Inetlab.SMPP

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

Table of Contents

Introduction

The Inetlab SMPP library implements SMPP protocol for two-way SMS messaging over TCP/IP. It allows to communicate with the SMSC (Short Message Service Center) or SMS provider. Using the library, you can send SMS messages to customers, receive messages from mobile devices and process delivery receipts. It supports long text messages in any encoding.

This is a robust SMPP framework for building production-grade solutions. Inetlab SMPP is helpful in such tasks as:

- notifying users
- command receiving from mobile subscribers (i.e. accounts balance requests)
- creation of SMS Gateway for SMS traffic reselling
- and many other applications.

The Inetlab SMPP library is fully compliant with SMPP specifications v3.3, v3.4, v5.0 and comes with a comprehensive set of code samples. Enjoy exploring our demo applications, knowledge base and best support from our development team. Inetlab developers will review your code and even analyze your Wirshark network SMPP data logs!

SMPP Client Features

- Sending long Text messages as concatenated segments
- Sending Binary messages
- Sending Flash SMS
- Sending WAP Push
- Receiving SMS messages from mobile phones
- Intuitive SMS building with fluent interface
- Keeping connection to SMPP server alive
- Working with any language including Arabic, Chinese, Hebrew, Russian, Greek and Unicode messages support
- Reliable bulk SMS-sending at up to 500 messages per second rate
- SSL/TLS support
- and many more

SMPP Server Features

- Multiple concurrent client connections support
- Receiving SMS messages from connected clients
- Sending Concatenated Text messages
- Sending Delivery receipts
- Message status query support
- Message rate limit and throttling
- Ability to forward received messages to next SMPP server
- SSL/TLS support
- Tests availability of client with enquiry_link command
- Proxy Protocol for load-balancing support
- and many more

How to try the library

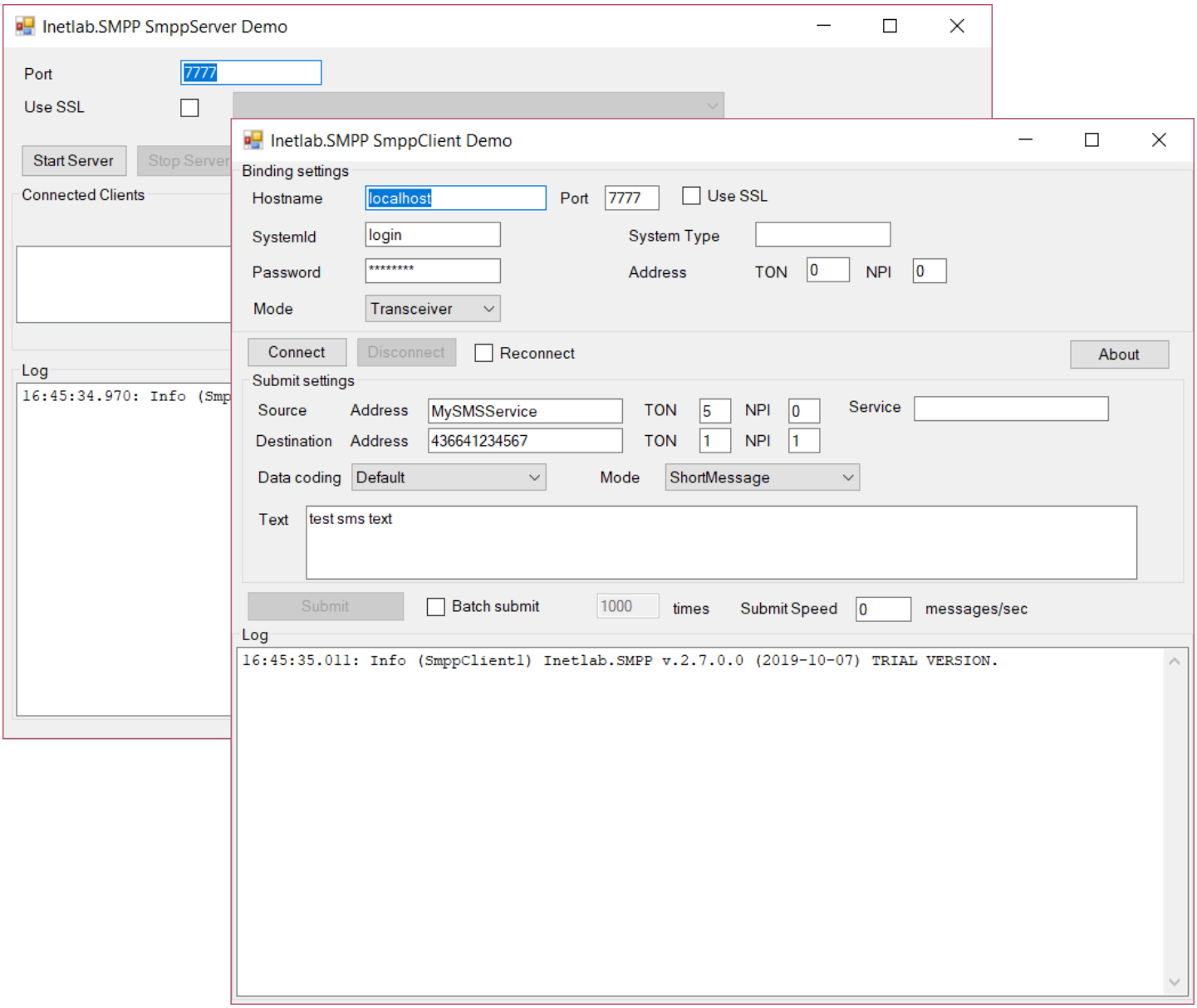
Get samples

Latest source code of the samples for Inetlab.SMPP library you can find on the link. Or you can download zip archive with all samples.

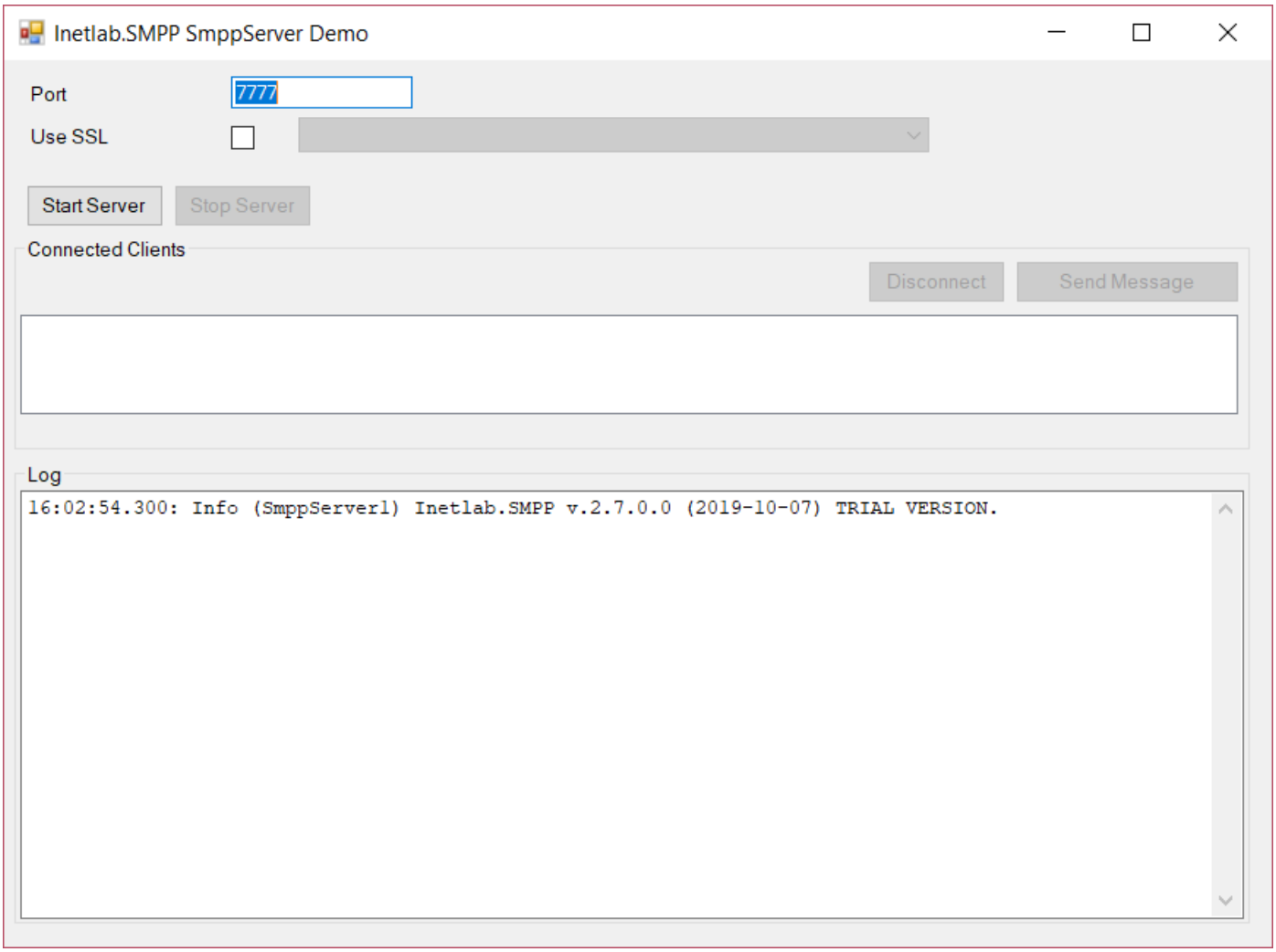
Start Demo Apps

Install Visual Studio 2017 or Visual Studio 2019 on your PC before starting the following .bat file.

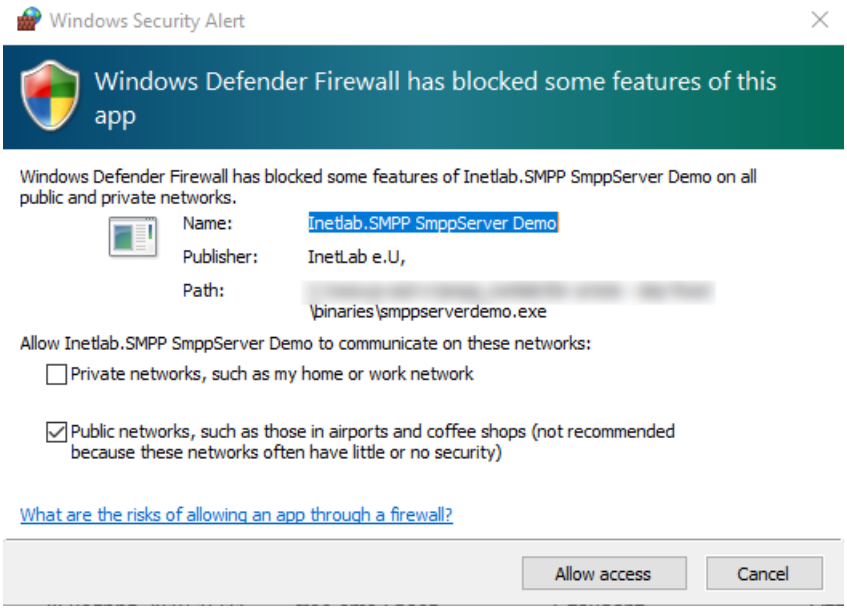
Unpack ZIP and run file run_demo.bat in “smpp-samples-master” folder. In a console window appeared you might see the question – respond with “Y” for starting samples compilation. After this (and further launches) of run_demo.bat you should see two demo-applications started: "Inetlab.SMPP SmppServer Demo" and "Inetlab.SMPP SmppClient Demo".



Press button "Start Server" in the "Inetlab.SMPP SmppServer Demo" application



You might see firewall warning "Windows Security Alert" after button "Start Server" is pressed.



For the application SmppServerDemo.exe to work correctly, you need to accept this Windows Defender Firewall request by pressing "Allow access" button.

After starting SMPP-server, the "Start Server" button will be disabled and the "Stop Server" button will become clickable. Since that moment your computer acts as an SMPP-server reachable at addresses: localhost:7777, 127.0.0.1:7777 as well as via IP-address of your PC in the local network (Ethernet or Wi-Fi) at port 7777.

Demo-program starts SMPP-server on port 7777 by default. Of course, you can type in any port number you prefer before

getting server started. Mind the server port when connecting with SMPP-client on the next steps.

