Inetlab.SMPP

.NET implementation of SMPP protocol for two-way SMS messaging

Table of Contents

Introduction

Overview

Adding a full-featured SMS functionality to your own product can be a tough job. Of course, you can always send direct queries to some Web-to-SMS gateway, but they do not always work properly, or don't support certain options you may need.

Inetlab.SMPP is a .NET library working via SMPP v3.4 protocol. With it you can build SMS sending functions into your own .NET application with minimum efforts. Now you don't have to spend hours trying to figure out the protocols; instead you can develop a ready-made application with Inetlab.SMPP library.

The library supports developing both SMPP client and SMPP server applications capable of sending thousands of SMS messages per minute, working in both directions (that is, sending and receiving messages). Inetlab.SMPP library provides a simple way to implement all standard SMS functions, starting from concatenated messages and flash SMS to immediate SMS delivery via asynchronous multi-thread sending and the support for keeping an SMSC connection alive with EnquireLink. A server-side application made with Inetlab.SMPP supports multiple client connections, works flawlessly thanks to multi-threading and provides 100% safety by means of a secure SSL connection.

Inetlab.SMPP is available both as a .NET library and in source codes.

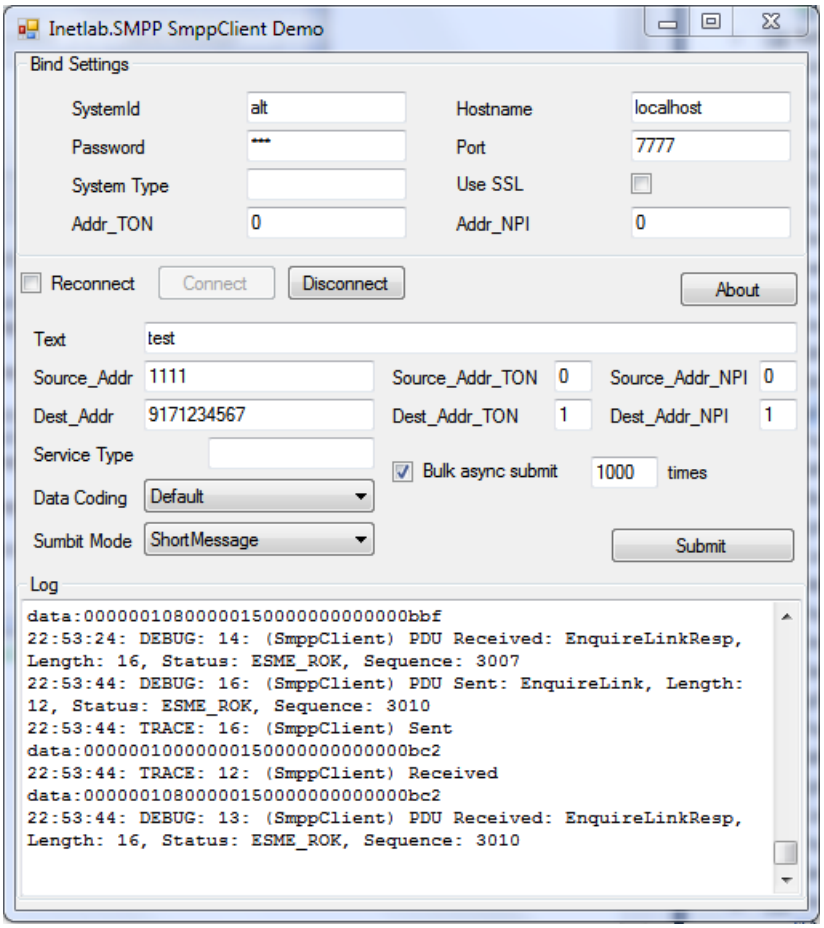
Inetlab.SMPP features:

- Supports concatenated messages in full
- Sends and receives SMS messages
- Works with any language including Arabic, Chinese, Hebrew, Russian, Greek. Also supports Unicode messages.
- Supports Flash SMS and WAP Push messages in full
- Reliable bulk SMS-sending - up to 500 messages per second
- Keeps a connection to SMSC server alive via EnquireLink
- Works via SSL connection
- Server-side application supports multiple client connections

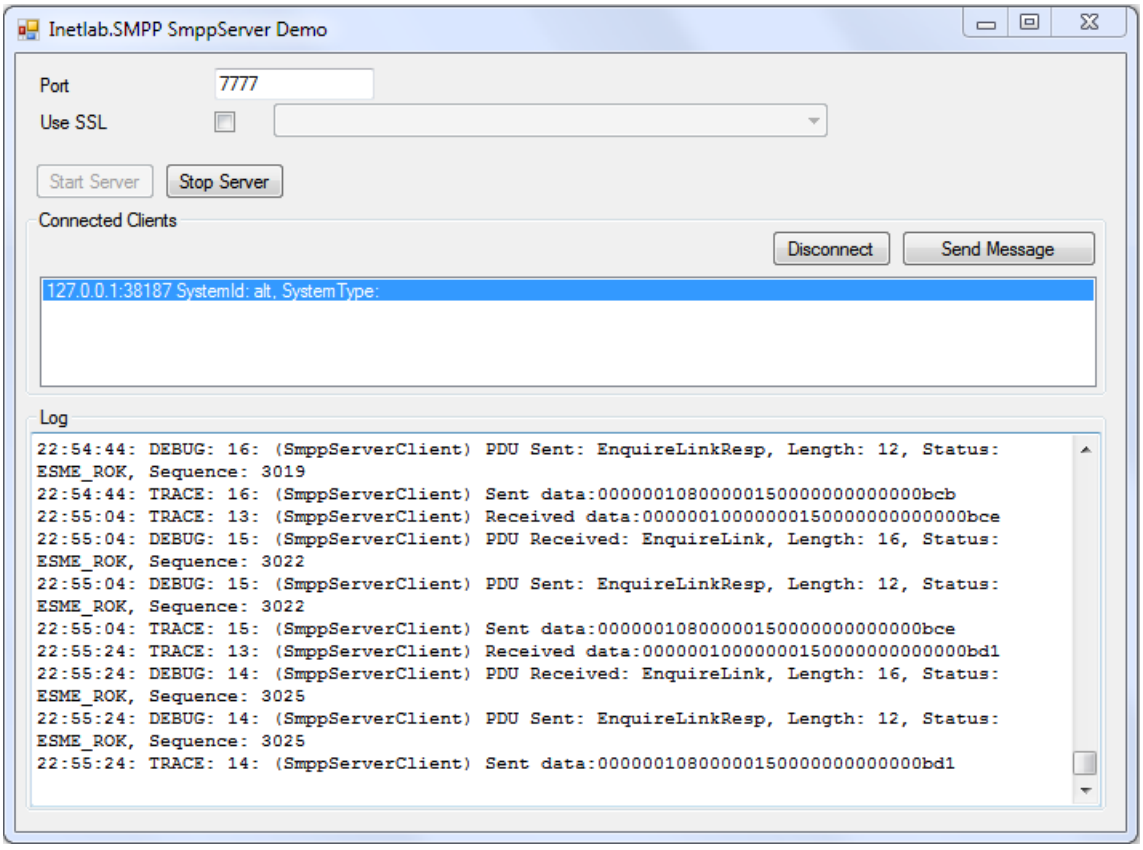
Demo applications

You can download sample application and learn how to implement solutions with Inetlab.SMPP library

SMPP Client


Inetlab.SMPP SmppClient Demo
Bind Settings
SystemId
alt
Hostname
localhost
Password
***
Port
7777
System Type
Use SSL
Addr_TON
0
Addr_NPI
0
Reconnect
Connect
Disconnect
About
Text
test
Source_Addr
1111
Source_Addr_TON
0
Source_Addr_NPI
0
Dest_Addr
9171234567
Dest_Addr_TON
1
Dest_Addr_NPI
1
Service Type
Bulk async submit
1000
times
Data Coding
Default
Sumbit Mode
ShortMessage
Submit
Log
data:00000010800000150000000000000bbf
22:53:24: DEBUG: 14: (SmppClient) PDU Received: EnquireLinkResp,
Length: 16, Status: ESME_ROK, Sequence: 3007
22:53:44: DEBUG: 16: (SmppClient) PDU Sent: EnquireLink, Length:
12, Status: ESME_ROK, Sequence: 3010
22:53:44: TRACE: 16: (SmppClient) Sent
data:00000010000000150000000000000bc2
22:53:44: TRACE: 12: (SmppClient) Received
data:00000010800000150000000000000bc2
22:53:44: DEBUG: 13: (SmppClient) PDU Received: EnquireLinkResp,
Length: 16, Status: ESME_ROK, Sequence: 3010

