|  |
| --- |
| **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](#_Toc308736009)

[2 Components 3](#_Toc308736010)

[2.1 Protocol Buffers – the Protobuf namespace 3](#_Toc308736011)

[2.1.1 Getting Started 3](#_Toc308736012)

[2.1.2 Optional and Required Fields 3](#_Toc308736013)

[2.1.3 Repeated Fields 3](#_Toc308736014)

[2.1.4 Packed Repeated Fields 4](#_Toc308736015)

[2.1.5 Types Mapping 4](#_Toc308736016)

[2.2 Messaging – the Messaging namespace 5](#_Toc308736017)

[2.2.1 Messaging Overview 5](#_Toc308736018)

[2.2.2 MessageStream 5](#_Toc308736019)

[2.2.3 MessageDispatcher 5](#_Toc308736020)

[2.2.4 Understanding Data Transfer Objects 6](#_Toc308736021)

# 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
* Networking namespace
* Nodes namespace

# 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 | Unsigned Varint |
| int | Signed Varint |
| long | Signed Varint |
| uint | Unsigned Varint |
| ulong | Unsigned Varint |
| string | String |
| byte[] | Length-Delimited |
| Guid | Length-Delimited (16 bytes) |
| Enum | Unsigned Varint |
| ICollection<T> | Repeated T |
| Nullable<T> | Optional T |
| *Any other class* | Attempted to be serialized as an Embedded Message |

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 |
| Embedded Message | Length-Delimited |
| FixedInt32 | Fixed32 |
| FixedInt64 | Fixed64 |
| FixedUInt32 | Fixed32 |
| FixedUInt64 | Fixed64 |
| SignedVarint | Signed Varint |
| String | String |
| UnsignedVarint | Varint |
| Guid | Length-Delimited (16 bytes) |

## Messaging – the Messaging namespace

### Messaging Overview

A client code reads messages from and writes messages to a network using the MessageStream class.

By design, a slave node has several UDP peer-to-peer connections to other nodes. A slave node listens for incoming messages on a specific port and (maybe) replies to other nodes on specific addresses and ports. In order to track messages and ensure they are delivered the MessageDispatcher class is used.

All messages are packed into DTOs ([Data Transfer Objects](#_Understanding_Data_Transfer)).

### 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);

    }

}

### MessageDispatcher

The core messaging class. It’s responsibilities:

1. Ensuring messages are delivered by sending delivery notifications.
2. Buffering of incoming message.
3. Asynchronous sending and receiving messages.

The MessageDispatcher class aggregates at least one message stream that is used to read incoming messages from.

In the case of UDP the problem is that an input stream will receive messages from all other nodes together, while sending to other nodes should be done through different output streams. Thus:

1. Each dispatcher has a FromId (an unique ID of a sender).
2. MessageDispatcher invokes a specific *stream resolver* in order to determine the output stream.

A user of a MessageDispatcher should pass these in the constructor:

dispatcher = new MessageDispatcher(Options.ClientId,  
    ResolveStream, new MessageStream(client.GetStream()));

In the simplest case (e.g., TCP is used) a stream resolver can simply return the input stream:

private static bool ResolveStream(Guid id, out MessageStream stream)  
{  
    stream = dispatcher.InputStream;  
    return id == Options.ServerId;  
}

In a common case (UDP) a client code should hold message streams (on UDP streams) to all the communicating nodes and return one of these streams by the specified unique ID.

### Understanding Data Transfer Objects

There are two main problems when sending and receiving messages:

1. Messages in [Protocol Buffers](http://code.google.com/p/protobuf/) format are not self-describing. That means that in common case we cannot predict what type of message we should read as we doesn’t know its structure.
2. MessageDispatcher should be able to send delivery notifications immediately after receiving of a message without knowing of its structure.

To resolve them the concept of [DTO](http://en.wikipedia.org/wiki/Data_transfer_object)s where used. The DTO in Cloudy consists of:

|  |  |  |
| --- | --- | --- |
| Property | Type | Description |
| FromId | Guid | The unique identifier of a sender |
| TrackingId | Int32 | The unique (within all existing DTOs) identifier of this DTO during transferring |
| Tag | Nullable<Int32> | User specified value that indicates a type of the message |
| Value | T | User-specific data (serialized as embedded message) |

Actually, a developer can now nothing about DTOs in order to use Cloudy, because Cloudy hides using of them. If you simply know the type of an expected message, you can invoke the MessageDispatcher.Receive<TResult> method directly. Else, all you need is to receive a message:

int? tag;  
Guid fromId;  
ICastableValue dto = dispatcher.Receive(out fromId, out tag);

And extract needed data after analyzing of a message type:

if (tag == Tags.Says)  
{  
    Console.WriteLine("Server says: {0}", dto.Get<SaysValue>().Message);  
}

Where Message is just a serializable property of the serializable SaysValue class.

MessageStream is able to read and write tagged messages via hidden usage of DTO’s:

messageStream.Write(tags[i], new A { UIntValue = values[i]});  
...  
int? tag;  
uint value = messageStream.Read(out tag).Get<A>().UIntValue;