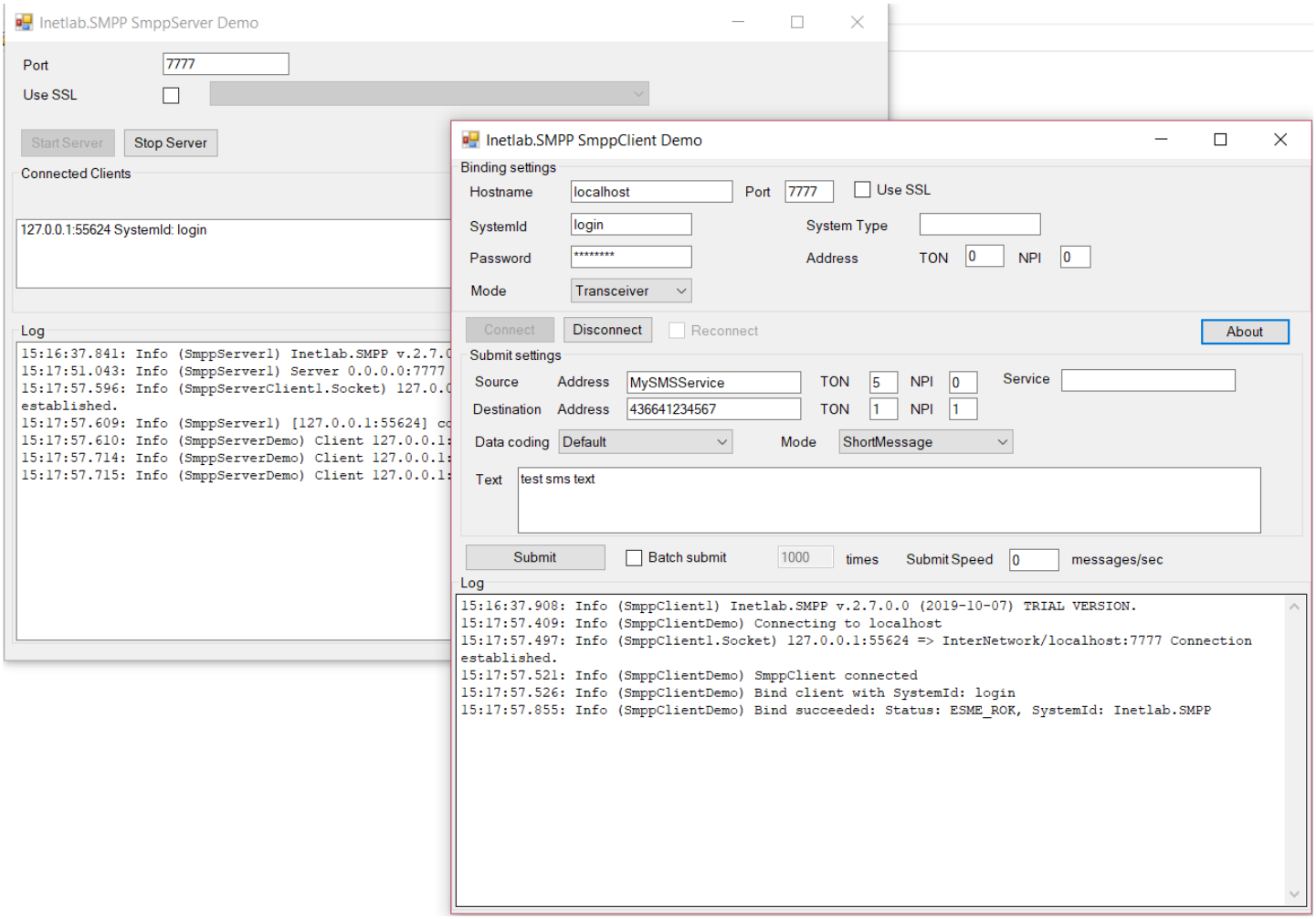
Showcases

Connect "SmppClient Demo" to "SmppServer Demo"

Demo-application "Inetlab.SMPP SmppClient Demo" already set with default server address (localhost) and port (7777) values matching default demo-server application settings. Press "Connect", to get SMPP-client connected to the SMPP-server implemented by "Inetlab.SMPP SmppServer Demo".

Application "Inetlab.SMPP SmppServer Demo" now should have a line of text in the field "Connected Clients" showing "SystemId" of the SMPP-client connected. In our case, it should be "SystemId: login".

Now Log-fields of both applications should contain some lines of debug information related to the connection established.



There should be a record "Bind succeeded: Status: ESME_ROK, SystemId: Inetlab.SMPP" in the Log-field of "Inetlab.SMPP SmppClient Demo" window.

Submit batch messages from client to server

Let's make a batch sending of messages from client to server. Check the checkbox "Batch submit" next to "Submit" button in "Inetlab.SMPP SmppClient Demo" window. There is a preset value of 1000 for sending 1000 test messages in a batch. Default Submit speed is "0" – which means there is no delay performed between each message submission. Press "Submit" button to start.

The new record saying "Submit message batch. Count: 1000. Text: test sms text." should appear in the LOG-field of "Inetlab.SMPP SmppClient Demo" window. It should be followed by records "Batch sending completed. Submitted: 1000, Elapsed: 147 ms, Performance: 6802.721 m/s" (your digits may vary). It means the SMPP-client have just sent 1000 messages to the server.

In Log-field of server application window you will see plenty of similar records (a thousand in fact):

... [timestamp]: Info (SmppServerDemo) Client 127.0.0.1:55624 sends message From:MySMSService, To:436641234567, Text: test sms text[TRIAL] [timestamp]: Info (SmppServerDemo) SMS Received: test sms text[TRIAL] ...

This example does not use any kind of looping in a code. It just prepares the collection of messages (1000 of identical messages in this example) and sends to the server with a single command. The code performs sending with single asynchronous operation. Collection may contain messages with varying recipient numbers and message bodies. The software automatically collects all related server responses and returns them as a single collection. "Message speed" parameter sets the delay between messages. It is useful to avoid "throttling" (throttling error) – special kind of an SMPP-server restriction applied to an SMPP-clients sending messages too fast. You can read more about throttling in the article Throttling error.

Submit Cyrillic text message in UCS2 encoding from client to server

Let's put some text containing Cyrillic symbols into "Text" field of "Inetlab.SMPP SmppClient Demo"– for example "это тестовое sms". Choose UCS2 in dropdown menu "Data coding". Press "Submit". There should be a new record in the Log-field of "Inetlab.SMPP SmppServer Demo" window:

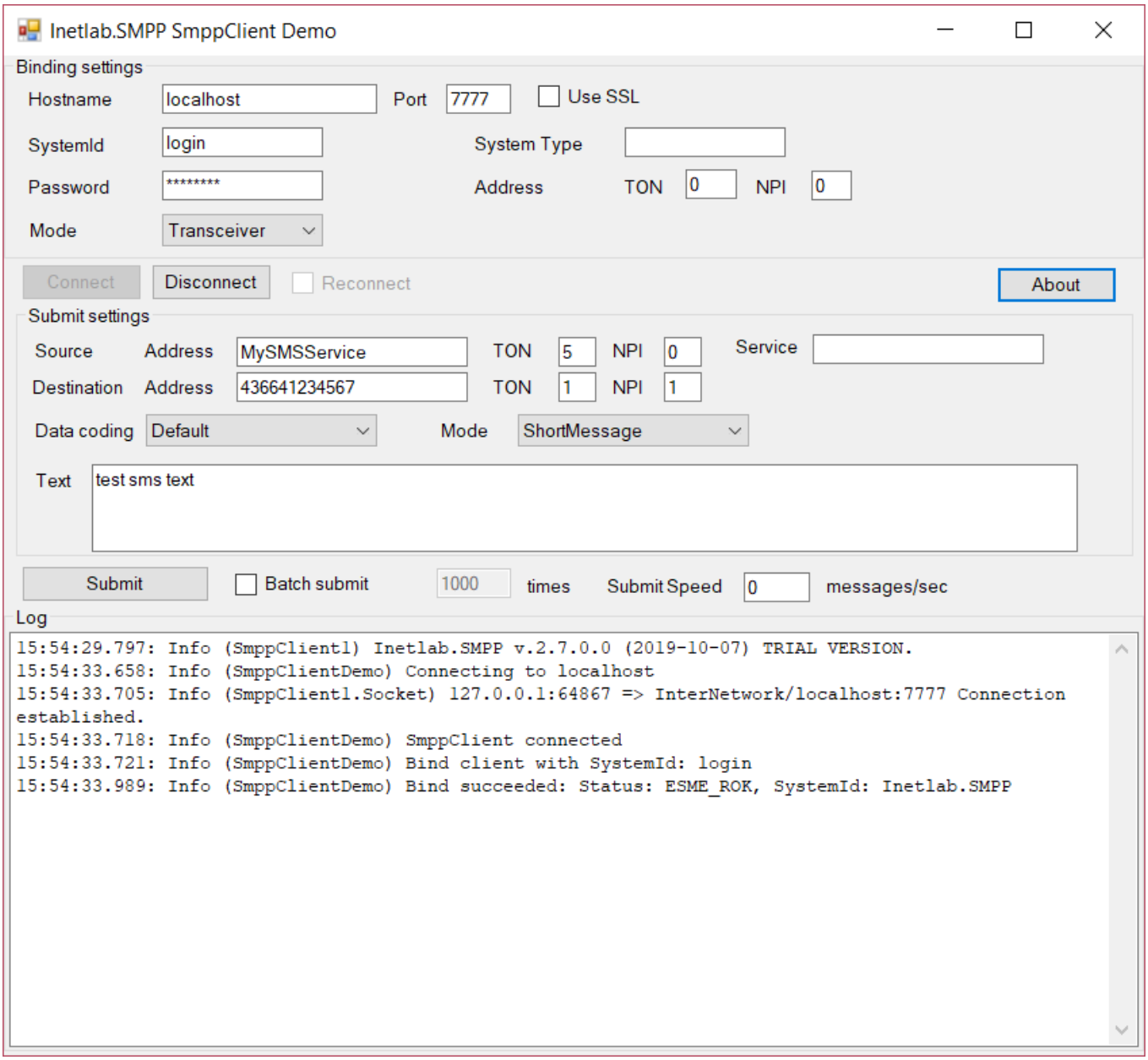
[timestamp]: Info (SmppServerDemo) Client 127.0.0.1:53233 sends message From:MySMSService, To:436641234567, Text: это тестовое sms[TRIAL] [timestamp]: Info (SmppServerDemo) SMS Received: это тестовое sms[TRIAL]

Message successfully delivered to the SMPP-server. Please note, if you keep the default value in "Data coding" dropdown, you will see all Cyrillic symbols arrived to server as question marks.

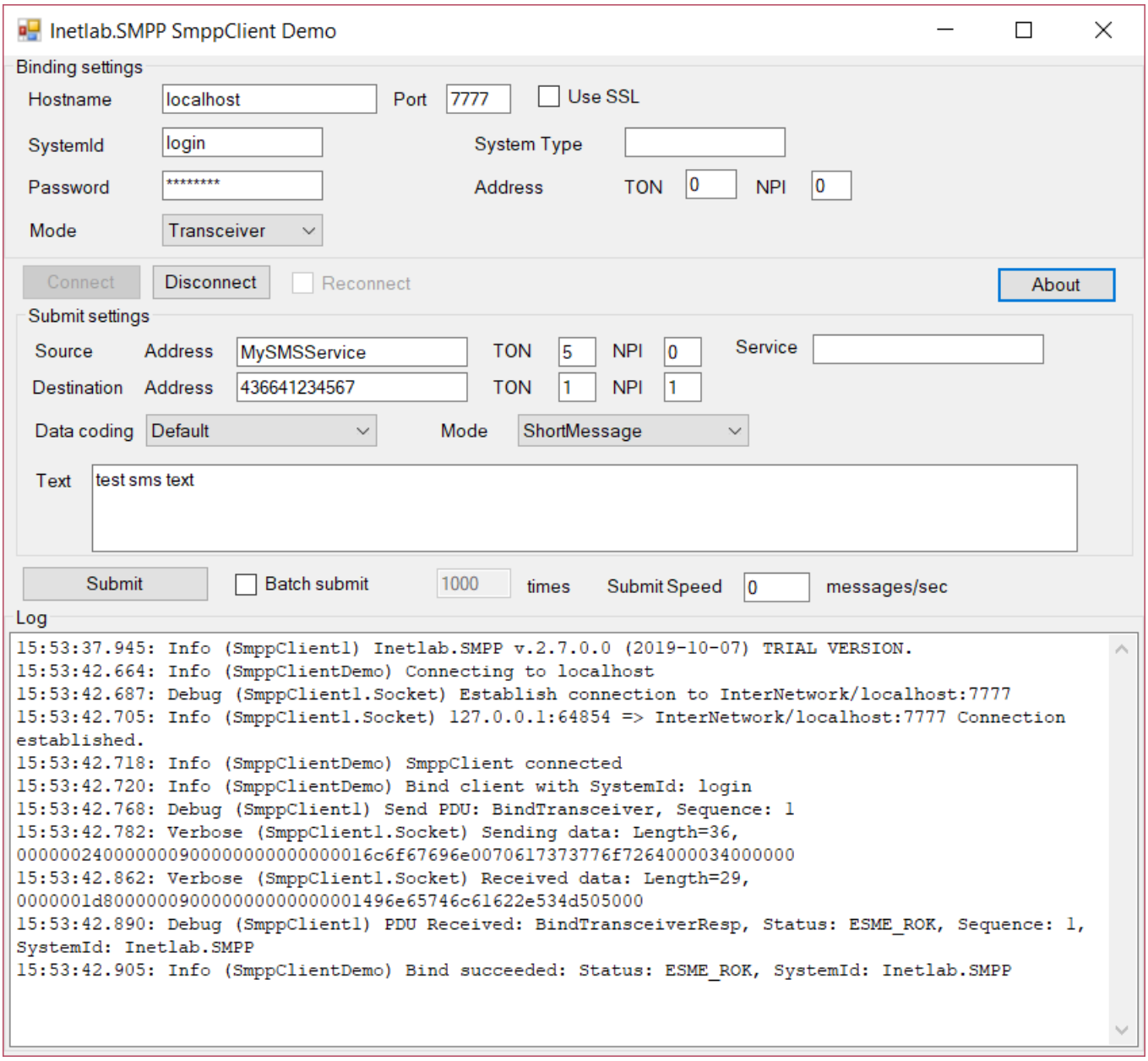
You can read more about text encoding in the article Map Encoding.