SMPP Server


Inetlab.SMPP SmppServer Demo
Port
7777
Use SSL
Start Server
Stop Server
Connected Clients
Disconnect
Send Message
127.0.0.1:38187 SystemId: alt, SystemType:
Log
22:54:44: DEBUG: 16: (SmppServerClient) PDU Sent: EnquireLinkResp, Length: 12, Status:
ESME_ROK, Sequence: 3019
22:54:44: TRACE: 16: (SmppServerClient) Sent data:00000010800000150000000000000bcb
22:55:04: TRACE: 13: (SmppServerClient) Received data:00000010000000150000000000000bce
22:55:04: DEBUG: 15: (SmppServerClient) PDU Received: EnquireLink, Length: 16, Status:
ESME_ROK, Sequence: 3022
22:55:04: DEBUG: 15: (SmppServerClient) PDU Sent: EnquireLinkResp, Length: 12, Status:
ESME_ROK, Sequence: 3022
22:55:04: TRACE: 15: (SmppServerClient) Sent data:00000010800000150000000000000bce
22:55:24: TRACE: 13: (SmppServerClient) Received data:00000010000000150000000000000bd1
22:55:24: DEBUG: 14: (SmppServerClient) PDU Received: EnquireLink, Length: 16, Status:
ESME_ROK, Sequence: 3025
22:55:24: DEBUG: 14: (SmppServerClient) PDU Sent: EnquireLinkResp, Length: 12, Status:
ESME_ROK, Sequence: 3025
22:55:24: TRACE: 14: (SmppServerClient) Sent data:00000010800000150000000000000bd1

Changelog

[2.6.13] - 2019-08-14

Fixed

- multithread-issue with ConnectedClients in SmppServer class
- set SmppClient.SystemID and SmppClient.SystemType properties when client is bound.

[2.6.12] - 2019-07-26

Added

- convert UserDataHeader to and from byte array
- SmppTime functions for formatting and parsing scheduled delivery times and expiry times in PDU.
- EnsureReferenceNumber method that sets next reference number for a list of concatenated PDUs.
- property InactivityTimeout in the class SmppClientBase. Default is 2 minutes. Connection will be dropped when in specified period of time no SMPP message was exchanged. InactivityTimeout doesn't work when EnquireLinkInterval is defined.

Fixed

- SmppServer: when client.ReceiveSpeedLimit is set to any value, first message is always throttled.
- text splitting: Incorrect message length of 1st PDU when text encoded in GSM encoding and contains extended characters
- ReferenceNumber=0 for submitted concatenated PDUs.

[2.6.11] - 2019-04-20

Fixed

- Connection failed. Error Code: 10048. Only one usage of each socket address (protocol/network address/port) is normally permitted. Occurs when call Connect method from different threads at the same time.

[2.6.10] - 2019-04-19

Fixed

- exceptions by incorrect disconnect.

[2.6.9] - 2019-04-15

Fixed

- Request property is null in received response PDU class.

Added

- ReceiveBufferSize and SendBufferSize properties for SmppClientBase.

[2.6.8] - 2019-03-27

Fixed

- wrong text splitting in SMS builder for GSMPackedEncoding.

[2.6.7] - 2019-03-27

Fixed

- StackOverflowException by submitting array of SubmitMulti.
- destination addresses serialization for SubmitMulti

- short message length calculation

[2.6.6] - 2019-03-25

Fixed

- exception in GetMessageText method for DeliverSm with empty text.

[2.6.5] - 2019-03-18

Fixed

- exception in GetMessageText method for DeliverSm without receipt.

[2.6.4] - 2019-03-15

Fixed

- missed last character in the last segment of the concatenated message created with SMS builders.

Added

- Extension method smppPdu.GetMessageText(EncodingMapper) as replacement for MessageText property in a PDU class.
- TLVCollection.RegisterParameter(ushort tag) method for registering custom TLV parameter type for any tag value. It helps to represent some complex parameters as structured objects. Example: var parameter = pdu.Parameters.Of();

Changed

- MessageText property in PDU classes is obsolete. Use the function client.EncodingMapper.GetMessageText(pdu) or pdu.GetMessageText(client.EncodingMapper) to get th message text contained in the PDU.

[2.6.3] - 2019-03-04

Fixed

- failed to raise some events with attached delegate that doesn't has target object.

Improved

- FileLogger multi-threading improvements.

[2.6.2] - 2019-02-07

Added

- ILogFactory interface with implementations for File and Console

Fixed

- client hangs by Dispose when it was never connected

[2.6.1] - 2019-02-04

Fixed

- Cannot send 160 characters in one part SMS in GSM Encoding

[2.6.0] - 2019-01-14

Added

- ProxyProtocolEnabled property for SmppServerClient class. This property should be enabled in evClientConnected event handler to detect proxy protocol in the network stream of connected client.

- Signed with Strong Name
- ClonePDU, Serialize methods for SmppPDU classes.
- SMS.ForData method for building concatenated DataSm PDUs.
- SMS.ForDeliver is able to create delivery receipt in MessagePayload parameter.

Fixed

- SmppServer stops accepting new connections by invalid handshake
- Text splitter for building concatenated message parts
- Event evClientDataSm didn't raise in the SmppServer.
- Sometimes SmppServerClient doesn't disconnect properly in SmppServer
- concurrency issues in MessageComposer
- library sends response with status ESME_ROK when SmppServer has no attached event handler for a request PDU. It should send unsuccess status f.i. ESME_RINVCMDID.

API Changes

- Replaced methods AddMessagePayload, AddSARReferenceNumber, AddSARSequenceNumber, AddSARTotalSegments and AddMoreMessagesToSend with corresponding classes in Inetlab.SMPP.Parameters namespace.
- Renamed the property "Optional" to "Parameters" in PDU classes. (backwards-compatible)
- Removed unnecessary TLV constructor with length parameter. Length is always equal to value array length.
- Removed ISmppMessage interface
- Renamed namespace Inetlab.SMPP.Common.Headers to Inetlab.SMPP.Headers
- Rename propery UserDataPdu to UserData for classes SubmitSm, SubmitMulti DeliverSm, ReplaceSm. (backwards-compatible)
- MessageInPayload method tells SMS builder to send complete message text in message_payload parameter. With optional messageSize method parameter you can decrease the size of message segment if you need to send concatenation in SAR parameters.
- Simplified ILog interface

[2.5.4] - 2018-09-16

Changed

- MessageComposer.Timeout property to TimeStamp

Added

- SmppClient.Submit methods with IEnumerable parameter
- better documentation

Fixed

- Hanlde SocketException OperationAborted when server stops

[2.5.3] - 2018-09-08

Fixed

- SubmitSpeedLimit is ignored
- sometimes SMPP PDU reading is failed

[2.5.2] - 2018-08-06

Fixed

- Messages with data coding Class0 (0xF0) are split up in wrong way

[2.5.1] - 2018-07-30

Fixed

- wrong BindingMode for SmppServerClient after Unbind.

[2.5.0] - 2018-07-29

Added

- Automatic detection for Proxy protocol https://www.haproxy.com/blog/haproxy/proxy-protocol/ ### Implemented
- Unbind logic for SmppClient and SmppServerClient classes

[2.4.1] - 2018-06-19

Fixed

- issue with licensing module

[2.4.0] - 2018-05-30

Added

- Automatic connection recovery.

[2.3.2] - 2018-04-20

Added

- MessageComposer allows to get its items for concatenated messages. ### Changed
- creation for user data headers types.

[2.3.1] - 2018-04-18

Fixed

- PDU reader and writer
- split text on concatenation parts

[2.3.0] - 2018-03-18

Added

- SmppClientBase.SendQueueLimit limits the number of sending SMPP messages to remote side. Delays further SMPP requests when limit is exceeded.

Changed

- SmppServerClient.ReceiveQueueLimit replaced with SmppClientBase.ReceivedRequestQueueLimit

Improved

- improved: processing of connect and disconnect.

[2.2.0] - 2018-02-01

Improved

- better processing of request and response PDU

Changed

- Flow Control. SmppServerClient.ReceiveQueueLimit defines allowed number of SMPP requests in receive queue. If receive queueu is full, library stops receive from network buffer and waits until queue has a place again. It is better alternative for ESME_RMSGQFUL response status. ### Fixed

- MessageComposer raises evFullMessageReceived sometimes two times by processing concatenated message with two parts.

[2.1.2] - 2017-12-11

Improved

- internal queue for processing PDU.

