MyChatClient
{
    public static void main(String[] args) throws Exception
    {
        //Create a serializerManger that can serialize objects.
        SerializeManager manager = new SerializeManager();
        //Make it capable of serializing strings.
        manager.addSerializer(new StringSerializer());

        //Creates the client with the default Connector.
        IClient client = new Client(new Connector(), manager);

        //Hook up a listener to the connection event.
        client.addConnectedListener(new EventHandler<IClient, Boolean>() {
            public void onAction(IClient sender, Boolean eventArgs) {
                System.out.println("We connected to the server!");
            }
        });

        //Hook up a listener to the recived event.
        client.addRecivedListener(new EventHandler<IClient, Object>() {
            public void onAction(IClient sender, Object message) {
                //Print the message if it's a string.
                if(message instanceof String) {
                    System.out.println((String)message);
                }
            }
        });

        //Hook up a listener for the disconnected event.
        client.addDisconnectedListener(new EventHandler<IClient, DCCause>() {
            public void onAction(IClient sender, DCCause cause) {
                System.out.println("We disconnected! Cause : " + cause);
                System.exit(-1);
            }
        });

        //Connect to the server specified
        client.connect("MyServerHostIPorName", 1337...
    }
}
```