What detailed log looks like?

Log-fields of client and server are populated with new information thanks to a Logger embedded in the Inetlab.SMPP library. The embedded logger creates text records reflecting the meaning of current operations automatically. The default logging level is "Info". For example, this is how Log-field of SMPP-client looks like when launched and connected to an SMPP-server (log level "Info").



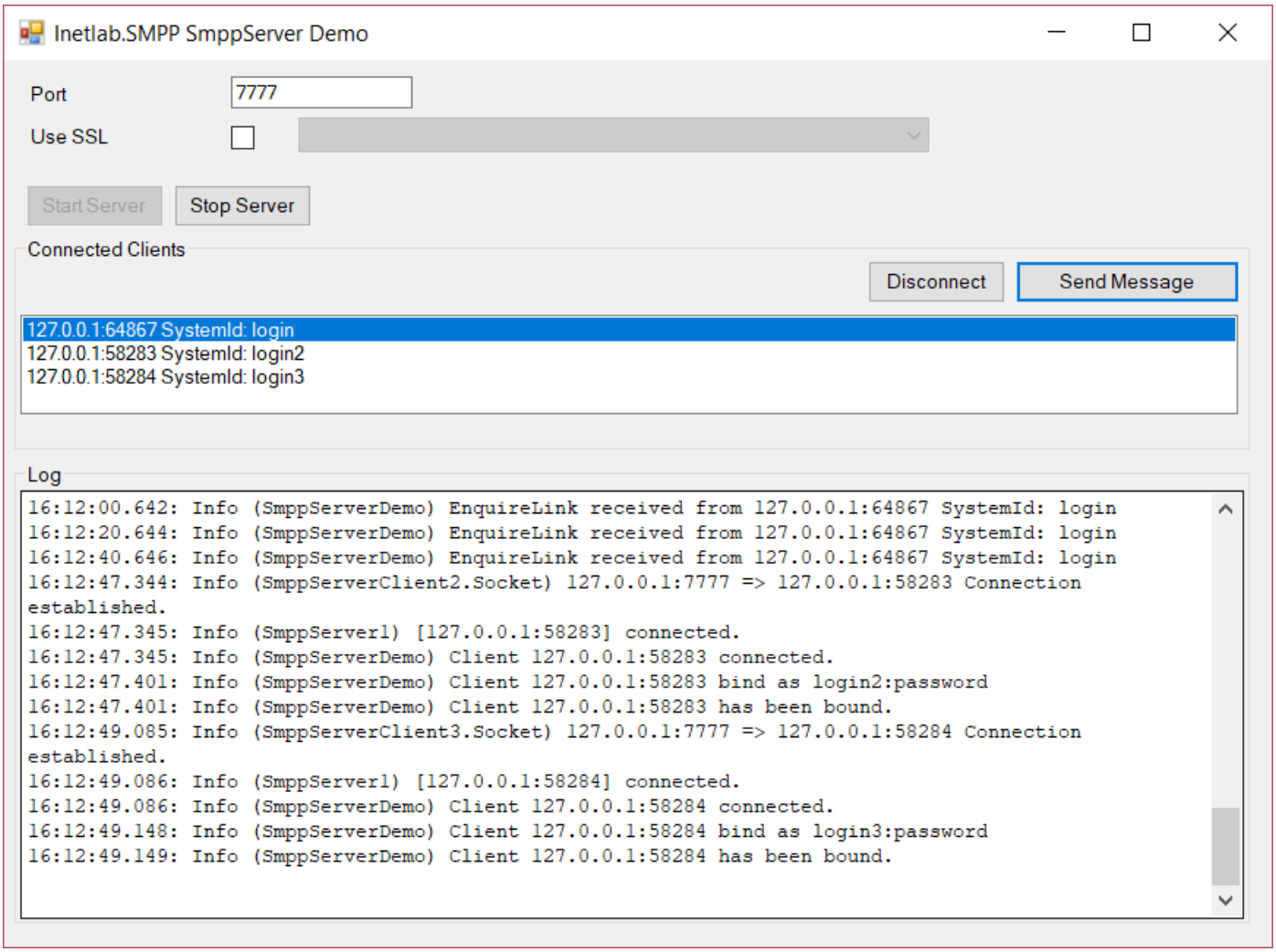
To change logging level it is necessary to change logging settings in the source code of SMPP-client and compile the project again. For example, by setting "Verbose" level in logger settings (showing much more technical details when “Info”) you will get more information in the logger output. Log-field will have additional records marked as “Verbose” and “Debug”.



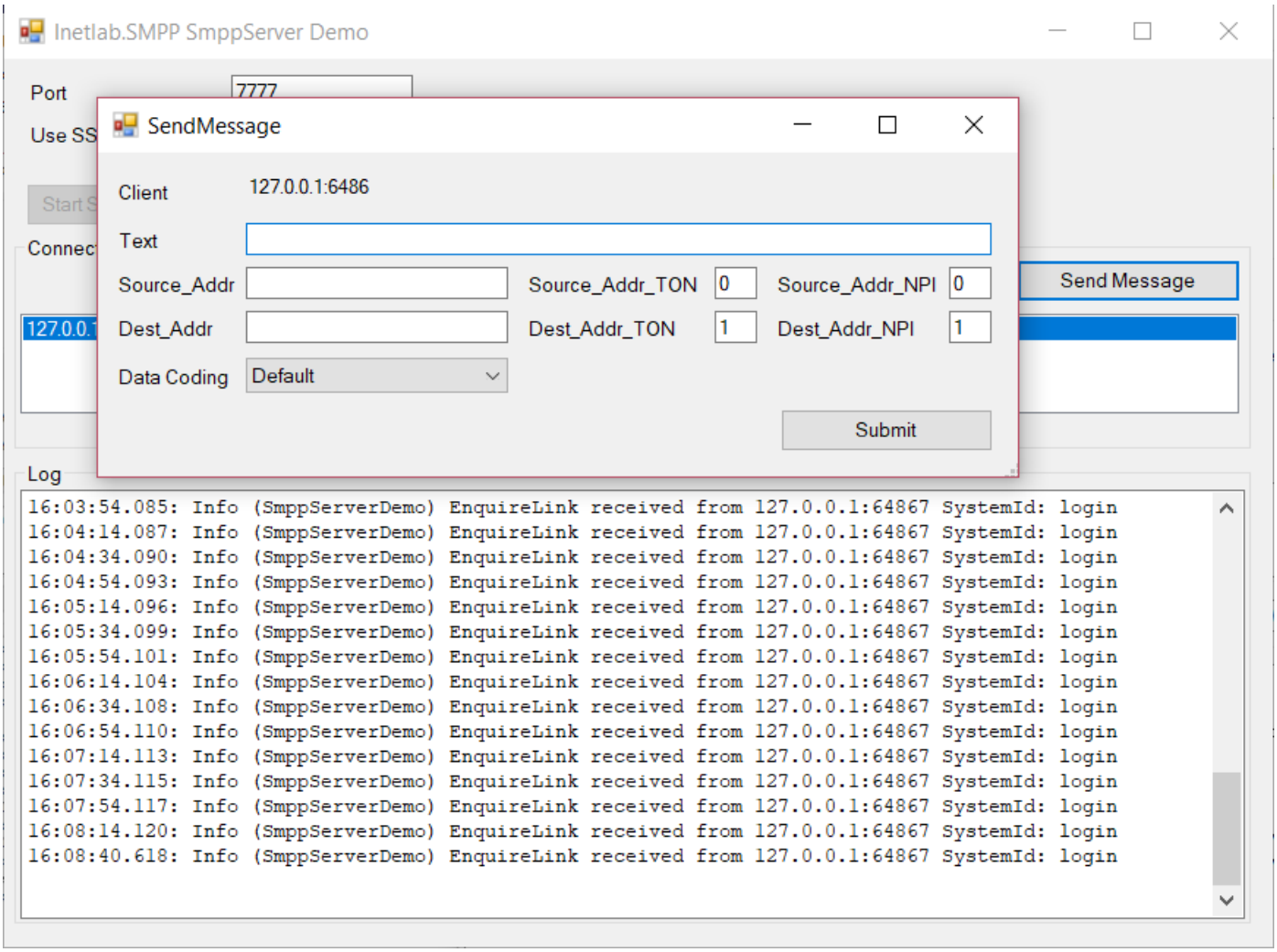
You can read more logging and logging levels in the article Creating local and global Logger.

Send a message from server to client selected

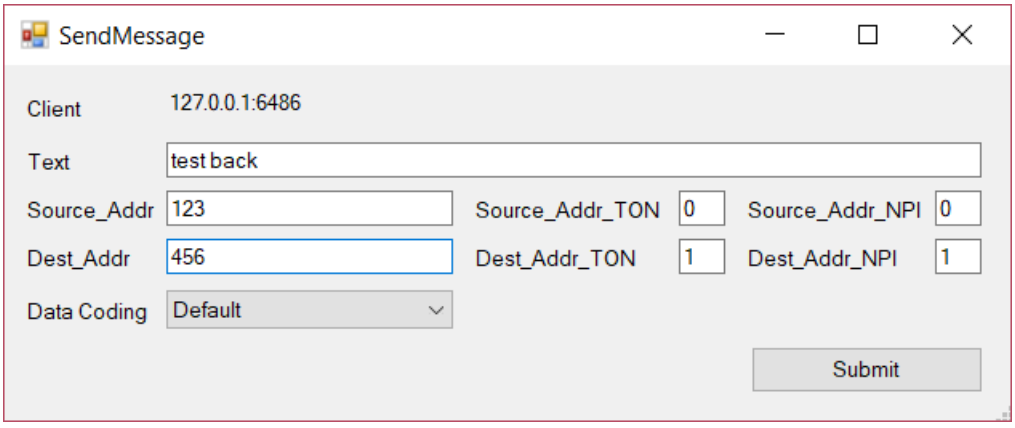
There is a list of all SMPP-clients connected and their respective logins in the “Connected Clients” field of "Inetlab.SMPP SmppServer Demo" window.



If you click a line containing client login (for example, "SystemId: login") and the press "Send Message" button at the right side of the window, you will be able to message the SMPP-client by filling a form.



Let's fill the form with arbitrary information:



After filling all fields and pressing "Submit" button the message will be sent to the SMPP-client. By having a look at the Log-field of SMPP-client application, we can confirm if the message received. The similar record should appear:

... [timestamp]: Info (SmppClientDemo) DeliverSm received : Sequence: 1, SourceAddress: 123, Coding: Default, Text: test back[TRIAL] ...

As next step you can begin to create your own SMPP Client or SMPP Server application.

SMPP Client

Creating SMPP-client and Connect

You need to know the address and port of SMPP-server to establish connection to it. Let us create an instance of an SMPP-client and proceed with asynchronous method <xref:Inetlab.SMPP.SmppClient.ConnectAsync*> using server data as arguments.

```
SmppClient _client = new SmppClient();
bool connected = await _client.ConnectAsync("localhost", 7777);
```

This example illustrates how an SMPP-client connects local SMPP-server available at "localhost", port 7777. It is described in according article how to create local SMPP-server.

Run the code inside the asynchronous method. After execution, the **connected** variable will receive information about operation result (Boolean true/false).