[2.1.1] - 2017-12-10

Improved

- processing of connect and disconnect

Added

- From and To methods with SmeAddress parameter to SMS Builders

[2.1.0] - 2017-10-18

Added

- SendSpeedLimit property for SmppClientBase class, that limits number of requests per second to remote side
- Priority processing for response PDUs.
- Name property to distinguish instances in logger
- Deliver method in SmmpServerClient class
- SubmitData method in SmppClientBase class

[2.0.1] - 2017-10-06

Added

- decode receipt for IntermediateDeliveryNotification

Fixed

- sequence number generation

[2.0.0] - 2017-08-15

- first version for .NET Standard 1.4

Hello World

```
public static async Task SendHelloWorld()
{
    using (SmppClient client = new SmppClient())
    {

        if (await client.Connect("localhost", 7777))
        {
            BindResp bindResp = await client.Bind("1", "2");

            if (bindResp.Header.Status == CommandStatus.ESME_ROK)
            {
                var submitResp = await client.Submit(
                    SMS.ForSubmit()
                        .From("111")
                        .To("222")
                        .Coding(DataCodings.UCS2)
                        .Text("Hello World!"));

                if (submitResp.All(x => x.Header.Status == CommandStatus.ESME_ROK))
                {
                    client.Logger.Info("Message has been sent.");
                }
            }

            await client.Disconnect();
        }
    }
}
```

Migration from v1.x to 2.x

How to solve some compile issues:

SmppClient

1. Missing `BatchMonitor` class. Use instead `client.Submit(IEnumerable<SubmitSm> batch)`. It waits when all responses will be received for a batch.
2. Events `evBindComplete`, `evSubmitComplete`, `evQueryComplete` were depricated. It is possible to use async await pattern or ContinueWith for corresponding `Bind`, `Submit`, `Query` methods.
3. `_client.AddressRange`, `_client.AddrNpi` and `_client.AddrTon` must be specified as `_client.EsmeAddress = new SmeAddress(AddressRange, AddrTon, AddrNpi);`
4. Sequence number and Command Status are moved to Header property of PDU.
    - `data.Status` replaced with `data.Header.Status`,
    - `data.Sequence` replaced with `data.Header.Sequence`
5. `client.GetMessageText` is moved to `client.EncodingMapper.GetMessageText`
6. PDU Properties
    - `SourceAddrTon`, `SourceAddrNpi`, `SourceAddr` replaced with `SourceAddress` of type `SmeAddress`
    - `DestAddrTon`, `DestAddrNpi`, `DestAddr` replaced with `DestinationAddress` of type `SmeAddress`
    - `UserDataPdu` replaced with `UserData`
    - `Optional` replaced with `Parameters`
7. Property `MessageText` in `SubmitSm`, `SubmitMulti`, `DeliverSm`, `DataSm`, `ReplaceSm` classes is deprecated. Use the method `pdu.GetMessageText(client.EncodingMapper)`.
8. Method `SmppClientBase.MapEncoding` moved to `SmppClientBase.EncodingMapper.MapEncoding`

SmppServer

1. Namespace for `SmppServerClient` class changed to `Inetlab.SMPP`.
2. IPEndPoint of the server must be specified in `SmppServer` constructor, instead of `Start` method of this class.

Serialization

Method `submitSm.Serialize` can be replaced with extension method:

```
   byte[] pduData = submitSm.Serialize(client.EncodingMapper);
```

Static method `SubmitSm.Deserialize` can be replaced with code:

```
   byte[] pduData = ...;

SubmitSm pdu =  pduData.Deserialize<SubmitSm>(client.EncodingMapper);
```

Getting Started with Inetlab.SMPP

Connecting to SMPP server

For connecting to SMPP server you need to get SMPP Account (username and password) from any SMPP provider or mobile network operator.

```
readonly SmppClient _client = new SmppClient();

public async Task Connect()
{
    if (await _client.Connect("smpp.server.com", 7777))
    {
        _log.Info("Connected to SMPP server");

        BindResp bindResp = await _client.Bind("username", "password", ConnectionMode.Transceiver);

        if (bindResp.Header.Status == CommandStatus.ESME_ROK)
        {
            _log.Info("Bound with SMPP server");
        }
    }
}
```

In Bind method you can specify @Inetlab.SMPP.Common.ConnectionMode. There are 3 modes you can use:

- Transmitter - allows only to send SMPP commands to the SMSC and receive corresponding SMPP responses from the SMSC.
- Receiver - allows only to receive SMPP commands from SMSC and send corresponding SMPP responses.
- Transceiver - allows to send and receive SMPP commands in SMSC.

Default ConnectionMode is Transceiver.

If you use same SMPP account in several applications you must bind only one application in Receiver or Transceiver connection mode. Other applications should be bound with Transmitter mode. Otherwise SMPP server can deliver messages to an application where you don't expect them.

Sending SMS message

For sending simple SMS message from your service number (short code **1111**) to the phone number **79171234567** and with requested delivery receipt you can use fluent PDU builder:

```
public async Task SendMessage()
{
    IList<SubmitSmResp> responses = await _client.Submit(
        SMS.ForSubmit()
            .Text("Test Test Test Test Test Test Test Test Test Test")
            .From("1111")
            .To("79171234567")
            .Coding(DataCodings.UCS2)
            .DeliveryReceipt()
    );
}
```

or the same code, but with explicit instance creation:

```
public SubmitSm CreateSubmitSm()
{
    SubmitSm sm = new SubmitSm();
    sm.UserData.ShortMessage = _client.EncodingMapper.GetMessageBytes("Test Test Test Test Test Test Test Test Test Test", DataCodings.Default);
    sm.SourceAddress = new SmeAddress("1111", AddressTON.NetworkSpecific, AddressNPI.Unknown);
    sm.DestinationAddress = new SmeAddress("79171234567", AddressTON.Unknown, AddressNPI.ISDN);
    sm.DataCoding = DataCodings.UCS2;
    sm.SMSCReceipt = SMSCDeliveryReceipt.SuccessOrFailure;

    return sm;
}
```

There is only one difference. First code allows you to send text with any length, but second code only up to 160 symbols. First code returns list of responses because it will split long message text on smaller concatenated parts.

Receiving SMS messages

When client is bound with SMPP server in Transceiver or Receiver mode it will be able to receive SMS messages. Every time when SMPP client receives SMS message it raises @Inetlab.SMPP.SmppClient.evDeliverSm event. Event handler method should be attached before bind to server.

```
_client.evDeliverSm += new DeliverSmEventHandler(client_evDeliverSm);
```

Your code should be able to receive concatenated long SMS messages. Message Composer helps to combine them and get full message text. Delivery Receipt can also be received with this handler method.

```
private readonly MessageComposer _composer = new MessageComposer();

private void client_evDeliverSm(object sender, DeliverSm data)
{
    try
    {
        //Check if we received Delivery Receipt
        if (data.MessageType == MessageTypes.SMSCDeliveryReceipt)
        {
            //Get MessageId of delivered message
            string messageId = data.Receipt.MessageId;
            MessageState deliveryStatus = data.Receipt.State;
        }
        else
        {
            // Receive incoming message and try to concatenate all parts
            if (data.Concatenation != null)
            {
                _composer.AddMessage(data);

                _log.Info("DeliverSm part received : Sequence: {0} SourceAddr: {1} Concatenation ( {2} )
Coding: {3} Text: {4}",
                    data.Header.Sequence, data.SourceAddress, data.Concatenation, data.DataCoding,
_client.EncodingMapper.GetMessageText(data));


                if (_composer.IsLastSegment(data))
                {
                    string fullMessage = _composer.GetFullMessage(data);
                    _log.Info("Full message: " + fullMessage);
                }
            }
            else
            {
                _log.Info("DeliverSm received : Sequence: {0} SourceAddr : {1} Coding : {2} MessageText :
{3}",
                        data.Header.Sequence, data.SourceAddress, data.DataCoding,
_client.EncodingMapper.GetMessageText(data));
            }
        }
    }
    catch (Exception ex)
    {
        data.Response.Header.Status = CommandStatus.ESME_RX_T_APPN;
        _log.Error("Failed to process DeliverSm", ex);
    }
```

Possible reasons why you don't receive incoming messages:

- SMPP account doesn't have right to receive SMS messages.
- Wrong SMS routing configuration on SMPP server.
- SMPP client has been bound as Transmitter.
- SMPP client was not attached to evDeliverSm event handler.
- SMPP account is used in two or more application. SMSC sends messages to application where @Inetlab.SMPP.PDU.DeliverSm is not expected.

SMPP Client FAQ

Can library split text into multiple concatenated SMS-parts?

Text will be split automatically when you use SMS builders. Following example covers most usage scenarios

```
Client.Submit(SMS.ForSubmit()
     .Text("Test Test Test Test Test Test Test Test Test Test")
     .From("1111")
     .To("2222")
     .DeliveryReceipt()
     .Coding(DataCodings.Default)
     );
```

How can I send Flash SMS?

