10/21/2012

SPANG API WALKTHROUGH

Version 1.0 | Spang

Table of Contents

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

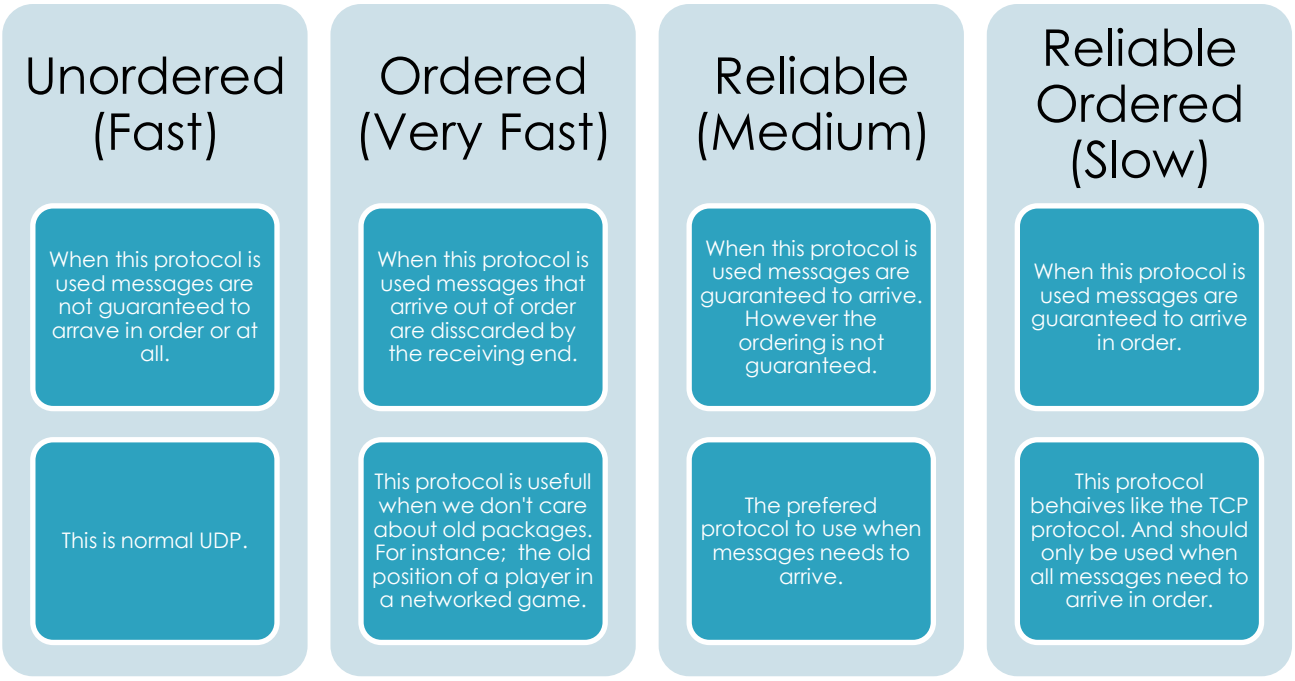
Now that we have a connection, we can use it to send and recive messages.

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 And we want to discard out of date messages since old positional information is irrelevant.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.recive() : 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.ReciveConnection(31337);

    //Sends "Hello" in UTF-8 format.
    connection.Send(Encoding.UTF8.GetBytes("Hello"));

    //Recives a responce from the client.
    byte[] recievedResponse = connection.Recive();

    //Outputs responce.
   Console.WriteLine(Encoding.UTF8.GetString(recivedResponce),
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.recive();

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

  //Hook up a recived labada.
  server.Recived += (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.SendToAllExept(e.ConnectionID, e.Message, Protocol.Reliable);
   }
  }
  //Hook up a disconnected lambada.
  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.SendToAllExept(e.ConnectionID,
        string.Format("A new member with ID: {0} has dissconnected",
e.ConnectionID), Protocol.Reliable);
  };
  while(true)
  {
   //This server does nothing exept 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);

          //Hock 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...
```

Serialization

The important types in the serialization API are *SerializeManager*, *ISerializer* and *Serializer<T>*. The serialization framework is responsible for converting Java and c# Objects to byte arrays and back again. It is not responsible for what is done with the results though.

Currently the high level Network API uses the result from the serialization and sends it over the network.

It is then deserialized to the appropriate object at the remote end.

ISerializer and Serializer<T>

Resonsibilities:

Serializing and Deserializing a single type of object such as a String or a custom object CoolMessage.

The *ISerialize* and *Serializer<T>* are never used directly instead subclasses of these do the actual serialization and deserialization. An example of this is *StringSerializer* that can serialize and deserialize Strings.

Example:

Serialization and Deserialization of a CoolMessage class.