In most cases the successful connect is followed by bind operation.

Authentication (Bind)

The login and password issued by an SMPP-provider is required to pass authentication at SMPP-server.

Authentication to be performed after connection established successfully. Login and password transmitted to server with asynchronous method <xref:Inetlab.SMPP.SmppClient.BindAsync*>. In <xref:Inetlab.SMPP.SmppClient.BindAsync*> method you can also specify ConnectionMode. When calling <xref:Inetlab.SMPP.SmppClient.BindAsync*> the third parameter (Connection Mode) is optional and, if not specified, by default it is <xref:Inetlab.SMPP.Common.ConnectionMode.Transceiver>.

```
if (await _client.ConnectAsync("localhost", 7777))
{
    BindResp bindResp = await _client.BindAsync("Login", "Password", ConnectionMode.Transceiver);
}
```

Calls to methods <xref:Inetlab.SMPP.SmppClient.ConnectAsync*> and <xref:Inetlab.SMPP.SmppClient.BindAsync*> are to be accompanied with "await" operator. It guarantees getting to **Bind** operation only after **Connect** was successful.

In the following example, variable **bindResp** contains the result of <xref:Inetlab.SMPP.SmppClient.BindAsync*> execution, in particular the server response and status.

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

<xref:Inetlab.SMPP.Common.CommandStatus.ESME_ROK> status confirms successful execution of authentication command and means you can proceed with sending and/or receiving messages. The <xref:Inetlab.SMPP.SmppClient> object will change its <xref:Inetlab.SMPP.SmppClientBase.Status> to <xref:Inetlab.SMPP.Common.ConnectionStatus.Bound>.

Read more about statuses in the article Sending Commands and Getting Responses.

Connection recovery

Connection recovery can be activated with <xref:Inetlab.SMPP.SmppClient.ConnectionRecovery> property.

```
SmppClient _client = new SmppClient();
_client.ConnectionRecovery = true;
```

This works only after first successful bind. <xref:Inetlab.SMPP.SmppClient> triggers following events by connection recovery:

- event <xref:Inetlab.SMPP.SmppClient.evRecoverySucceeded> - when bind was successful.
- event <xref:Inetlab.SMPP.SmppClientBase.evDisconnected> - when bind was failed.

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

For the first successful **bind** you need to write a Connect method that repeats Connect and Bind until the status <xref:Inetlab.SMPP.Common.CommandStatus.ESME_ROK> in <xref:Inetlab.SMPP.PDU.BindResp> is received. The extension method <xref:Inetlab.SMPP.SmppClientExtensions.RetryUntilBindAsync(Inetlab.SMPP.SmppClient,Inetlab.SMPP.Common.SmppClientConnectionOptions,System.String,System.String,System.TimeSpan,Inetlab.SMPP.Common.ConnectionMode,System.Int32,System.Threading.CancellationToken)> can help you.

The delay time between recovery attempts can be changed with the property <xref:Inetlab.SMPP.SmppClient.ConnectionRecoveryDelay>. Default is 2 minutes.

Create and send messages

There are several ways to create a message. The most convenient way is by using helper class <xref:Inetlab.SMPP.SMS>:

```
IList<SubmitSm> pduList = SMS.ForSubmit().From("111").To("79171234567").Text("Hello World!").Create(_client);
```

This example will produce a collection **pduList** containing single short message. If the message is longer when 140 octets, it will be automatically split to parts. All message parts are also placed into a collection **IList<<xref:Inetlab.SMPP.Common.CommandSet.SubmitSm>>**. The mobile phone automatically concatenates received message parts into a single longer message.

The library also provides a way to create `SumitSm PDU` manually:

```
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;
}
```

This method does not have support for long messages and splitting. However, when you need to create <xref:Inetlab.SMPP.PDU> messages and set properties not supported by <xref:Inetlab.SMPP.SMS> class this method is very useful.

Actual message transmission is performed by calling method <xref:Inetlab.SMPP.SmppClient.SubmitAsync*> and passing either an argument of `SubmitSm PDU` either arrays/collections using method overloads.

```
IEnumerable<SubmitSmResp> responses = await _client.SubmitAsync(pduList);
```

The <xref:Inetlab.SMPP.SmppClient.SubmitAsync*> method supports batch sending. It is possible to send **pduList** containing thousands of PDUs by a single call to <xref:Inetlab.SMPP.SmppClient.SubmitAsync*> method. The method will return results after <xref:Inetlab.SMPP.SmppClient> have received server responses for all <xref:Inetlab.SMPP.PDU>'s sent.

Please note, the order of <xref:Inetlab.SMPP.PDU.SubmitSmResp> in **response** collection may not match the order of PDUs in **pduList** collection. The relation between commands sent and server responses may be established by `resp.Header.Sequence` property.

Successful processing of <xref:Inetlab.SMPP.PDU.SubmitSm> on server side produces server response containing status `response.Header.Status = ESME_ROK`.

It is possible that several <xref:Inetlab.SMPP.PDU.SubmitSmResp> will produce a response with error status:

- <xref:Inetlab.SMPP.Common.CommandStatus.SMPPCLIENT_NOCONN> - connection failed during sending attempt.
- <xref:Inetlab.SMPP.Common.CommandStatus.SMPPCLIENT_UNBOUND> - you are probably trying to send commands via <xref:Inetlab.SMPP.SmppClient> without authentication (Bind).
- <xref:Inetlab.SMPP.Common.CommandStatus.SMPPCLIENT_RCVTIMEOUT> - response to request is not arrived during certain time.

Receive messages

An SMPP-server sends SMS to SMPP-client by using command <xref:Inetlab.SMPP.Common.CommandSet.DeliverSm>. It may contain inbound SMS as well as delivery report.

There is an event <xref:Inetlab.SMPP.SmppClient.evDeliverSm> in class <xref:Inetlab.SMPP.SmppClient>. The event rises on <xref:Inetlab.SMPP.Common.CommandSet.DeliverSm> command arrival. Any method subscribed to that event will receive information about inbound messages.

...

```
_client.evDeliverSm += OnDeliverSm;
```

...

```
private void OnDeliverSm(object o, DeliverSm deliverSm)
{
    if (deliverSm.MessageType == MessageTypes.SMSCDeliveryReceipt)
    {
        _log.Info("Delivery Receipt received");
    }
    else
    {
        _log.Info("Incoming SMS received");
    }
}
```

The InetLab.SMPP library allows to concatenate received parts of the long message into a single message the same way as mobile phone does. The class <xref:Inetlab.SMPP.Common.MessageComposer> is used for that.

If you do not receive inbound messages you may need to look for an answer in according Troubleshooting article.

Tracking message sending and delivery

In the context of the SMPP protocol, it's crucial to keep a record of messages being delivered to the recipient.

Tracking sending

The SMPP server generates a unique <xref:Inetlab.SMPP.PDU.SubmitSmResp.MessageId> for each <xref:Inetlab.SMPP.PDU.SubmitSm> PDU received from the SMPP client and sends it back in the response <xref:Inetlab.SMPP.PDU.SubmitSmResp>. In the case of a concatenated message, which consists of multiple parts, each part is assigned a unique MessageId generated by the server.

A response with the status ESME_ROK and the defined MessageId means that the server has accepted a <xref:Inetlab.SMPP.PDU.SubmitSm> PDU for delivery.

For more information, you can refer to articles on how to create and send messasge with an SMPP client or how the SMPP server receives the message.

Tracking delivery

To obtain a delivery receipt, the property `RegisteredDelivery` must be set to 1 in the `SubmitSm` PDU. The delivery receipt contains the field `MessageId`, which matches the `MessageId` of the response `SubmitSmResp` for the original message/part sent earlier.

> **NOTE**
>
> Sometimes, the delivery receipt in the `DeliverSm` PDU is received earlier than the `SubmitSmResp`. You must consider this when implementing your tracking mechanisms.

The following code example is a simple handler for the event <xref:Inetlab.SMPP.SmppClient.evDeliverSm>. It gets the `MessageId` and status from received delivery report <xref:Inetlab.SMPP.Common.CommandSet.DeliverSm>.

...

```
_client.evDeliverSm += OnDeliverSmTracking;
```

...

```
private void OnDeliverSmTracking(object o, DeliverSm deliverSm)
{
    if (deliverSm.MessageType == MessageTypes.SMSCDeliveryReceipt)
    {
        _log.Info("Delivery Receipt received");

        string messageId = deliverSm.Receipt.MessageId;
        MessageState deliveryStatus = deliverSm.Receipt.State;
    }
}
```

Read more details about tracking message delivery status

SMPP Server

Create SMPP-server and Connect (with sample app)

The following minimal code structure creates SMPP-server:

```
SmppServer _server = new SmppServer(new IPEndPoint(IPAddress.Any, 7777));
await _server.StartAsync();
```

First line prepares SMPP-server to be started at the port 7777. The second line actually starts the server and getting server ready to accept connection requests from TCP/IP clients. The SMPP-server creates instance of <xref:Inetlab.SMPP.SmppServerClient> class and raises the event <xref:Inetlab.SMPP.SmppServer.evClientConnected> for each TCP/IP client connected.

All clients connected are added automatically to the collection <xref:Inetlab.SMPP.SmppServer.ConnectedClients> of <xref:Inetlab.SMPP.SmppServer> object.

You may implement any preliminary checks in the event-handler subscribed to <xref:Inetlab.SMPP.SmppServer.evClientConnected> event. For example, you may perform an IP-address check and disconnect “wrong” clients immediately.

Please explore the sample SMPP Server program at the link.

Client authentication (Bind)

Each SMPP-client has to send **Bind** command to start working with SMPP-servers. Bind stands for authentication according to the SMPP protocol. There must be an event-handler attached to <xref:Inetlab.SMPP.SmppServer.evClientBind> event to enable to bind-request processing by SMPP-server. The event is raised each time SMPP-server receives Bind command:

```
_server.evClientBind += (sender, client, bindPdu) =>
{
    //process Bind PDU
};
```

By using an empty event-handler as in the example above you allow any SMPP-client authenticate on your server. If there is no event-handler attached, the authentication will not succeed. Consequently, the SMPP-server will return response <xref:Inetlab.SMPP.PDU.BindResp> to the SMPP-client containing <xref:Inetlab.SMPP.Common.CommandStatus.ESME_RBINDFAIL> status.

It is common to implement various authentication rules, login checks and other security checks in the event-handler subscribed to <xref:Inetlab.SMPP.SmppServer.evClientBind>.

Keeping connection active (InactivityTimeout and EnquireLink)

InactivityTimeout

To save server resources, it is useful to disconnect inactive clients. In general, inactive client is the one who neither sends neither receives commands (messages).

There is a parameter <xref:Inetlab.SMPP.SmppServerClient.InactivityTimeout> with default value of 2 minutes for <xref:Inetlab.SMPP.SmppServerClient> instances. The <xref:Inetlab.SMPP.SmppServer> closes connection to clients based on this timer. It is possible to disable <xref:Inetlab.SMPP.SmppServerClient.InactivityTimeout> by assigning it value **TimeSpan.Zero**.

Example of setting <xref:Inetlab.SMPP.SmppServerClient.InactivityTimeout> for 15 seconds once an SMPP-client is connected:

```
SmppServer _server = new SmppServer(new IPEndPoint(IPAddress.Any, 7777));

_server.evClientConnected += (s, client) =>
{
    client.InactivityTimeout = TimeSpan.FromSeconds(15);
};

await _server.StartAsync();
```

<xref:Inetlab.SMPP.SmppServerClient.InactivityTimeout> is possible to set inside event-handler for <xref:Inetlab.SMPP.SmppServer.evClientConnected> event only.

EnquireLink

When there is no messages to send/receive but the connection has to be kept the EnquireLink command is engaged.

It is possible to perform an automatic connection check for <xref:Inetlab.SMPP.SmppServerClient> or <xref:Inetlab.SMPP.SmppClient> using property <xref:Inetlab.SMPP.SmppClientBase.EnquireLinkInterval>. The <xref:Inetlab.SMPP.SmppClientBase.EnquireLinkInterval> is the inactivity time interval after which the command EnquireLink is sent automatically.

EnquireLink example:

```
SmppServer _server = new SmppServer(new IPEndPoint(IPAddress.Any, 7777));

_server.evClientBind += (s, client, bind) =>
{
    client.EnquireLinkInterval = TimeSpan.FromSeconds(15);
};

await _server.StartAsync();
```

In that example, we configure the <xref:Inetlab.SMPP.SmppServerClient> instance to check connection each 15 seconds of SMPP-client inactivity. It is possible to set EnquireLinkInterval in event-handlers for <xref:Inetlab.SMPP.SmppServer.evClientConnected> event or <xref:Inetlab.SMPP.SmppServer.evClientBind> event.