In order to send Flash SMS you need to specify one of the following data coding in the @Inetlab.SMPP.PDU.SubmitSm class: UnicodeFlashSMS, DefaultFlashSMS

How can I set sequence number before sending PDU

@Inetlab.SMPP.SMS Builder has @Inetlab.SMPP.Builders.IBuilder.Create method that returns @Inetlab.SMPP.PDU.SubmitSm list with sequence numbers set to 0. You can assign next number from the @Inetlab.SMPP.Common.SequenceGenerator and pass this PDU list to @Inetlab.SMPP.SmppClient.Submit(SubmitSm[]) method.

```
 IList<SubmitSm> list = SMS.ForSubmit()
                .From(_config.ShortCode, AddressTON.NetworkSpecific, AddressNPI.Unknown)
                .To(message.PhoneNumber)
                .Text(message.Text)
                .DeliveryReceipt()
                .Set(s=>s.Tag = "myId")
                .Create(_client);

            foreach (SubmitSm sm in list)
            {
                sm.Sequence = _client.SequenceGenerator.NextSequenceNumber();
            }

           var resp = await _client.Submit(list);
```

Example: Read messages from database and send them as fast as possible

```
var messageList = _query.GetNext1000UnsentMessages();

List<SubmitSm> pdus = new List<SubmitSm>();
foreach (var message in messageList)
{
    var pduBuilder = SMS.ForSubmit()
                    .From(_config.ShortCode)
                    .To(message.PhoneNumber)
                    .Text(message.Text);

    pdus.AddRange(pduBuilder.Create(_client));
}

SubmitSmResp[] resp = _client.Submit(pdus.ToArray());
```

How to create SubmitMulti PDUs for multiply recipients

```
var pduBuilder = SMS.ForSubmitMulti()
                  .From(_config.ShortCode)
                  .Text(message.Text);

foreach (string phoneNumber in message.PhoneNumbers) {
   pduBuilder.To(phoneNumber);
}

IList<SubmitMulti> pduList = pduBuilder.Create(_client);
```

Delivery Receipt

Receipt format

Often you want to get delivery status for SMS message. SMPP protocol provides ability to request delivery receipt in submitted PDU. There are two ways how you can do it with the component.

```
submitSm.RegisteredDelivery = 1;
```

or

```
   IList<SubmitSmResp> responses = client.Submit(
         SMS.ForSubmit()
         .Text("Test").From("1111").To("79171234567")
         .DeliveryReceipt()
        );
```

As the result SMPP server will deliver receipt to client application. On the client side it can be received using @Inetlab.SMPP.SmppClient.evDeliverSm event. Delivery receipt format is SMSC vendor specific, but typical format is

id:IIIIIIIIII sub:SSS dlvrd:DDD submit date:YYMMDDhhmm done date:YYMMDDhhmm stat:DDDDDDD err:E Text: . . . .

This text format is represented in the library as @Inetlab.SMPP.Common.Receipt class.

It has following properties:

MessageId - The message ID allocated to the message by the SMSC when originally submitted. You can get it from @Inetlab.SMPP.PDU.SubmitSmResp or @Inetlab.SMPP.PDU.SubmitMultiResp.

Submitted - Number of short messages originally submitted. This is only relevant when the original message was submitted to a distribution list within @Inetlab.SMPP.PDU.SubmitMulti.

Delivered - Number of short messages delivered to distribution list with SubmitMulti.

SubmitDate - The time and date at which the short message was submitted.

DoneDate - The time and date at which the short message reached it's final state.

ErrorCode - Network specific error code or an SMSC error code for the attempted delivery of the message.

Text - The first 20 characters of the short message.

State - The final status of the message. The value could be on of the following:

STATE	DESCRIPTION
Delivered	Message is delivered to destination
Expired	Message validity period has expired
Deleted	Message has been deleted
Undeliverable	Message is undeliverable
Accepted	Message is in accepted state (i.e. has been manually read on behalf of the subscriber by customer service)

STATE	DESCRIPTION
Unknown	Message is in invalid state
Rejected	Message is in a rejected state

Note

Library sends @Inetlab.SMPP.PDU.DeliverSmResp with status ESME_RX_T_APPN to SMPP server when evDeliverSm event hendler method throws an exception.

How to tie submitted message with delivery receipt

SMS message in SMPP protocol actually is represented as one or many PDUs. When text is longer that 140 octets library sends text as concatenated SMS parts (PDU). One part can be represented as @Inetlab.SMPP.PDU.SubmitSm class or @Inetlab.SMPP.PDU.SubmitMulti class.

Before sending @Inetlab.SMPP.PDU.SubmitSm or @Inetlab.SMPP.PDU.SubmitMulti PDU you need to assign @Inetlab.SMPP.Common.SmppHeader.Sequence number to it.

```
public async Task SendMessage(TextMessage message)
{
    IList<SubmitSm> list = SMS.ForSubmit()
        .From(_config.ShortCode)
        .To(message.PhoneNumber)
        .Text(message.Text)
        .DeliveryReceipt()
        .Create(_client);

    foreach (SubmitSm sm in list)
    {
        sm.Header.Sequence = _client.SequenceGenerator.NextSequenceNumber();
        _clientMessageStore.SaveSequence(message.Id, sm.Header.Sequence);
    }

  var responses = await _client.Submit(list);

    foreach (SubmitSmResp resp in responses)
    {
        _clientMessageStore.SaveMessageId(message.Id, resp.MessageId);
    }
}
```

At the same time you need to store @Inetlab.SMPP.Common.SmppHeader.Sequence to the database. For one *message.Id* you need to store several @Inetlab.SMPP.Common.SmppHeader.Sequence.

In response to @Inetlab.SMPP.PDU.SubmitSm PDU your application receives @Inetlab.SMPP.PDU.SubmitSmResp PDU. This response has same @Inetlab.SMPP.Common.SmppHeader.Sequence number and @Inetlab.SMPP.PDU.SubmitSmResp.MessageId generated by the server.

When you receive delivery receipt in the event @Inetlab.SMPP.SmppClient.evDeliverSm, server sends same @Inetlab.SMPP.Common.Receipt.MessageId which you can use for updating status of submitted SMS text.

```
private void ClientOnEvDeliverSm(object sender, DeliverSm data)
{
    if (data.MessageType == MessageTypes.SMSCDeliveryReceipt)
    {
        _clientMessageStore.UpdateMessageStatus(data.Receipt.MessageId, data.Receipt.State);
    }
}
```

SMS Text considered as delivered when all sms parts are in Delivered state.

For this purpose you can create 2 tables in the database.

1) **outgoing_messages** for all outgoing SMS messages

NAME	DESCRIPTION
messageId	id of the message
messageText	long message text

2) **outgoing_message_parts** for all PDUs generated for each message

NAME	DESCRIPTION
messageId	reference to messageId field in the *outgoing_messages*
sessionId	any unique id generated when SmppClient connects to the server. sequenceNumber is unique only in one SMPP session.
sequenceNumber	number generated before sending PDU
serverMessageId	message id received from the server.
status	status received in the delivery receipt

SubmitMulti. Send message to multiple destinations

The @Inetlab.SMPP.PDU.SubmitMulti command is used to submit SMPP message for delivery to multiple recipients or to one or more Distribution Lists.

Recipients can be specified with multiple invocation of method @Inetlab.SMPP.Builders.ISubmitMultiBuilder.To(System.String)

```
await _client.Submit(SMS.ForSubmitMulti()
     .ServiceType("test")
     .Text("Test Test")
     .From("MyService")
     .To("1111")
     .To("2222")
     .To("3333")
 );
```

this can be done from phone numbers collection

```
var pduBuilder = SMS.ForSubmitMulti()
    .ServiceType("test")
    .Text("Test Test")
    .From("MyService");

foreach (string phoneNumber in phoneNumbers)
{
    pduBuilder.To(phoneNumber);
}
```

another possibility is to create Inetlab.SMPP.Common.DistributionList

```
List<IAddress> destList = new List<IAddress>();

destList.Add(new SmeAddress("11111111111", AddressTON.Unknown, AddressNPI.ISDN));
destList.Add(new DistributionList("my_destribution_list_on_SMPP_Server"));

var submitResponses = await _client.Submit(SMS.ForSubmitMulti()
    .ServiceType("test")
    .Text("Test Test")
    .From("MyService")
    .ToDestinations(destList)
);
```

When @Inetlab.SMPP.PDU.SubmitMultiResp response received it means SMPP server stored message for further delivery to recipients.

@Inetlab.SMPP.PDU.SubmitMulti message for destination address is accepted by SMPP server only when you receive ESME_ROK in all responses in then result list IList<@Inetlab.SMPP.PDU.SubmitMultiResp> and destination address does not exist in @Inetlab.SMPP.PDU.SubmitMultiResp.UnsuccessfulDeliveries of response.

Connection Recovery

Connection recovery works only after first successful bind. SmppClient triggers following events by connection recovery:

- event evConnected - when connected to the server.
- event evRecoverySucceeded - when bind was successful.
- event evDisconnected - when bind was failed.

Connection won't be recovered when you call direct the method client.Disconnect() .

For first successful bind you need to write a Connect method so that it repeats Connect and Bind until it receives status ESME_ROK in BindResp.

Note

If you send Bind in evConnected event handler method, it can cause SmppException when second Bind method is called for already bound client.