```
//A class that contains some data.
class CoolMessage {
     public int SomeIntVar;
     public float SomeFloatVar;
}

//The serializing class.
class CoolMessageSerializer : Serializer<CoolMessage> {
     public CoolMessageSerializer() {
          super(CoolMessage.class);
     }
     @Override
     protected void serializeInternal(Packer packer, CoolMessage message)
{
          packer.packInt(message.SomeIntVar);
          packer.packFloat(message.SomeFloatVar);
     }
     @Override
     public CoolMessage deserialize(UnPacker unpacker) {
          CoolMessage message = new CoolMessage();
          message.SomeIntVar = unpacker.unpackInt();
          message.SomeFloatVar = unpacker.unpackFloat();
     }
}
```

As can be seen in the example, a Packer object is used to serialize the message and an *UnPacker* object is used to deserialize it. For each method in Packer that packs a primitive, a corresponding Method can be found in UnPacker that unpacks that primitive.

It is important that the order that bytes get packed using the packer is identical to the order in which they are unpacked. If this is not the case the behavior is

Undefined. An exception can be thrown but it is not certain that it will. To avoid this, the *serializeInternal* and deserialize methods should be unittested for each new serializer.

SerializeManager:

Responsibilities:

Manages mapping between Types and *ISerializer* able to deserialize that type. It also maps each type to a unique id that is used as an identifier when a byte array gets deserialized.

At the moment of creation SerializeManager is quite useless it cannot serialize or deserialize anything. For it to be able to do this serializers needs to be added to it using a registation method.

```
SerializeManager.registerSerializer(serializer : ISerializer);
```

After a serializer is registered in this way the *SerializeManager* can use it to serialize and deserialize objects that the *ISerializer* can serialize.

Ex:

```
SerializeManger manager = new SerializeManger();
manager.regiserSerializer(new StringSerializer());
```

Now the *serializeManager* is capable of serializing Strings.

(NOTE: The order in which *ISerializers* are added is important since this impact what unique ID's will be generated for any specific serializer. The SerializeManager uses this unique id to determine what *ISerializer* will be used, so if serializes are registered in different order on different *SerializeManagers* they will not be able to understand each other and serialization will be pointless)

```
SerializationManager.serialize(Packer packer, Object toSerialize);
```

Serializes the object provided. The result of the serialization can be found in the packer provided. (It is required that an *ISerialize* that can serialize the object has been registered.)

```
SerializationManager.deserialize(UnPacker unPacker) : Object
```

Deserializes a byte array contained in *unPacker* into an appropriate object.

Sending sensor-data

The spang.android.sensors package on the Android side includes all infrastructure needed for collecting and encoding data from the sensors of an Android device to the computer. It uses a number of classes and interfaces to do this, and here we'll walk through their responsibilities.

The class *SensorListBuilder* is used to create a list of *ISensors* containing all the sensors available in the device. The way this is done is by checking if the hardware sensors exist through null-checks. This makes for some ugly code, but unfortunately the Android framework doesn't supply a better option. The ID's for the sensors are fetched from the *SensorMappings* enum, and are used to make sure that all sensors are handled correctly on the server side.

The *ISensor* interface is implemented by the class *SpangSensor*, and includes methods needed for reading, encoding and updating sensor data. The sending of sensor data is implemented using an Android service; *SpangSensorService*. It looks for activated sensors, and sends their values over the network with an interval specified by each sensor's sampling-rate.

If you want to send sensor-data in your application, you need to bind to a *SpangSensorService* and activate the sensors you want to use with the start() method.

When the sensor-data is received by the server, the Phone class interprets it in the OnSensor method.