An automatic connection check is started only after successful **Bind**. Client without **Bind** considered inactive and will be disconnected if <xref:Inetlab.SMPP.SmppServerClient.InactivityTimeout> was set.

Please note, values <xref:Inetlab.SMPP.SmppServerClient.InactivityTimeout> and <xref:Inetlab.SMPP.SmppClientBase.EnquireLinkInterval> are to be set for each instance of <xref:Inetlab.SMPP.SmppServerClient> created i.e. for each client connected.

The event handler for event <xref:Inetlab.SMPP.SmppServer.evClientDisconnected> is called when client disconnects.

Receive messages

To receive messages it is necessary to create an event handler for <xref:Inetlab.SMPP.SmppServer.evClientSubmitSm> event. The <xref:Inetlab.SMPP.SmppServer.evClientSubmitSm> event is raised each time packet with <xref:Inetlab.SMPP.PDU.SubmitSm> command arrives. The remote SMPP-client uses this command to send SMS to SMPP-server (SMS Center).

```
_server.evClientSubmitSm += (sender, client, submitSm) =>
{
    // process SubmitSm PDU here
};
```

Even with an empty event handler attached, the SmppServerClient will automatically generate <xref:Inetlab.SMPP.PDU.SubmitSmResp> packet and put the <xref:Inetlab.SMPP.Common.CommandStatus.ESME_ROK> status in **submitSm.Response** field. In addition, the unique identifier "MessageId" for each message/part received will be created and placed into response packet. The event handler allows you to change **submitSm.Response.MessageId** or any other property of <xref:Inetlab.SMPP.PDU.SubmitSmResp> object.

```
submitSm.Response.MessageId = "myUnuqueID";
```

If there is no event handler attached to <xref:Inetlab.SMPP.SmppServer.evClientSubmitSm> event, the server sends to client the response <xref:Inetlab.SMPP.PDU.SubmitSmResp> containing status <xref:Inetlab.SMPP.Common.CommandStatus.ESME_RSUBMITFAIL>.

It is common to implement various rules for inbound messages (such as processing, saving, sending, storing, parts collecting, etc.) inside the event handler attached to <xref:Inetlab.SMPP.SmppServer.evClientSubmitSm> event.

Send messages

The SMPP-server creates <xref:Inetlab.SMPP.SmppServerClient> object automatically for each SMPP-client connected. Calling method <xref:Inetlab.SMPP.SmppServerClient.DeliverAsync*> of the <xref:Inetlab.SMPP.SmppServerClient> object sends a message to the respective SMPP-client.

To start, it is necessary to choose <xref:Inetlab.SMPP.SmppServerClient> instance from the list available at <xref:Inetlab.SMPP.SmppServer.ConnectedClients> property. You may use any <xref:Inetlab.SMPP.SmppServerClient> properties as criteria for choosing the recipient/SMPP-client.

For example, let us crate arbitrary message at SMPP-server and send it to the SMPP-client. To choose a recipient SMPP-client from the list we will use SystemID value (SMPP-client login). The message will be sent to the first client having SystemID matching field "To" value of the message.

Assuming the SMPP-server already created, minimally configured and started and the server parameter will be passed to the method, the sending method will be as follows:

```
public static async Task SendSms(SmppServer _server)
{
    //prepare message data
    string sender = "123";
    string recipient = "456";
    string text = "hello!";

    //searching recepient by criteria
    SmppServerClient clientRecepient = _server.ConnectedClients.FirstOrDefault(c => c.SystemID == recipient);

    //creating message and sending
    if (clientRecepient != null)
    {
        IList<DeliverSm> textMessage =
SMS.ForDeliver().From(sender).To(recipient).Text(text).Create(clientRecepient);
        IEnumerable<DeliverSmResp> response = await clientRecepient.DeliverAsync(textMessage);
    }
}
```

To have a message sent in the example above, the SMPP-server must have an SMPP-client with SystemId “456” already connected when method SendSms is called.

Deliver messages from sender to recipient

By combining receive message and send message examples we can implement basic way to deliver messages from sender to recipient via SMPP-server. Let's make a method for receiving inbound messages from an SMPP-client, searching suitable recipient among SMPP-clients connected and sending the message to it. For the sake of example, let us consider any SMPP-client having SystemID (login) equal to message "To" field as "suitable".

```
private static async Task Main(string[] args)
{
    LogManager.SetLoggerFactory(new ConsoleLogFactory(LogLevel.Verbose));

    SmppServer _server = new SmppServer(new IPEndPoint(IPAddress.Any, 7777));

    _server.evClientBind += (s, c, p) => { }; //allow all to authenticate on the server
    _server.evClientSubmitSm += async (smppServer, smppServerClient, submitSm) => await
ForwardSms(smppServer, smppServerClient, submitSm);

    await _server.StartAsync();

    Console.ReadLine();
}

private static async Task ForwardSms(object smppServer, SmppServerClient smppServerClient, SubmitSm submitSm)
{
    SmppServer _server = (SmppServer)smppServer;

    //prepare message
    string fromField = submitSm.SourceAddress.ToString();
    string toField = submitSm.DestinationAddress.ToString();
    string textField = submitSm.GetMessageText(smppServerClient.EncodingMapper);

    //search recepient
    SmppServerClient clientRecepient = _server.ConnectedClients.FirstOrDefault(c => c.SystemID == toField);

    //send
    if (clientRecepient != null)
    {
        IList<DeliverSm> textMessage =
SMS.ForDeliver().From(fromField).To(toField).Text(textField).Create(clientRecepient);
        var result = await clientRecepient.DeliverAsync(textMessage);
    }
}
```

There is an event handler created and named "ForwardSms". It is subscribed to <xref:Inetlab.SMPP.SmppServer.evClientSubmitSm> event, responsible for inbound messages. The event handler is getting the inbound message as a third argument (submitSm) and prepares it for sending further. In addition, it picks the suitable recipient out of the list of SMPP-clients connected (<xref:Inetlab.SMPP.SmppServer.ConnectedClients> property) and forwards message to it.

To test the setup, connect two SMPP-clients (sender and recipient) to this SMPP-server. Sender login is not important, but for recipient login use "client002". Now send a message from the first SMPP-client and put "client002" in the "Destination (To)" field. The message should arrive to client with login "client002". It will work even if you have only one SMPP-client connected but "To" field contains its login. Of course, the way of choosing the recipient is totally up to the developer.

Using a similar approach it is possible to implement the SMPP Gateway for forwarding messages via SMPP-client connected to another SMPP-server.

Implementing SMPP Gateway (with sample app)

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

Please note the SMPP Gateway sample program is available at the link.

Such application should start at least one <xref:Inetlab.SMPP.SmppServer> to be able to receive SMPP commands on a TCP port and several <xref:Inetlab.SMPP.SmppClient> instances to send message to other SMPP servers (SMSC, Provider).

When a customer sends <xref:Inetlab.SMPP.PDU.SubmitSm> command to your server, you need to send back a response <xref:Inetlab.SMPP.PDU.SubmitSmResp> with assigned <xref:Inetlab.SMPP.PDU.SubmitSmResp.MessageId>. Later, when you forward this message to another server, you will receive another <xref:Inetlab.SMPP.PDU.SubmitSmResp.MessageId> from SMSC.

This SMSC <xref:Inetlab.SMPP.PDU.SubmitSmResp.MessageId> should also be replaced in <xref:Inetlab.SMPP.PDU.DeliverSm> (<xref:Inetlab.SMPP.Common.Receipt>) for the target client.

You might want to implement smart routing for incoming messages. F.i. when you are going to forward SMS message you can estimate which <xref:Inetlab.SMPP.SmppClient> connection accepts destination phone number and costs less.

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

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

These four values help later to find a corresponding client that should receive a delivery receipt from the provider:

- Client's sequence number
- Client's MessageId
- Provider's sequence number
- Provider's MessageId

When <xref:Inetlab.SMPP.PDU.DeliverSm> comes from the provider and contains "DeliveryReceipt", you should do the following steps:

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

Example of forwarding message from one client to another is on the page "Message delivery from sender to recipient"

Control SMPP responses

By default, the InetLab SMPP-server dispatches <xref:Inetlab.SMPP.Common.CommandSet.SubmitSmResp> commands automatically one by one after each <xref:Inetlab.SMPP.SmppServer.evClientSubmitSm> event handled. You may want to override this flow in order to process, save or prepare responses in a more controllable way. For example, you may like saving data to SQL in bulk instead of single "per message" transactions. Also, you may like to control the moment when responses are sent to the client.

Automatic response sending can be prevented in an event handler by changing Response field to **null**:

```
private static void onSubmitSm(object sender, SmppServerClient client, SubmitSm submitSm)
{
    SubmitSmResp generatedResp = submitSm.Response;

    //preventing response automated return
    submitSm.Response = null;

    //custom routine for the response
    Task handleTask = HandleRequestAsync(submitSm);
}

private static async Task HandleRequestAsync(SubmitSm submitSm)
{
    //Generating and sending response manually
    SmppServerClient client = submitSm.Client as SmppServerClient;
    var response = new SubmitSmResp(submitSm);
    await client.SendResponseAsync(response);
}
```

In the example above, we prevent auto-response by setting submitSm.Response field to **null**. Then we call a method and supply it with <xref:Inetlab.SMPP.Common.CommandSet.SubmitSm> data. The method generates and submits a response to the client.

Troubleshooting

Common Mistakes

Incoming messages not received

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 applications. SMSC sends messages to application where DeliverSm is not expected.

Wrong message text encoding

Please clarify with SMPP provider which encoding (character set) is expected for @Inetlab.SMPP.Common.DataCodings value.

Read more about encoding on page "Mapping DataCodings to .NET Encoding".

Message concatenation does not work

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

Read more about contatenation on page "Concatenation".

Library version was changed

If you observe plenty of syntax errors or command syntax changes, probably you are using outdated library version. Otherwise you might have library updated but using older version codebase.

Read more on page "Migration from v1.x to 2.x".

Lost of Connection

General Case

Lost of connection can be caused by :

- Router crash/reboot. Any of the routers along the route from one side to the other may crash or be rebooted; this causes a loss of connection if data is being sent at that time. If no data is being sent at that exact time, then the connection is not lost.
- Network cable is unplugged. Any network cables unplugged along the route from one side to the other will cause a loss of connection without any notification.

Lost of connection in Inetlab.SMPP library is detected within ENQUIRE_LINK request or when any other SMPP PDU is being sent. It can happen that a client detects disconnection earlier than a server. If the server is configured to allow only one connection for an SMPP account it may reject the subsequent bind requests by responding with BIND_RESP and status ESME_RALYBND. Once the server detects connection staled, it accepts the bind request again.

If you face such situation in your application you need to reconnect to the SMPP provider in 1-5 Minutes. Inetlab.SMPP library also provides connection recovery feature for SmppClient.

Also please be aware of SmppServer timeout settings.

SMPPCLIENT_NOCONN Status

The SmppResponse with status <xref:Inetlab.SMPP.Common.CommandStatus.SMPPCLIENT_NOCONN> comes after an unsuccessful attempt to send SMPP-command. Basically, it means there is no connection to the server. This may happen for the number of reasons (for example, if TLS versions do not match) and at various stages (before/after Connect, before/after Bind, during sending). If this is happening constantly and often, you should try to find what causes it in the first instance.

To continue normal operation, you should reestablish connection with the server and resend all SubmitSm with a new sequence number. The sequence number can be generated automatically when you set **submtitSm.Header.Sequence = 0** or you can generate it manually with the code:

```
submitSm.Header.Sequence = _client.SequenceGenerator.NextSequenceNumber();
```

Throttling error

SMSC can limit number of submitted PDU for SMPP account. When allowed message limit is exceeded, server returns status <xref:Inetlab.SMPP.Common.CommandStatus.ESME_RTHROTTLED>.

To avoid throttling error you can specify a number of messages per second in <xref:Inetlab.SMPP.SmppClient>. For this purpose you can define <xref: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, TimeSpan.FromMinutes(1));

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

Please note that speed limit works properly only on release version without attached debugger.

Built-in Logging

Logging is a universal approach to detecting problems and debugging your software.

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

For example:

- NLog
- Log4Net

The library provides <xref:Inetlab.SMPP.Logging.ConsoleLogFactory> and <xref:Inetlab.SMPP.Logging.FileLogFactory> classes.

When the application starts you need to register global <xref:Inetlab.SMPP.Logging.ILogFactory> for the library.

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

or you can set <xref:Inetlab.SMPP.SmppClientBase.Logger> property when you create instances of <xref:Inetlab.SMPP.SmppClient>, <xref:Inetlab.SMPP.SmppServerClient> or <xref:Inetlab.SMPP.SmppServer>

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

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

Implementation example for <xref:Inetlab.SMPP.Logging.ILog> and <xref:Inetlab.SMPP.Logging.ILogFactory> interfaces:

```
public class ConsoleLogFactory : ILogFactory
{
    private readonly 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;
        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 static 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 "";
    }



}
```

Read more about Logging at page "Creating global and local logger".