Throttling error

SMSC can limit number of submitted PDU for SMPP account. When allowed message limit exceeded, server returns status ESME_RTHROTTLED. To avoid throttling error you can specify a number of messages per second in @Inetlab.SMPP.SmppClient. For this purpose you can define @Inetlab.SMPP.SmppClientBase.SendSpeedLimit property.

```
//Send 10 messages per second
client.SendSpeedLimit = 10;

//Send 1 message every 5 seconds
client.SendSpeedLimit = 1f/5f;

//Send 100 message every 1 minute
client.SendSpeedLimit = new LimitRate(100, TimeStamp.FromMinutes(1));

//Disable send speed limit
client.SendSpeedLimit = LimitRate.NoLimit;
```

SMPP Server FAQ

How to send message to connected client

In following code target client is selected and @Inetlab.SMPP.PDU.DeliverSm message is sent to this client.

```
public async Task DeliverToClient(SmppServerSample.TextMessage message)
{
    string systemId = GetSystemIdByServiceAddress(message.ServiceAddress);

    SmppServerClient client = FindClient(systemId);

    await client.Deliver(SMS.ForDeliver()
        .From(message.PhoneNumber)
        .To(message.ServiceAddress)
        .Text(message.Text)
    );

}
```

How to send messages out from the server on client bind

```
private void OnClientBind(object sender, SmppServerClient client, Bind pdu)
{
    if (client.BindingMode == ConnectionMode.Transceiver || client.BindingMode == ConnectionMode.Receiver)
    {
        //Start messages delivery

        Task messagesTask = DeliverMessagesAsync(client, pdu);

    }
}


private async Task DeliverMessagesAsync(SmppServerClient client, Bind pdu)
{
    var messages = _messageStore.GetMessagesForClient(pdu.SystemId, pdu.SystemType);

    foreach (TextMessage message in messages)
    {
        var pduBuilder = SMS.ForDeliver()
            .From(message.PhoneNumber)
            .To(message.ServiceAddress)
            .Text(message.Text);

        var responses = await client.Deliver(pduBuilder);

        _messageStore.UpdateMessageState(message.Id, responses);
    }
}

public interface IServerMessageStore
{
    IEnumerable<TextMessage> GetMessagesForClient(string systemId, string systemType);
    void UpdateMessageState(string messageId, DeliverSmResp[] responses);
}

public class TextMessage
{
    public string Id { get; set; }
    public string PhoneNumber { get; set; }
    public string Text { get; set; }

    public string ServiceAddress { get; set; }
}
```

How to set MessageId

MessageId must be set on the server side. When server receives SubmitSm or SubmitMulti PDU it generates corresponding response and sets MessageId.

You can change MessageId property in evClientSubmitSm and evClientSubmitMulti event handlers

```
private void ServerOnClientSubmitSm(object sender, SmppServerClient client, SubmitSm data)
{
    data.Response.MessageId = Guid.NewGuid().ToString().Substring(0, 8);
}
```

SMPP Gateway

When you resell SMPP traffic you need to implement SMPP Gateway or SMPP Proxy.

Such application should start at least one @Inetlab.SMPP.SmppServer to be able to receive SMPP commands on TCP port and several @Inetlab.SMPP.SmppClient instances to send message to another SMPP servers (SMSC, Provider).

When customer sends @Inetlab.SMPP.PDU.SubmitSm command toy your server you need to send back a response @Inetlab.SMPP.PDU.SubmitSmResp with assigned @Inetlab.SMPP.PDU.SubmitSmResp.MessageId. Later when you forward this message to another server you will receive another @Inetlab.SMPP.PDU.SubmitSmResp.MessageId from SMSC . This SMSC @Inetlab.SMPP.PDU.SubmitSmResp.MessageId should also be replaced in @Inetlab.SMPP.PDU.DeliverSm (@Inetlab.SMPP.Common.Receipt) for the target client.

You might want to implement smart routing for incomming messages. F.i. when you are going to forward SMS message you can estimate which @Inetlab.SMPP.SmppClient connection accepts destination phone number and which costs cheaper.

When you need only forward SubmitSm messages I suggest following steps:

- Receive SubmitSm from client.
  - Save client's @Inetlab.SMPP.Common.SmppHeader.Sequence number to the database. Possible good idea to save entire PDU.
  - Send @Inetlab.SMPP.PDU.SubmitSmResp to client with his @Inetlab.SMPP.Common.SmppHeader.Sequence number and @Inetlab.SMPP.PDU.SubmitSmResp.MessageId generated on your server side.
- In another process/thread send this @Inetlab.SMPP.PDU.SubmitSm PDU to some SMPP provider.
  - Change @Inetlab.SMPP.PDU.SubmitSm @Inetlab.SMPP.Common.SmppHeader.Sequence number to the next sequence number for the @Inetlab.SMPP.SmppClient that connected to that SMPP provider.
  - Receive Provider's @Inetlab.SMPP.PDU.SubmitSmResp.MessageId in @Inetlab.SMPP.PDU.SubmitSmResp
  - Store Provider's @Inetlab.SMPP.PDU.SubmitSmResp.MessageId and @Inetlab.SMPP.Common.SmppHeader.Sequence number to the same database table as for client's @Inetlab.SMPP.Common.SmppHeader.Sequence number.

These four values:

- Client's sequence number
- Client's MessageId
- Provider's sequence number
- Provider's MessageId help later to find corresponding client that should receive delivery receipt from provider.

When @Inetlab.SMPP.PDU.DeliverSm comes from the provider and contains Delivery Receipt, you should do following steps

- Get Provider's @Inetlab.SMPP.Common.Receipt.MessageId from delivery @Inetlab.SMPP.Common.Receipt.
- Find client's @Inetlab.SMPP.PDU.SubmitSmResp.MessageId and corresponding SMPP user.
- Replace Provider's @Inetlab.SMPP.Common.Receipt.MessageId in @Inetlab.SMPP.PDU.DeliverSm PDU with client's @Inetlab.SMPP.PDU.SubmitSmResp.MessageId
- Send Delivery Receipt to the @Inetlab.SMPP.SmppServerClient that belongs to SMPP user.
- If there is no active connection with the client, place @Inetlab.SMPP.PDU.DeliverSm PDU to the outgoing persistent queue (another database table) and send it when client connects.

Concatenation

Different SMPP providers support different concatenation ways. There are 3 ways:

1) message text in the field **short_message** and concatenation parameters in **user data header**

SMS Builder classes uses this type of concatenation by default. Example how to submit SubmitSm PDUs:

```
public async Task SendConcatenatedMessageInUDH(TextMessage message)
{
    var builder = SMS.ForSubmit()
        .From(_config.ShortCode, AddressTON.NetworkSpecific, AddressNPI.Unknown)
        .To(message.PhoneNumber)
        .Text(message.Text);

    var resp = await _client.Submit(builder);
}
```

Example how to get concatenation parameters from PDU user data header:

```
public Concatenation GetConcatenationFromUDH(SubmitSm data)
{
    ConcatenatedShortMessages8bit udh8 = data.UserData.Headers.Of<ConcatenatedShortMessages8bit>
().FirstOrDefault();

    if (udh8 == null) return null;

    return new Concatenation(udh8.ReferenceNumber, udh8.Total, udh8.SequenceNumber);
}
```

Example how you can manually create SubmitSm instance that contains only one message part with concatenation parameters in user data header:

```
public SubmitSm CreateSumbitSmWithConcatenationInUDH(ushort referenceNumber, byte totalParts, byte partNumber,
string textSegment)
{

    SubmitSm sm = new SubmitSm();
    sm.SourceAddress = new SmeAddress("1111");
    sm.DestinationAddress = new SmeAddress("79171234567");
    sm.DataCoding = DataCodings.Default;
    sm.RegisteredDelivery = 1;
    sm.UserData.ShortMessage = _client.EncodingMapper.GetMessageBytes(textSegment, sm.DataCoding);

    sm.UserData.Headers.Add(new ConcatenatedShortMessage16bit(referenceNumber, totalParts, partNumber));

    return sm;
}
```

2) message text in the field **short_message** and concatenation parameters in **SAR TLV parameters** (sar_msg_ref_num, sar_total_segments, sar_segment_seqnum, more_messages_to_send)

Example how to create SubmitSm instances with SMS Builder:

```
var builder = SMS.ForSubmit()
    .From(_config.ShortCode, AddressTON.NetworkSpecific, AddressNPI.Unknown)
    .To(message.PhoneNumber)
    .Text(message.Text);

builder.ConcatenationInSAR();

var resp = await _client.Submit(builder);
```

Example how to get concatenation parameters from TLV Parameters:

```
public Concatenation GetConcatenationFromTLVOptions(SubmitSm data)
{
    ushort refNumber = 0;
    byte total = 0;
    byte seqNum = 0;

    var referenceNumber = data.Parameters.Of<SARReferenceNumberParamter>().FirstOrDefault();
    if (referenceNumber != null)
    {
        refNumber = referenceNumber.ReferenceNumber;
    }
    var totalSegments = data.Parameters.Of<SARTotalSegmentsParameter>().FirstOrDefault();
    if (totalSegments != null)
    {
        total = totalSegments.TotalSegments;
    }
    var sequenceNumber = data.Parameters.Of<SARSequenceNumberParameter>().FirstOrDefault();
    if (sequenceNumber != null)
    {
        seqNum = sequenceNumber.SequenceNumber;
    }

    return new Concatenation(refNumber, total, seqNum);
}
```

3) message text in the TLV parameter **message_payload** and concatenation parameters in **SAR TLV parameters**

Example how to create SubmitSm instances with SMS Builder:

```
var builder = SMS.ForSubmit()
    .From(_config.ShortCode, AddressTON.NetworkSpecific, AddressNPI.Unknown)
    .To(message.PhoneNumber)
    .Text(message.Text);

builder.MessageInPayload();

var resp = await _client.Submit(builder);
```

泷 Tip

Please ask your SMPP provider which type of concatenation is supported.

