|  |
| --- |
| **Cloudy Message Passing Library** |
| Documentation |
|  |
| The Cloudy Message Passing Library is a .NET library for development of scalable parallel applications. |
|  |
| **Pavel Perestoronin** |
| **10/21/2011** |
|  |

[1 Overview 3](#_Toc307436101)

[2 Components 3](#_Toc307436102)

[2.1 Protocol Buffers – the Protobuf namespace 3](#_Toc307436103)

[2.1.1 Getting Started 3](#_Toc307436104)

[2.1.2 Optional and Required Fields 3](#_Toc307436105)

[2.1.3 Repeated Fields 3](#_Toc307436106)

[2.1.4 Packed Repeated Fields 4](#_Toc307436107)

[2.1.5 Types Mapping 4](#_Toc307436108)

[2.2 Messaging Utility Classes – the Messaging namespace 5](#_Toc307436109)

[2.2.1 MessageStream 5](#_Toc307436110)

[2.3 Helper Classes – the Helpers namespace 5](#_Toc307436111)

[2.3.1 UdpStream 5](#_Toc307436112)

# Overview

The library consists of the following separate parts interacting with one another:

* [Protobuf](#_Protocol_Buffers) namespace ([Protocol Buffers](http://code.google.com/p/protobuf/) implementation)
* [Messaging](#_Messaging_Utility_Classes) namespace
* [Helper classes](#_Helper_Classes_–)

# Components

## Protocol Buffers – the Protobuf namespace

### Getting Started

In order to serialize an object of the specific class you should firstly mark this class with the ProtobufSerializable attribute and each serializable field – with the ProtobufField attribute:

[ProtobufSerializable]

public class A

{

    /// <summary>

    /// Initializes the default values.

    /// </summary>

    public A()

    {

        B = 666;

    }

    [ProtobufField(1)]

    public uint B { get; set; }

}

Then you’ll be able to serialize an object by creating the serializer and calling the Serialize method and deserialize calling the Deserialize method:

[Test]

public void TestSerializeBasic()

{

    Serializer serializer = Serializer.CreateSerializer(typeof(A));

    object o = new A { B = 150 };

    AssertExtensions.AreEqual(new byte[] { 0x08, 0x96, 0x01 },

            serializer.Serialize(o));

}

### Optional and Required Fields

All properties are optional by default. This means that if a field has no value set then the related tag will not appear in a target message. This behavior is recommended because you’ll not be able to remove a required field and not break a protocol.

But the possibility to define a required field there is:

[ProtobufField(1, required: true)]

public string D { get; set; }

### Repeated Fields

The Cloudy can serialize collections. All you need is to define a property as ICollection:

[ProtobufField(1)]

public ICollection<uint> List { get; set; }  
...

Serializer serializer = Serializer.CreateSerializer(typeof(D));

object o = new D { List = new uint[] { 1, 2, 3 } };

AssertExtensions.AreEqual(new byte[] { 0x08, 0x01, 0x08, 0x02, 0x08, 0x03 },

serializer.Serialize(o));

### Packed Repeated Fields

Packed repeated field is serialized as length-delimited field: sequentially serialized values are used instead of repeating of a single tag with a single value.

[ProtobufSerializable]

public class E

{

    [ProtobufField(4, packed: true)]

    public ICollection<uint> List { get; set; }

}

### Types Mapping

By default the .NET types are serialized into the following Protobuf types:

|  |  |
| --- | --- |
| .NET Type | Protobuf Type |
| bool | Varint |
| int | Signed Varint |
| long | Signed Varint |
| uint | Varint |
| ulong | Varint |
| string | String |
| byte[] | Length-Delimited |
| Guid | Length-Delimited (16 bytes) |
| Enum | Varint |

If you want to change a target Protobuf type (e.g. serialize int as Fixed32) then you may specify the dataType parameter of the ProtobufSerializable attribute:

[ProtobufSerializable]

public class H

{

    [ProtobufField(2, dataType: DataType.FixedInt32)]

    public int Fixed32 { get; set; }

}

Data types are mapped into the target Protobuf types as follows:

|  |  |
| --- | --- |
| DataType | Protobuf Type |
| Bool | Varint |
| Bytes | Length-Delimited |
| FixedInt32 | Fixed32 |
| FixedInt64 | Fixed64 |
| FixedUInt32 | Fixed32 |
| FixedUInt64 | Fixed64 |
| SignedVarint | Signed Varint |
| String | String |
| UnsignedVarint | Varint |
| Guid | Length-Delimited (16 bytes) |

## Messaging Utility Classes – the Messaging namespace

### MessageStream

This is the utility class for convenient sequential reading and writing of messages. Wraps a Stream object and provides the Read and Write methods. Thread-safe.

#### Example

using (MemoryStream stream = new MemoryStream())

{

    MessageStream messageStream = new MessageStream(stream);

    foreach (object message in

        new object[] { new A { B = 1 }, new A { B = 2 } })

    {

        messageStream.Write(message);

    }

}

## Helper Classes – the Helpers namespace

### UdpStream

Implements the [Stream](http://msdn.microsoft.com/en-us/library/system.io.stream.aspx) interface. That allows interacting with an UDP connection as if it was simply a [Stream](http://msdn.microsoft.com/en-us/library/system.io.stream.aspx). This is useful in UDP-messaging via the [Protobuf](#_Protocol_Buffers_–) protocol.

Yes, there is the [NetworkStream](http://msdn.microsoft.com/en-us/library/system.net.sockets.networkstream.aspx) class, but unfortunately [one can’t use NetworkStream for UDP](http://social.msdn.microsoft.com/Forums/en/netfxnetcom/thread/238e2a15-2cf1-4cc8-8e18-87d1e9372178).

#### Example

UdpStream stream1 = new UdpStream(new UdpClient(new IPEndPoint(IPAddress.Any, 1234)));

UdpStream stream2 = new UdpStream(new UdpClient());

stream2.Client.Connect("localhost", 1234);

byte[] buffer = new byte[] { 0x01, 0x02, 0x03, 0x04 };

stream2.Write(buffer, 0, buffer.Length);

foreach (byte b in buffer)

{

    Assert.AreEqual(b, (byte)stream1.ReadByte());

}