Diagnostic

Metrics

To monitor <xref:Inetlab.SMPP.SmppClient> or <xref:Inetlab.SMPP.SmppServerClient> performance you can use <xref:Inetlab.SMPP.SmppClientBase.Metrics> property. It contains many useful values calculated automatically.

The <xref:Inetlab.SMPP.Metrics.ISmppClientMetrics> type of the <xref:Inetlab.SMPP.SmppClientBase.Metrics> property has the following structure:

- Sent - Gets the metrics for messages sent.
  - InQueue - Number of messages stored in the queue (awaiting to be sent to the network).
  - Total - Total number of messages sent to the network.
  - Requests
    - Total - Total number of requests sent to the network.
    - PerSecond - Transferring speed for the last second. Messages per second.
    - AvgPerSecond - Average speed since last Reset. Messages per second.
    - WaitingForResponse - Gets number of request PDUs that didn't receive a response with the same sequence number.
    - ResponseTime - shows how fast the remote side processes the requests sent from the application.
      - Minimum - Minimum response time for the request sent to remote side.
      - Average - Average response time for the request sent to remote side.
      - Maximum - Maximum response time for the request sent to remote side.
  - Responses
    - Total - Total number of responses sent to the network.
    - PerSecond - Transferring speed for the last second. Messages per second.
    - AvgPerSecond - Average speed since last Reset. Messages per second.
- Received - Gets the metrics for messages received.
  - InQueue - Number of received requests stored in the queue (received but not processed in the worker threads).
  - Total - Total number of messages received from the network.
  - Requests
    - Total - Total number of requests received.
    - PerSecond - Transferring speed for the last second. Messages per second.
    - AvgPerSecond - Average speed since last Reset. Messages per second.
    - WaitingForResponse - Number of received requests that didn't processed by the application.
    - ResponseTime - shows how fast the application processes the requests from remote side.
      - Minimum - Minimum response time for the request received from remote side.
      - Average - Average response time for the request received from remote side.
      - Maximum - Maximum response time for the request received from remote side.
  - Responses
    - Total - Total number of responses received.
    - PerSecond - Transferring speed for the last second. Messages per second.
    - AvgPerSecond - Average speed since last Reset. Messages per second.

All properties in the metrics objects are constantly changed during message transferring. If you want to fix the metric's value for further analysis, you can use the method Snapshot().

```
var clientMetrics = _client.Metrics.Snapshot();
```

This method copies all current values to new metrics object where the values won't change anymore. The same Snapshot method is also available for child properties, for example:

```
var requestMetrics = clientMetrics.Sent.Requests.Snapshot();
```

or

```
var responseMetrics = clientMetrics.Received.Responses.Snapshot();
```

You can also use Reset() method for the cases when you need to analyze some part of the process and begin to watch values from zero. The Reset() method is also available for each child metric object.

```
// start sent requests metrics from zero
_client.Metrics.Sent.Requests.Reset();

//send a batch of messages
var responses = await _client.SubmitAsync(batch);

//take snapshot of metrics values
var sentRequests = _client.Metrics.Sent.Requests.Snapshot();

// display metrics values as string
string performance = sentRequests.ToString();
```

Special events

You can use special events in base class <xref:Inetlab.SMPP.SmppClientBase> for tracking PDUs:

- with the event <xref:Inetlab.SMPP.SmppClientBase.evPduReceiving> you can monitor all incoming PDUs.
- event <xref:Inetlab.SMPP.SmppClientBase.evPduSending> is invoked before sending the PDU to network.

Telemetry

Introduction

With the telemetry feature in the Inetlab.SMPP library, you can easily gain valuable insights into your SMPP applications. By enabling telemetry and exporting spans to a collector, you can effortlessly monitor and analyze the behavior and performance of your application. This feature allows you to identify issues, optimize performance, and integrate with popular monitoring tools like Azure Application Insights or OpenTelemetry. This article will provide you with a step-by-step guide on configuring your project and utilizing the telemetry feature effectively, enabling you to make informed decisions and deliver high-performance SMPP applications.

Prerequisites

To use the telemetry feature in the Inetlab.SMPP library, ensure you have at least version 2.9.28 installed and that your application targets .NET 6.0 or later. Additionally, set up a telemetry collector or monitoring tool like Jaeger, Azure Application Insights or OpenTelemetry to receive exported spans.

Configuration

To enable telemetry and export spans to a telemetry collector like Jaeger, follow these steps:

1) Install Jaeger

Begin by installing Jaeger in your environment. You can find installation instructions specific to your platform on the Jaeger website (https://www.jaegertracing.io).

2) Install the necessary packages:

```
dotnet add package OpenTelemetry.Exporter.OpenTelemetryProtocol
dotnet add package OpenTelemetry.Extensions.Hosting
```

3) Configure the telemetry provider:

```
    // Configure spans exporter to OLTP collector.
    services.AddOpenTelemetry()
        .WithTracing(b =>
        {
            b.ConfigureResource(r => r.AddService("MyApp", "1.0"));
            b.AddSource("Inetlab.SMPP");
            b.AddOtlpExporter();
        });
```

Jaegger accepts OpenTelemetry Protocol (OLTP) on port 4317 (gRPC) and 4318 (HTTP)

Monitoring and Analysis

Jaeger is a popular distributed tracing system that provides powerful monitoring and analysis capabilities for your SMPP applications. Follow these steps to monitor and analyze telemetry data:

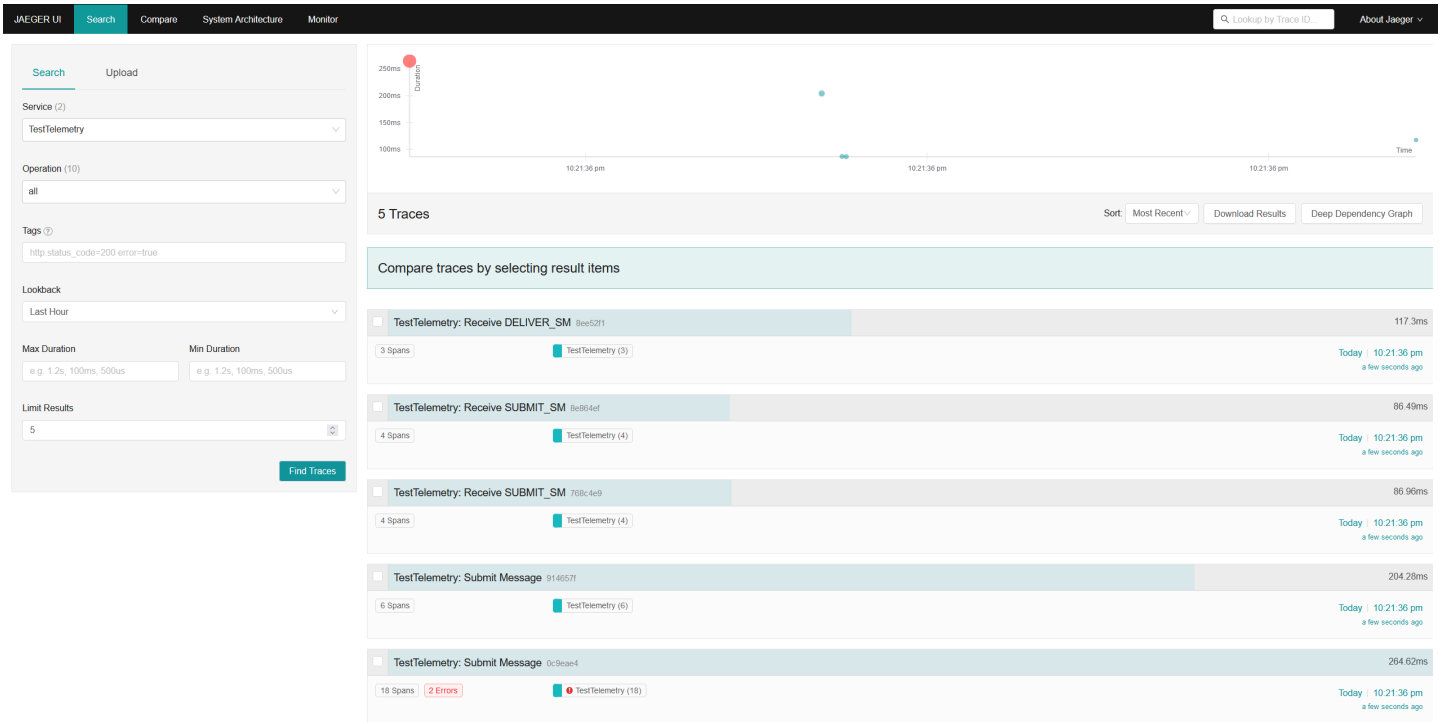
1) Start your SMPP application:

With the Jaeger exporter configured, start your SMPP application. It will now generate and export spans to Jaeger.

2) Access the Jaeger UI:

Open your web browser and navigate to the Jaeger UI. The default URL is usually http://localhost:16686. You should see the

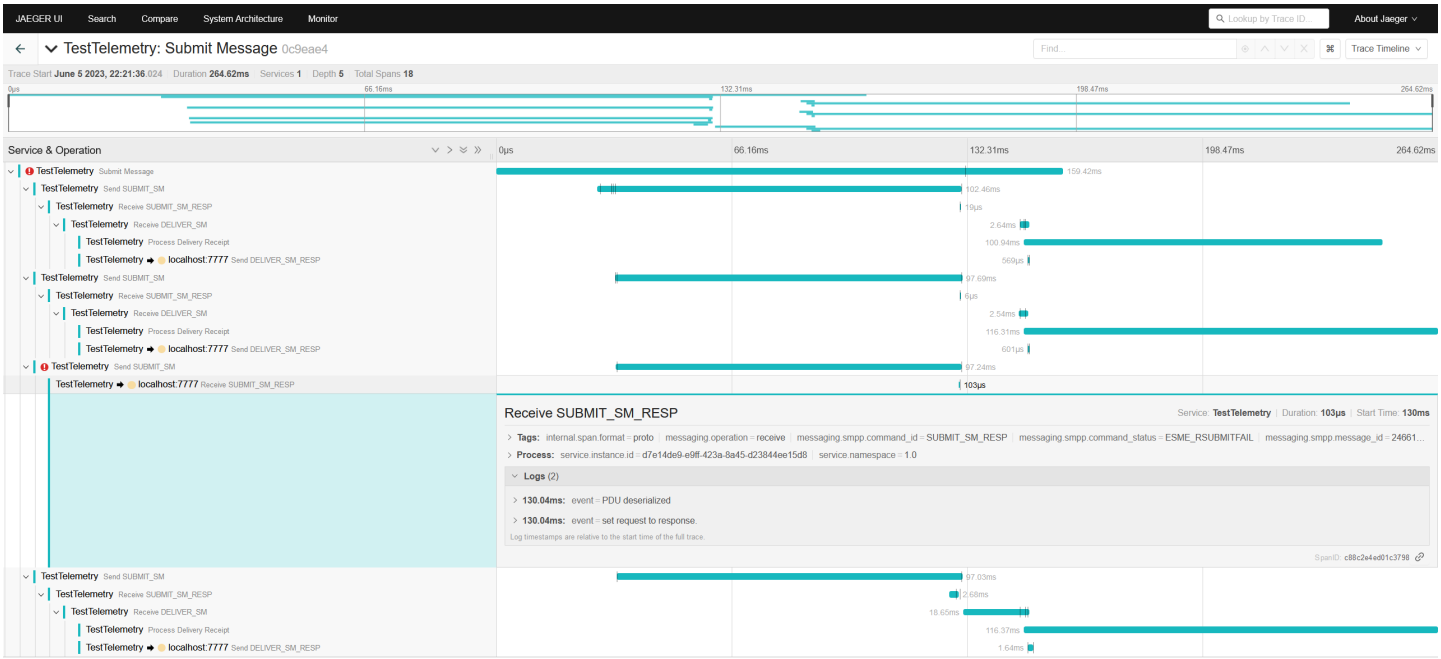
Jaeger UI dashboard.