SMPP Address

SMPP Address (SME Address) is comprised of 3 parameters: **Address**, **TON**, **NPI**.

Address is text field that represents originator and/or recipient of the message.

TON defines Type of Number

NAME	VALUE
Unknown	0
International	1
National	2
Network Specific	3
Subscriber Number	4
Alphanumeric	5
Abbreviated	6

NPI defines Numeric Plan Indicator

NAME	VALUE
Unknown	0
ISDN (E163/E164)	1
Data (X.121)	3
Telex (F.69)	4
Land Mobile (E.212)	6
National	8
Private	9
ERMES	10
Internet (IP)	14
WAP Client Id	18

Most used SME address examples

Mobile phone number:

address: +79171234567, TON: 1, NPI: 1

phone number must be provided in format `<country code><area code><subscriber number>`

Short number:

address: 55555, TON: 3, NPI: 0

Alphanumeric string:

address: MyService, TON: 5, NPI: 0

Enquire Link

@Inetlab.SMPP.PDU.EnquireLink is SMPP command that allows to check communication between ESME and SMSC.

To enable periodically link check, you need to set following property:

```
    _client.EnquireLinkInterval = TimeSpan.FromSeconds(30);
```

@Inetlab.SMPP.SmppClientBase.EnquireLinkInterval defines an idle interval after last received PDU. EnquireLink command won't be sent when client and server are sending PDUs.

Message Composer: How to combine concatenated messages

SMS message with long text must be split to small parts (segments). In GSM Standard maximal length of the one short message is 140 bytes. Inetlab.SMPP library provides an ability to combine all parts back into full message text. This can be done with @Inetlab.SMPP.Common.MessageComposer class.

@Inetlab.SMPP.Common.MessageComposer supports all types of PDUs: @Inetlab.SMPP.PDU.SubmitSm, @Inetlab.SMPP.PDU.SubmitMulti, @Inetlab.SMPP.PDU.DeliverSm. You should invoke @Inetlab.SMPP.Common.MessageComposer.AddMessage``1(``0) method in each event handler for received PDU. @Inetlab.SMPP.Common.MessageComposer saves PDU in memory and waits for last segment of the message text and raises @Inetlab.SMPP.Common.MessageComposer.evFullMessageReceived event. When PDU has no concatenation parameters this event will be raised right after calling @Inetlab.SMPP.Common.MessageComposer.AddMessage``1(``0) method.

When @Inetlab.SMPP.Common.MessageComposer didn't receive last segment for a long time it raises @Inetlab.SMPP.Common.MessageComposer.evFullMessageTimeout event. Default timeout is 60 secodns.

```
private readonly SmppClient _client = new SmppClient();
private readonly MessageComposer _composer = new MessageComposer();

public MessageComposerSample()
{
    _client.evDeliverSm += client_evDeliverSm;

    _composer.evFullMessageReceived += OnFullMessageReceived;
    _composer.evFullMessageTimeout += OnFullMessageTimedout;
}

private void client_evDeliverSm(object sender, DeliverSm data)
{
    _composer.AddMessage(data);
}

private void OnFullMessageTimedout(object sender, MessageEventHandlerArgs args)
{
    DeliverSm pdu = args.GetFirst<DeliverSm>();
    _log.Info(string.Format("Incomplete message received from {0}", pdu.SourceAddress));
}

private void OnFullMessageReceived(object sender, MessageEventHandlerArgs args)
{
    DeliverSm pdu = args.GetFirst<DeliverSm>();
    _log.Info(string.Format("Full message received from {0}: {1}", pdu.SourceAddress, args.Text));
}
```

@Inetlab.SMPP.Common.MessageComposer provides also methods for detecting last segment and getting full message:

```
private void client_evDeliverSmInline(object sender, DeliverSm data)
{
    _composer.AddMessage(data);
    if (_composer.IsLastSegment(data))
    {
       string receivedText = _composer.GetFullMessage(data);
    }
}
```

Mapping DataCodings to .NET Encoding

For each @Inetlab.SMPP.SmppClient instance you can define which Encoding will be used for specified @Inetlab.SMPP.Common.DataCodings.

```
//Set GSM Packed Encoding for data_coding Latin1 (0x3)
client.MapEncoding(DataCodings.Latin1, new Inetlab.SMPP.Encodings.GSMPackedEncoding());
```

By default @Inetlab.SMPP.SmppClient has following @Inetlab.SMPP.Common.DataCodings to Encoding mappings:

```
MapEncoding(DataCodings.Default, new Encodings.GSMEncoding());

MapEncoding(DataCodings.Class0FlashMessage, new Encodings.GSMEncoding());
MapEncoding(DataCodings.Class1MEMessage, new Encodings.GSMEncoding());
MapEncoding(DataCodings.Class2SIMMessage, new Encodings.GSMEncoding());
MapEncoding(DataCodings.Class3TEMessage, new Encodings.GSMEncoding());

MapEncoding(DataCodings.Class0, new Encodings.GSMEncoding());
MapEncoding(DataCodings.Class1, new Encodings.GSMEncoding());
MapEncoding(DataCodings.Class2, new Encodings.GSMEncoding());
MapEncoding(DataCodings.Class3, new Encodings.GSMEncoding());


MapEncoding(DataCodings.UCS2, Encoding.BigEndianUnicode);
MapEncoding(DataCodings.Class1MEMessageUCS2, Encoding.BigEndianUnicode);
MapEncoding(DataCodings.Class2SIMMessageUCS2, Encoding.BigEndianUnicode);
MapEncoding(DataCodings.Class3TEMessageUCS2, Encoding.BigEndianUnicode);
MapEncoding(DataCodings.UnicodeFlashSMS, Encoding.BigEndianUnicode);
```

Note

Before change mapping settings please clarify with SMPP provider which encoding is expected (character set) for @Inetlab.SMPP.Common.DataCodings value .

National Language tables

These tables allow to use different character sets in SMS messages. You can choose a language by adding User Data Header. There is ability to replace standard GSM 7 bit default alphabet table for whole text (*Locking shift table*) or only extension table (*Single shift table*). Code bellow shows abilities how you can specify desired character set:

```
Client.Submit(SMS.ForSubmit()
                .Text(text).From(sourceAddress).To(phone)
                .NationalLanguageSingleShift(NationalLanguage.Turkish)
                );

Client.Submit(SMS.ForSubmit()
                .Text(text).From(sourceAddress).To(phone)
                .NationalLanguageLockingShift(NationalLanguage.Spanish)
                );
```

or

```
submitSm.UserDataPdu.Headers.Add(new NationalLanguageSingleShift(NationalLanguage.Spanish));

submitSm.UserDataPdu.Headers.Add(new NationalLanguageLockingShift(NationalLanguage.Turkish));
```

Library is also able to detect national language User Data Header in received PDU and show text with correct character set in the value of property @Inetlab.SMPP.PDU.SubmitSm.MessageText.

Links

- GSM 03.38
- National language shift tables
- Data Coding Scheme

SSL/TLS Connection

Inetlab.SMPP library supports SSL connection between client and server.

For @Inetlab.SMPP.SmppServer class you can set server certificate and supported SSL/TLS protocols.

For @Inetlab.SMPP.SmppClient class you can specify supported SSL/TLS protocols and optionally client certificate for authentification.

```
using (SmppServer server = new SmppServer(new IPEndPoint(IPAddress.Any, 7777)))
{
    server.EnabledSslProtocols = SslProtocols.Tls12;
    server.ServerCertificate = new X509Certificate2("server_certificate.p12", "cert_password");

    server.Start();

    server.evClientConnected += (sender, client) =>
    {
        var clientCertificate = client.ClientCertificate;
        //You can validate client certificate and disconnect if it is not valid.
    };

    using (SmppClient client = new SmppClient())
    {
        client.EnabledSslProtocols = SslProtocols.Tls12;
        //if required you can be authenticated with client certificate
        client.ClientCertificates.Add(new X509Certificate2("client_certificate.p12", "cert_password"));

        if (await client.Connect("localhost", 7777))
        {
            BindResp bindResp = await client.Bind("username", "password");

            if (bindResp.Header.Status == CommandStatus.ESME_ROK)
            {
                var submitResp = await client.Submit(
                    SMS.ForSubmit()
                        .From("111")
                        .To("436641234567")
                        .Coding(DataCodings.UCS2)
                        .Text("Hello World!"));

                if (submitResp.All(x => x.Header.Status == CommandStatus.ESME_ROK))
                {
                    client.Logger.Info("Message has been sent.");
                }
            }

            await client.Disconnect();
        }
    }
}
```

Logging and Troubleshooting

Wireshark

The best way to analyze SMPP Protocol is to capture network traffic with **Wireshark** tool http://www.wireshark.org/.

SMPP related Wiki article: https://wiki.wireshark.org/SMPP

Internal Logger

Inetlab.SMPP library provides build-in logging functionality based on @Inetlab.SMPP.Logging.ILog and @Inetlab.SMPP.Logging.ILogFactory interfaces. You can implement this interface with any kind of logging framework for your solution.