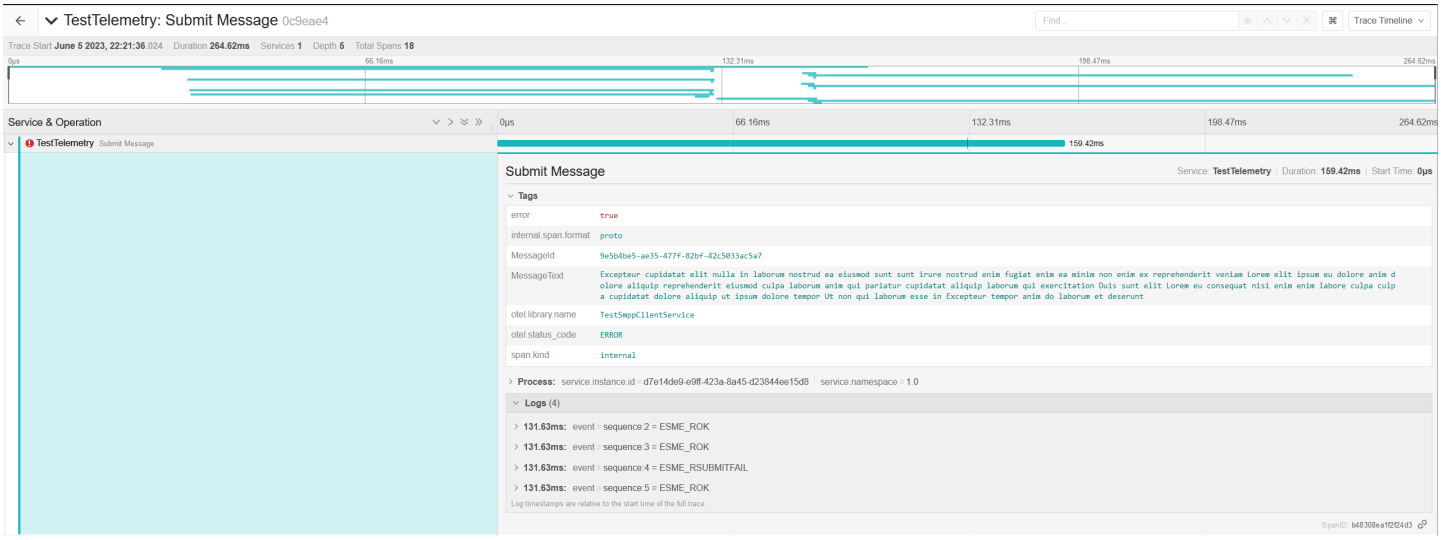
3) Analyze telemetry data:

In the Jaeger UI, you can analyze the telemetry data captured from your SMPP application.

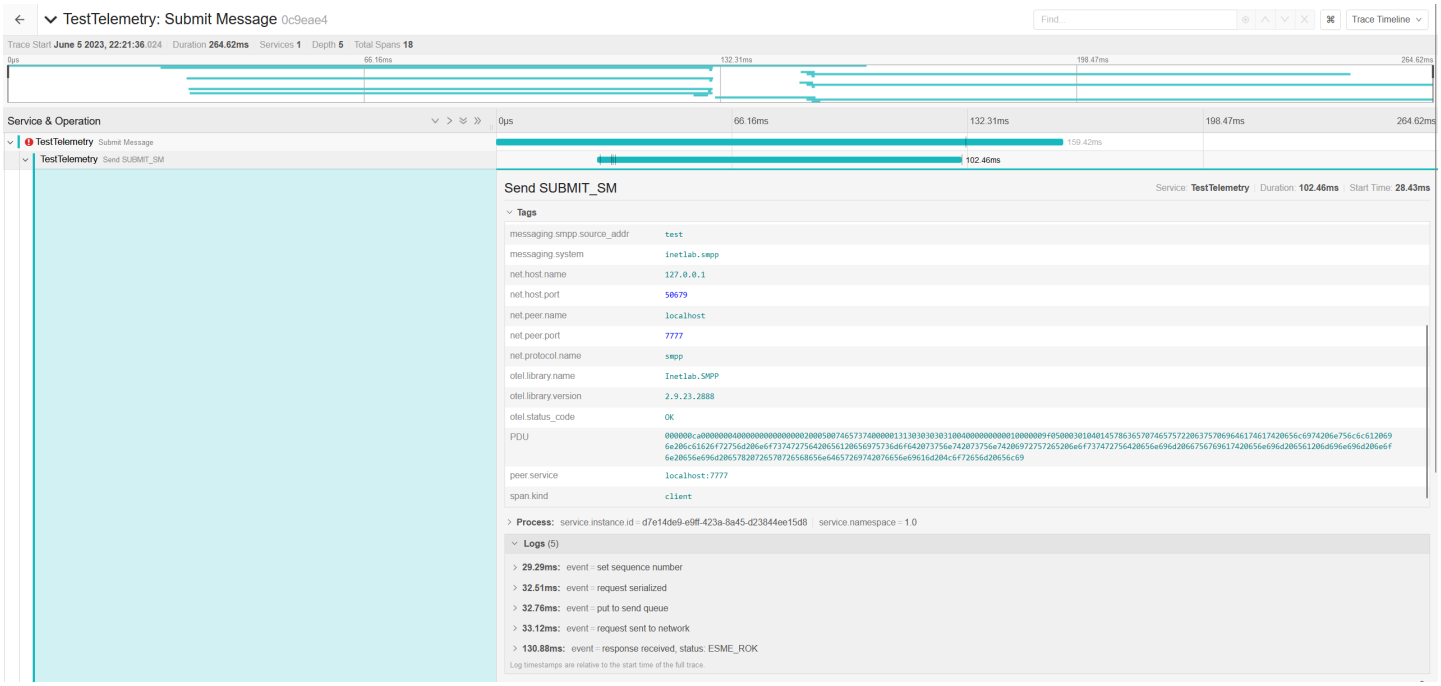
The trace view showcases a concatenated message with four SUBMIT_SM PDUs. In the trace, the third PDU is received with an ESME_RSUBMITFAIL status, indicating a submission failure. The trace view also displays the timely response from the SMSC, with SUBMIT_SM_RESP messages sent within approximately 100ms for each SUBMIT_SM command. Additionally, the trace reveals the relationship between the delivery receipts and each corresponding SUBMIT_SM, providing valuable insights into the message delivery process and its associated responses.



In the following screenshot, you can observe the activity generated by the TestTelemetry application. It provides detailed information, including the original message text and corresponding message ID. This allows you to easily track and identify specific messages within the telemetry data.



The screenshot displays the activity generated by the library, which includes tags with essential fields extracted from the SUBMIT_SM PDU. Moreover, it provides the actual byte array that was sent to the network, giving you visibility into the raw data. Additionally, the activity captures timing events for each step involved in sending the SUBMIT_SM PDU, allowing you to analyze the duration of each operation and identify any potential bottlenecks or performance issues.



Conclusion

In conclusion, the telemetry feature in the Inetlab.SMPP library provides .NET developers with valuable insights to monitor and optimize their SMPP applications. By configuring the project to export telemetry data, developers can gain a deeper understanding of their application's behavior and performance. This allows for effective issue identification and performance enhancements.

We value your feedback and suggestions. If you have any suggestions or questions, please feel free to reach out to us through the contact us form. We appreciate your input and are dedicated to continuously improving our library to meet your needs.

Tuning

Threading

Library creates for each connected client 3 worker threads that handle received request PDUs and call corresponding event handlers, f.i. evSubmitSm, evDeliverSm.

> **NOTE**
>
> Since version 2.9.15 the TaskScheduler.Default (thread pool) is used by default.

You can increase the number of worker threads for each client.

```
var scheduler = _client.ReceiveTaskScheduler as WorkersTaskScheduler;
if (scheduler != null)
    scheduler.WorkerCount = 10;
```

When your application establishes a lot of SMPP sessions, you may want to create a global task scheduler and set required number of worker threads for the process.

```
var scheduler = new WorkersTaskScheduler(100);
```

You need then assign this scheduler to the client. It is your responsibility to dispose the scheduler instance.

Before SmppClient connects to remote host.

```
client.ReceiveTaskScheduler = scheduler;
```

For the SmppServer when client connects.

```
public void server_evClientConnected(object sender, SmppServerClient client)
{
    client.ReceiveTaskScheduler = _scheduler;
}
```

TaskScheduler.Default can be used as well. In this case all event handlers for received requests will be scheduled to ThreadPool.

```
_client.ReceiveTaskScheduler = TaskScheduler.Default;
```

You can force the thread pool to create more worker threads with the command

```
   ThreadPool.SetMinThreads(200, 200);
```

It is especially useful for the SmppServer to improve performance immediately after start.

Networking

To reduce number of calls to network adapter and increase transfering speed, you can 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).

Memory

Memory consumption for a client can be reduced if you limit the number of requests received from the client but not processed by the application.

```
_client.ReceivedRequestQueueLimit = 1000;
```

If a client sends PDUs too fast, the library stops reading from network until a free slot is available in the receive queue. This is commonly referred to as "flow control" or "back pressure". This is also affects the responses sent by the client. They won't be read from network because of reading delay.

The number of request waiting for response from remote side can be also limited with the code

```
_client.SendQueueLimit = 1000;
```

The incomplete request objects consume memory of the running process. This property helps to reduce memory consumption and prevent unsuccess responses. When remote side cannot process messages fast enough, number of sent messages may exceed queue limit (ESME_RMSGQFUL). When this property is defined the sending to network will be delayed until the queue has a free slot (remote side has sent a response).

Wireshark

The best way to analyze SMPP Protocol is to capture network traffic with Wireshark tool.

SMPP related Wiki article is here.

FAQ

SMPP Cient

Sending Commands and Getting Responses

Concatenation

SMPP Connection Mode

Deivery Receipt

EnquireLink

Binary SMS

How to install the license file

Logging

Map Encoding

Message Composer

Performance (TPS)

SMPP Server (with sample app)

SMPP Address

SSL/TLS Connection

SubmitMulti. Send message to multiple destinations

MMS notifications

WAP Push

Implementing USSD (Unstructured Supplementary Service Data)

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 of usage scenarios

```
public static async Task SendLongText(SmppClient client)
{
    string longText = new string('A', 300);

    var resp = await client.SubmitAsync(
        SMS.ForSubmit()
            .From("short_code")
            .To("436641234567")
            .Coding(DataCodings.UCS2)
            .Text(longText)
    );
}
```

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: <xref:Inetlab.SMPP.Common.DataCodings.UnicodeFlashSMS>, <xref:Inetlab.SMPP.Common.DataCodings.DefaultFlashSMS>

How can I set sequence number before sending PDU

<xref:Inetlab.SMPP.SMS> Builder has <xref:Inetlab.SMPP.Builders.IBuilder`1.Create*> method that returns <xref:Inetlab.SMPP.PDU.SubmitSm> list with sequence numbers set to 0.

You can assign the next number from the <xref:Inetlab.SMPP.SmppClientBase.SequenceGenerator> and pass this PDU list to <xref:Inetlab.SMPP.SmppClient.Submit(Inetlab.SMPP.PDU.SubmitSm[])> method.

```
IList<SubmitSm> pduList = SMS.ForSubmit()
    .From("5555")
    .To("436641234567")
    .Text("test text")
    .Create(client);

foreach (SubmitSm submitSm in pduList)
{
    submitSm.Header.Sequence = client.SequenceGenerator.NextSequenceNumber();
}

var resp = await client.SubmitAsync(pduList);
```

How can I resend the PDU after failed response status

You can assign the next number from the <xref:Inetlab.SMPP.SmppClientBase.SequenceGenerator> and send the PDU again.

```
pdu.Header.Sequence = client.SequenceGenerator.NextSequenceNumber();
var resp = await client.SubmitAsync(pdu);
```

When you reset the sequence number to 0, the next number will be automatically assigned to the PDU when it is sent.

```
pdu.Header.Sequence = 0; // set next sequence number on submit
var resp = await client.SubmitAsync(pdu);
```

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

```
public class SMSMessage
{
    public string PhoneNumber { get; set; }
    public string Text { get; set; }
}

public static async Task SendMessageBatchAsFastAsPossible(SmppClient client)
{

    var messageList = GetNext100UnsentMessages();

    List<SubmitSm> pduList = new List<SubmitSm>();
    foreach (var message in messageList)
    {
        var pduBuilder = SMS.ForSubmit()
            .From("5555")
            .To(message.PhoneNumber)
            .Text(message.Text);

        pduList.AddRange(pduBuilder.Create(client));
    }

    SubmitSmResp[] resp = await client.SubmitAsync(pduList.ToArray());
}

private static IEnumerable<SMSMessage> GetNext100UnsentMessages()
{
    for (int i = 0; i < 100; i++)
    {
        yield return new SMSMessage
        {
            PhoneNumber = (436641234567 + i).ToString(),
            Text = $"Test {i}"
        };
    }
}
```

How to create SubmitMulti PDUs for multiply recipients

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

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

Sending Commands and Getting Responses

SMPP is based on the exchange of request and response protocol data units (PDUs) between the SMPP-client (ESME) and the SMPP-server (SMSC) over an underlying TCP/IP network connection.