- NLog
- Log4Net

Library provides @Inetlab.SMPP.Logging.ConsoleLogFactory and @Inetlab.SMPP.Logging.FileLogFactory classes.

When application starts you need to register global @Inetlab.SMPP.Logging.ILogFactory for the library.

```
   LogManager.SetLoggerFactory(new ConsoleLogFactory(LogLevel.Info));
```

or you can set @Inetlab.SMPP.SmppClientBase.Logger property when you create instances of @Inetlab.SMPP.SmppClient, @Inetlab.SMPP.SmppServerClient or @Inetlab.SMPP.SmppServer

```
client.Logger = new ConsoleLogger("MyClient", LogLevel.Verbose);
```

Library writes received and sent packet bytes in log when you enable Verbose log level. It can help us to analyze SMPP packets transfered between client and server.

Implementation example for @Inetlab.SMPP.Logging.ILog and @Inetlab.SMPP.Logging.ILogFactory interfaces:

```
public class ConsoleLogFactory : ILogFactory
{
    private LogLevel _minLevel;

    public ConsoleLogFactory( LogLevel minLevel)
    {
        _minLevel = minLevel;
    }

    public ILog GetLogger(string loggerName)
    {
        return new ConsoleLogger(loggerName, _minLevel);
    }
}

public class ConsoleLogger : ILog
{
    private readonly LogLevel _minLevel;

    public string Name { get; private set; }

    public ConsoleLogger(string loggerName, LogLevel minLevel)
    {
        Name = loggerName;
        _minLevel = minLevel;
    }
```

```
    public bool IsEnabled(LogLevel level)
    {
        return level >= _minLevel;
    }

    public void Write(LogLevel level, string message, Exception ex, params object[] args)
    {
        if (level < _minLevel) return;

        int threadId = Environment.CurrentManagedThreadId;

        string text = message;



        StringBuilder sb = new StringBuilder();
        sb.AppendFormat("{0:dd.MM.yyyy HH:mm:ss}:{1}:{2,3}: ({3}) ", DateTime.Now, GetLevelString(level),
threadId, Name);
        sb.AppendFormat(message, args);
        if (ex != null)
        {
            sb.Append(" Exception: ");
            sb.Append(ex.ToString());
        }


        Console.WriteLine(sb.ToString());

    }



    private string GetLevelString(LogLevel level)
    {
        switch (level)
        {
            case LogLevel.Fatal:
                return "FATAL";

            case LogLevel.Error:
                return "ERROR";

            case LogLevel.Warning:
                return "WARN ";

            case LogLevel.Info:
                return "INFO ";

            case LogLevel.Debug:
                return "DEBUG";

            case LogLevel.Verbose:
                return "TRACE";

        }

        return "";
    }



}
```

Special events

You can also use special events in base class @Inetlab.SMPP.SmppClientBase for tracking PDUs:

- with event @Inetlab.SMPP.SmppClientBase.evPduReceiving you can monitor all incoming PDUs.
- event @Inetlab.SMPP.SmppClientBase.evPduSending is invoked before sending PDU to network.

Metrics

To monitor @Inetlab.SMPP.SmppClient or @Inetlab.SMPP.SmppServerClient performance you can use send and receive queue metrics. @Inetlab.SMPP.SmppClientBase.Queue propery of type @Inetlab.SMPP.Common.QueueState provides following parameters:

PROPERTY NAME	DESCRIPTION
@Inetlab.SMPP.Common.QueueState.SendCount	A number of PDUs that stay in the send queue before sending to network
@Inetlab.SMPP.Common.QueueState.ReceiveCount	A number of PDUs that stay in the receive queue and wait for being processed with application event handlers.
@Inetlab.SMPP.Common.QueueState.ReceiveWorkersCount	A number of worker threads that process PDUs from receive queue and invoke event handlers in the application
@Inetlab.SMPP.Common.QueueState.IncompleteRequests	A number of request that didn't receive their response

Performance

Production

The speed ultimately is determined by

- how fast you can prepare the messages
- network bandwidth
- performance on the remote side (SMPP Server)
- how fast you can process responses

On good tuned production systems you can reach 500 messages/seconds.

Tuning

You can try to play with following optimization parameters:

Change number of threads that process received messages. Dafault is 3.

```
client.WorkerThreads = 10;
```

Change receive or send buffer size for the TCP socket

```
client.ReceiveBufferSize = 32 * 1024 * 1024;
client.SendBufferSize = 32 * 1024 * 1024;
```

A larger buffer size might delay the recognition of connection difficulties. Consider increasing the buffer size if you are using a high bandwidth, high latency connection (such as a satellite broadband provider.)

Local Test

Inetlab.SMPP performance on local machine with disabled logging shows following results

```
Performance: 20356 m/s
```

Following code demonstrate this

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net;
using System.Threading.Tasks;
using Inetlab.SMPP;
using Inetlab.SMPP.Common;
using Inetlab.SMPP.Logging;

namespace TestLocalPerformance
{
    class Program
    {
        static void Main(string[] args)
        {

            LogManager.SetLoggerFactory(new ConsoleLogFactory(LogLevel.Info));

            StartApp().ConfigureAwait(false);

            Console.ReadLine();
```

```
        }

        public static async Task StartApp()
        {

            using (SmppServer server = new SmppServer(new IPEndPoint(IPAddress.Any, 7777)))
            {
                server.evClientBind += (sender, client, data) => { /*accept all*/ };
                server.evClientSubmitSm += (sender, client, data) => {/*receive all*/ };
                server.Start();

                using (SmppClient client = new SmppClient())
                {
                    await client.Connect("localhost", 7777);

                    await client.Bind("username", "password");

                    Console.WriteLine("Performance: " + await RunTest(client, 50000) + " m/s");
                }
            }
        }

        public static async Task<int> RunTest(SmppClient client, int messagesNumber)
        {

            List<Task> tasks = new List<Task>();

            Stopwatch watch = Stopwatch.StartNew();

            for (int i = 0; i < messagesNumber; i++)
            {
                tasks.Add(client.Submit(
                    SMS.ForSubmit()
                        .From("111")
                        .To("222")
                        .Coding(DataCodings.UCS2)
                        .Text("test")));
            }

            await Task.WhenAll(tasks);

            watch.Stop();

            return Convert.ToInt32(messagesNumber / watch.Elapsed.TotalSeconds);

        }
    }
}
```

How to install license file

After purchase of developer license you should receive Inetlab.SMPP.license file per E-Mail. Also you can always generate a license file with your InetLab Account. It allows for Source Code license owners to add and update NuGet package in their projects.

From Embedded Resources

Add this file into the root of project where you have a reference on Inetlab.SMPP.dll. Change "Build Action" of the file to "Embedded Resource".

Set license before using Inetlab.SMPP classes in your code:

```
Inetlab.SMPP.LicenseManager.SetLicense(this.GetType().Assembly.GetManifestResourceStream(this.GetType(),
"Inetlab.SMPP.license"));
```

From string variable

Open your license file with any text editor and copy and paste the content into the string variable in your code.

Set license before using Inetlab.SMPP classes in your code:

```
            string licenseContent = @"
-----BEGIN INETLAB LICENSE------
EBAXG23FO4BR23LJMNAGCZLMFZQXG23F
GY4DEMJTG43DGBMAQFD4DPHQ2UEANACB
BY5I4D6XBCAACRUJXKZKI7K2N76CTXSC
NDJP2CIM4KHV5V7VCXT75R4XRDSLZZQS
2NKD6JHCIG4PNPUN5A7G4KRZQSZSNL44
NB2LTYRP5FATRVKCHD26FC64E2TSQFX5
Q6GWNF3HVVQIE2YKOO74C4FVR6HDUGD6
FYO4DHCPCPQ2GY3WQRMOFOXOZQ======
-----END INETLAB LICENSE--------";

            Inetlab.SMPP.LicenseManager.SetLicense(licenseContent);
```

END-USER LICENSE AGREEMENT

for all versions of components Inetlab.SMPP Inetlab MM7.NET

IMPORTANT-READ CAREFULLY:

This End-User License Agreement ("LICENSE") is a legal agreement between Licensee (either an individual or a single entity) and InetLab e.U. represented by Svetlana Tsynaeva, for the software package containing this LICENSE, which includes computer software and may include associated "online" or electronic documentation ("SOFTWARE"). The SOFTWARE also includes any updates and supplements to the original SOFTWARE provided to you by InetLab e.U.. By installing, copying or otherwise using the SOFTWARE, you agree to be bound by the terms of this LICENSE. If you do not agree to all the terms of this LICENSE, do not install or use the SOFTWARE.

SOFTWARE LICENSE

Copyright laws and international copyright treaties, as well as other intellectual property laws and treaties protect the SOFTWARE. This is a license agreement and NOT an agreement for sale. InetLab e.U. continues to own the copy of the SOFTWARE contained on the disk or CD-ROM and all copies thereof.

1. LICENSE TO USE SOFTWARE.

    1. DEVELOPER LICENSE. The SOFTWARE is licensed per individual developer. You may make copies on more than one computer, as long as the use of the SOFTWARE is by the same developer. Each developer working with the SOFTWARE must purchase a copy of the component for his/her own development needs.
    2. SOURCE CODE. Licensee has no right of access to the source code of the SOFTWARE, unless he purchases source code separately as defined in section 1.3.
    3. SOURCE CODE LICENSE. Licensee who purchases source code separately and in addition to previously purchased license(s) has right to use the source code for debugging, bug fixing and any other modifications. Under no circumstances may the source code be used in whole or in part, as the basis for creating a product that provides the same, or substantially the same, functionality as any InetLab e.U. product. Licensee may not distribute the source code, or any modification, enhancement, derivative work and/or extension thereto, in source code form. SOURCE CODE IS LICENSED AS IS. InetLab e.U. DOES NOT AND SHALL NOT PROVIDE ANY TECHNICAL SUPPORT FOR SOURCE CODE LICENSE.

2. DISTRIBUTION / REDISTRIBUTABLE CODE

    1. SAMPLE CODE. In addition to the LICENSE granted in Section 1, InetLab e.U. grants the Licensee the right to use and modify the source code versions of those portions of the SOFTWARE that are identified in the documentation as the Sample Code and located in the "SAMPLES" subdirectory(s) of the SOFTWARE.
    2. REDISTRIBUTABLE FILES. In addition to the LICENSE granted in Section 1, InetLab e.U. grants the Licensee a nonexclusive, royalty-free right to distribute the object code version of those portions of the SOFTWARE identified as the redistributable files ("REDISTRIBUTABLE FILES"), provided Licensee complies with the redistribution requirements.

    The following files in the SOFTWARE distribution are considered REDISTRIBUTABLE FILES under this LICENSE:

    - Inetlab.*.dll

    1. REDISTRIBUTION REQUIREMENTS. If Licensee redistributes the REDISTRIBUTABLE FILES, he/she agrees to (a) distribute the REDISTRIBUTABLE FILES in object code form only in conjunction with, and as part of her/his software application product which adds significant and primary functionality; (b) include a valid copyright notice on his/her SOFTWARE; and (c) indemnify, hold harmless, and defend InetLab e.U. from and against any claims or lawsuits, including attorney's fees, that arise or result from the use and distribution of his/her software application product.

    2. LIMITATIONS. Distribution by the Licensee of any executables, source code or other files distributed by InetLab e.U. as part of this SOFTWARE and not identified as a REDISTRIBUTABLE FILE is prohibited. Redistribution of REDISTRIBUTABLE FILES by Licensee's users without the appropriate redistribution LICENSE is prohibited.

Licensee shall not develop applications that provide an application programmable interface to the SOFTWARE. Licensee shall not develop applications that substantially duplicate the capabilities of the SOFTWARE or, in the reasonable opinion of InetLab e.U., compete with it.

Licensee MAY NOT distribute the SOFTWARE, in any format, to other users for development or compiling purposes. In particular, if Licensee creates a component/control using the SOFTWARE as a constituent component/control, Licensee MAY NOT distribute the component/control created with the SOFTWARE (in any format) to users for being used at design time and/or for development purposes.

1. ADDITIONAL RIGHTS AND LIMITATIONS

    1. RESTRICTIONS. Licensee may not alter, assign, create derivative works, decompile, disassemble, distribute, give, lease, loan, modify, rent, reverse engineer, sell, sub-license, transfer or translate in any way, by any means or any medium the SOFTWARE. Licensee will use its best efforts and take all reasonable steps to protect the SOFTWARE from unauthorized use, copying or dissemination.

    2. SUPPORT SERVICES. InetLab e.U. may provide you with support services related to the SOFTWARE ("Support Services"). Use of Support Services is governed by the policies and programs described in "online" documentation and/or in other InetLab e.U. provided materials. Any supplemental software code provided to you as part of the Support Services shall be considered part of the SOFTWARE and subject to the terms and conditions of this LICENSE. With respect to technical information you provide to InetLab e.U. as part of the Support Services, InetLab e.U. may use such information for its business purposes, including for product support and development. InetLab e.U. will not utilize such technical information in a form that personally identifies Licensee.

    3. The SOFTWARE is licensed as a single product and the software programs comprising SOFTWARE may not be separated.

    4. TERMINATION. If the SOFTWARE is used in any way not expressly and specifically permitted by this LICENSE, then the LICENSE shall immediately terminate. Upon the termination of the LICENSE, Licensee shall thereafter make no further use of the SOFTWARE, and Licensee shall return or destroy all licensed materials.

2. UPGRADES, ENHANCEMENTS AND UPDATES. From time to time, at its sole discretion, InetLab e.U. may provide enhancements, updates, or new versions of the SOFTWARE on its then standard terms and conditions thereof. This Agreement shall apply to such enhancements. Licensee is not entitled to updates or upgrades of the SOFTWARE unless such right is stated in additional agreement between Licensee and InetLab e.U.. If new version of the SOFTWARE is released within thirty (30) days from the day of purchase and the price of new version is equal or smaller than the price of purchased version of the SOFTWARE, Licensee is entitled to a new version at zero cost. Received new version shall be considered part of purchased version of the SOFTWARE and the number of licensed developers will stay the same as granted in Section 1.1.

3. COPYRIGHT. All title and intellectual property rights in and to the SOFTWARE (including but not limited to any images, photographs, animations, video, audio, music and text incorporated into the SOFTWARE) and any copies of the SOFTWARE are owned by InetLab e.U. or its suppliers. All title and intellectual property rights in and to the content which may be accessed through use of the SOFTWARE is the property of the respective content owner and may be protected by applicable copyright or other intellectual property laws and treaties. This LICENSE grants Licensee no rights to use such content. InetLab e.U. reserves all rights not expressly granted.

4. LIMITED WARRANTY. Licensee assumes all responsibility for the selection of the SOFTWARE as appropriate to achieve the results he/she intends. The SOFTWARE and documentation are not represented to be error-free. InetLab e.U. warrants that (a) the SOFTWARE shall perform substantially as described in its documentation for a period of thirty (30) days from purchase, and (b) any Support Services provided by InetLab e.U. shall be substantially as described in our accompanying materials, and our Support Team will make commercially reasonable efforts to solve any problem covered by our warranty. EXCEPT FOR THE FOREGOING LIMITED WARRANTY AND TO THE MAXIMUM EXTENT PERMITTED BY LAW, THE SOFTWARE IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT AND OF FITNESS FOR A PARTICULAR PURPOSE.

5. CUSTOMER REMEDIES. InetLab e.U. entire liability and Licensee's exclusive remedy shall be, at InetLab e.U. option, either (a) return of the price paid or (b) repair or replacement of the SOFTWARE that does not meet InetLab e.U. Limited Warranty and which is returned to InetLab e.U. with a copy of Licensee's receipt. SOFTWARE purchased other than directly from InetLab e.U. shall be returned to the place where it was purchased. This Limited Warranty is void if failure of the SOFTWARE has resulted from accident, abuse, or misapplication. Any replacement SOFTWARE will be warranted for the remainder of the original warranty period or remainder of the thirty (30) days from the day of purchase, whichever is longer.

6. NO LIABILITY FOR CONSEQUENTIAL DAMAGES. To the maximum extent permitted by law, in no event shall InetLab e.U. or its suppliers be liable for any special, incidental, indirect or consequential damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising out of use of or inability to use this SOFTWARE, or the failure to provide Support Services, even if InetLab e.U. or its dealer have been advised of the possibility of such damages. In any case, InetLab e.U. entire liability under any provision of this LICENSE shall be limited to the amount actually paid by the licensee for the SOFTWARE.

7. GENERAL PROVISION. Licensee shall have no right to sub-license any of the rights of this agreement, for any reason. In the event of the breach by Licensee of this Agreement, he/she shall be liable for all damages to InetLab e.U., and this Agreement shall be terminated. If any provision of this Agreement shall be deemed to be invalid, illegal, or unenforceable, the validity, legality, and enforceability of the remaining portions of this Agreement shall not be affected or impaired thereby. In the event of a legal proceeding arising out of this Agreement, the prevailing party shall be awarded all legal costs incurred.

8. TAXES AND DUTIES. Licensee shall be responsible for the payment of all taxes or duties that may now or hereafter be imposed by any authority upon this Agreement for the supply, use, or maintenance of the SOFTWARE, and if any of the foregoing taxes or duties are paid at any time by InetLab e.U., Licensee shall reimburse InetLab e.U. in full upon demand.

9. MISCELLANEOUS. This Agreement shall be governed by, construed and enforced in accordance with the laws of the Austria. Each party consents to the personal jurisdiction of the Austria and agrees to commence any legal proceedings arising out of this LICENSE shall be conducted solely in the courts located in the Austria. This is the entire agreement between you and InetLab e.U. which supersedes any prior agreement, whether written or oral, relating to this subject matter. Licensee acknowledges that he/she has read this Agreement, understands it, and agrees to be bound by its terms and conditions.