The SMPP protocol defines:

- a set of operations and associated Protocol Data Units (PDUs) for the exchange of short messages between an SMPP-client and an SMPP-server
- the data that an SMPP-client application can exchange with an SMPP-server during SMPP operations

Sending Commands

The SMPP-client is ready to exchange commands with SMPP-server right after establishing connection and successful bind (authentication).

All commands and responses are transmitted as PDUs. The command name is specified in the PDU header.

For example, the command SubmitSm serves for sending messages from SMPP-client to an SMPP-server and DeliverSm command serves for sending messages from SMPP-server to SMPP-client. There are commands for authentication, binary data transmission and many more described in the SMPP protocol specification.

When you call a method, the Inetlab.SMPP library automatically forms PDUs with respective SMPP-commands and other data inside. If needed, devlopers can form any PDU manually.

Getting Responses

Command responses contain important information. By analyzing responses you can figure out if SMPP-server accepted the message for delivery, is there a connection active, was authentication successful and other.

Please note: Every SMPP operation must consist of a request PDU and associated response PDU. The receiving entity must return the associated SMPP response to an SMPP PDU request.

Concatenation

The GSM standard defines a maximum of 140 octets for a single short message and thus does not support the transmission of more than these 140 octets per message. Therefore, a receiving SMSC will usually not accept a submit operation which will result in a short message of >140 octets, unless it has implemented an automatic concatenation mechanism, which divides a long message in multiple parts of 140 octets.

Various SMPP providers support various concatenation ways. Inetlab.SMPP library supports 3 ways:

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

SMS Builder class 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.SubmitAsync(builder);
}
```

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

```
public static 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 the 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.Concatenation(ConcatenationType.SAR);

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

Example how to get concatenation parameters from TLV Parameters:

```
public static 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.SubmitAsync(builder);
```

SMPP Connection Mode

The SMPP connection mode is to be specified by an SMPP-client when attempting to authenticate at SMPP-server i.e. when performing bind. It defines basic data exchange rule between client and server.

There are 3 SMPP connection modes available:

- 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.

Example of specifying the connection type by SMPP-client.

Delivery Receipt

Requesting the delivery receipt

SMPP protocol provides the ability to request a delivery receipt in `SubmitSm` PDU submitted to the server. There are two ways how you can do it with the InetLab.SMPP library.

```
submitSm.RegisteredDelivery = 1;
// or
submitSm.SMSCReceipt = SMSCDeliveryReceipt.SuccessOrFailure;
```

or

```
var resp = await client.SubmitAsync(
    SMS.ForSubmit()
        .From("short_code")
        .To("436641234567")
        .DeliveryReceipt()
        .Text("test text")
);
```

Receipt format

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

```
id:IIIIIIIII sub:SSS dlvrd:DDD submit date:YYMMDDhhmm done date:YYMMDDhhmm stat:DDDDDDD err:E Text:
```

This text format is represented in the library as <xref:Inetlab.SMPP.Common.Receipt> class.

The class properties:

PROPERTY	DESCRIPTION
MessageId	The message ID allocated to the message by the SMSC when originally submitted. You can get it from <xref:Inetlab.SMPP.PDU.SubmitSmResp> or <xref: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 <xref:Inetlab.SMPP.PDU.SubmitMulti>.
Delivered	Number of short messages delivered to a 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 its 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 one of the following:

Possible message state

STATE	DESCRIPTION
Delivered	Message is delivered to the 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)
Unknown	Message is in invalid state
Rejected	Message is in a rejected state

> **NOTE**
>
> Library sends <xref:Inetlab.SMPP.PDU.DeliverSmResp> with status <xref:Inetlab.SMPP.Common.CommandStatus.ESME_RX_T_APPN> to SMPP server when <xref:Inetlab.SMPP.SmppClient.evDeliverSm> event handler method throws an exception.

How to tie submitted message with delivery receipt

SMS message in SMPP protocol is actually 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 <xref:Inetlab.SMPP.PDU.SubmitSm> or <xref:Inetlab.SMPP.PDU.SubmitMulti> PDU you need to assign <xref: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.SubmitAsync(list);

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

At the same time you need to store <xref:Inetlab.SMPP.Common.SmppHeader.Sequence> in the database. For one *message.Id* you need to store several <xref:Inetlab.SMPP.Common.SmppHeader.Sequence>.

In response to <xref:Inetlab.SMPP.PDU.SubmitSm> PDU your application receives <xref:Inetlab.SMPP.PDU.SubmitSmResp> PDU.

This response has the same <xref:Inetlab.SMPP.Common.SmppHeader.Sequence> number and <xref:Inetlab.SMPP.PDU.SubmitSmResp.MessageId> generated by the server.

When you receive a delivery receipt in the event <xref:Inetlab.SMPP.SmppClient.evDeliverSm>, the server sends same <xref:Inetlab.SMPP.Common.Receipt.MessageId> which you can use for updating status of the 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 <xref:Inetlab.SMPP.Common.MessageState.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

Reade more on page "Track message sending and delivery"

Enquire Link

<xref:Inetlab.SMPP.PDU.EnquireLink> is SMPP command allowing to check communication between ESME and SMSC.

The command can be sent by both client and server.

The EnquireLink mechanism assumes sending special SMPP-request by one of the peers obtaining the proper response. When proper response is received with <xref:Inetlab.SMPP.Common.CommandStatus.ESME_ROK> status, the connection is considered active. Otherwise, if there is wrong response or no response at all - the connection is to be closed.

To enable periodical connecton check, you need to set the following property:

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

<xref:Inetlab.SMPP.SmppClientBase.EnquireLinkInterval> specifies the time to wait after the last PDU exchange before sending the command. <xref:Inetlab.SMPP.PDU.EnquireLink> request won't be sent when client and server are sending PDUs.

Read more on page Keeping connection active (InactivityTimeout and EnquireLink)

Binary SMS

Binary SMS can be

- ringtone
- image
- WML/WBXML

Inetlab.SMPP supports MMS notifications and WAP push messages.

Sending binary SMS

You can create any binary data and send it with the library

```
byte[] data =
ByteArray.FromHexString("FFFF002830006609EC592F55DCE901000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000005C67");

var resp = await _client.SubmitAsync(
    SMS.ForSubmit()
        .From("short_code")
        .To("436641234567")
        .Data(data)
);
```

Some binary messages you may need to send on specific application port

for SubmitSm instance

```
submitSm.UserData.Headers.Add(new ApplicationPortAddressingScheme16bit(0x1579, 0x0000));
```

using SubmitSm SMS builder

```
builder.Set(sm =>
{
    sm.UserData.Headers.Add(new ApplicationPortAddressingScheme16bit(0x1579, 0x0000));
});
```

Receiving binary SMS

Receive single binary message

```

private void OnClientDeliverSm(object sender, DeliverSm data)
{
    byte[] messageBytes = data.GetMessageBytes();

    string hexString = messageBytes.ToHexString();
}

```

Receive concatenated binary message

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

private void WireEvents()
{
    _client.evDeliverSm += OnDeliverSm;

    _composer.evFullMessageReceived += OnFullMessageReceived;

}

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

private void OnFullMessageReceived(object sender, MessageEventHandlerArgs args)
{
    byte[] fullMessage = args.Parts.SelectMany(x => x.GetMessageBytes()).ToArray();
}
```

How to install the 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 a 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(GetType().Assembly.GetManifestResourceStream(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);
```

Creating a global and local logger

You can turn on globall logging to analyze operations performed by the InetLab.SMPP library.

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

It creates the global (available to other instances) logger and specifies logging mode at the **Debug** level. After that operation, all logging records associated with logging level **Debug** and less, will be echoed into the console.

Logging depth is defined by the following <xref:Inetlab.SMPP.Logging.LogLevel> values (by decreasing of outputted information amount):

- LogLevel.All
- LogLevel.Verbose
- LogLevel.Debug
- LogLevel.Info
- LogLevel.Warning
- LogLevel.Error
- LogLevel.Fatal
- LogLevel.Off

You can get a logger instance from any method and output your own records into it as well:

```
ILog log = LogManager.GetLogger("MyLogger");
log.Info("Connected to SMPP server");
```

As a result, it will output "Connected to SMPP server" into the log/console and mark it as LogLevel. <xref:Inetlab.SMPP.Logging.LogLevel.Info>.

To log a single instance you need to create logger and specify it as an instance parameter. For example, you can specify an individual (local) logger for <xref:Inetlab.SMPP.SmppClient> instance:

```
ConsoleLogger _log = new ConsoleLogger("MyClientLogger", LogLevel.Info);
SmppClient smppClient = new SmppClient();
smppClient.Logger = _log;
```

This makes logger _log to output data from the related instance only.

Read more about logging on "Common Tools: Built-in Logging" page.

Mapping DataCodings to .NET Encoding

For each <xref:Inetlab.SMPP.SmppClient> instance, you can define which Encoding will be used for specified <xref:Inetlab.SMPP.Common.DataCodings>.

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

By default <xref:Inetlab.SMPP.SmppClient> has the following <xref:Inetlab.SMPP.Common.DataCodings> to Encoding mappings:

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

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

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


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

> **NOTE**
>
> Before changing mapping settings, please clarify with your SMPP provider the encoding expected (character set for <xref: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 an ability to replace standard GSM 7 bit default alphabet table for the whole text (*Locking shift table*) or only extension table (*Single shift table*).

Code bellow shows abilities how you can specify desired character set:

```
await client.SubmitAsync(SMS.ForSubmit()
    .Text(text).From("5555").To(phone)
    .NationalLanguageLockingShift(NationalLanguage.Spanish)
);
```

or

```
submitSm.UserData.Headers.Add(new NationalLanguageLockingShift(NationalLanguage.Spanish));
```

The library is also able to detect national language User Data Header in the received PDU and to show text with the correct character set in property <xref:Inetlab.SMPP.PDU.SubmitSm.MessageText>.

Links

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

Message Composer: How to combine concatenated messages

SMS message with long text must be split into 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 <xref:Inetlab.SMPP.Common.MessageComposer> class.

<xref:Inetlab.SMPP.Common.MessageComposer> supports all types of PDUs: <xref:Inetlab.SMPP.PDU.SubmitSm>, <xref:Inetlab.SMPP.PDU.SubmitMulti>, <xref:Inetlab.SMPP.PDU.DeliverSm>.

You should invoke <xref:Inetlab.SMPP.Common.MessageComposer.AddMessage``1(``0)> method in each event handler for PDU received. <xref:Inetlab.SMPP.Common.MessageComposer> saves PDU in memory and waits for the last segment of the message text and raises <xref:Inetlab.SMPP.Common.MessageComposer.evFullMessageReceived> event.

When PDU has no concatenation parameters this event will be raised right after calling <xref:Inetlab.SMPP.Common.MessageComposer.AddMessage``1(``0)> method.

When <xref:Inetlab.SMPP.Common.MessageComposer> didn't receive last segment for a long time it raises <xref:Inetlab.SMPP.Common.MessageComposer.evFullMessageTimeout> event. Default timeout is 60 seconds.

```
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($"Incomplete message received from {pdu.SourceAddress}");
}

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

<xref:Inetlab.SMPP.Common.MessageComposer> also provides 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);
    }
}
```

Performance (TPS)

The achievement of Transactions Per Second (TPS) in SMPP processing is influenced by several factors, including hardware specifications, software efficiency, network conditions, and the specific implementation details of your application. Here are some key considerations:

1. Hardware Specifications:
    - CPU: Utilize a multi-core processor with a high clock speed to handle the often CPU-intensive SMPP processing efficiently.
    - Memory (RAM): Ensure sufficient RAM to manage message processing effectively. The SMPP client typically retains the request Protocol Data Unit (PDU) in memory until the appropriate response is received and processed.
2. Network Infrastructure:
    - Bandwidth: Ensure that the server has enough network bandwidth to handle the volume of messages you expect.
    - Latency: Low network latency is crucial for fast communication between the SMPP client and the SMPP server.
3. Application Implementation:
    - The efficiency of the software is critical. Ensure it is well-optimized to handle a high volume of transactions.
    - Fine-tune the configuration of the SMPP client to maximize performance.
    - Minimize the time taken for message preparation.
    - Determine the optimal message batch size.
    - Process requests and responses as quickly as possible.
4. Monitoring and Tuning:
    - Continuously monitor the application performance during testing and production.
    - Use monitoring tools to identify any bottlenecks and optimize configurations accordingly.
    - When starting, use SmppClientDemo application to estimate single SMPP session performance with a SMPP server.
5. SMPP Server:

- Consider that achieving high TPS may depend on the capabilities and limitations of the SMPP server you are connecting to and the overall SMS infrastructure.
- Use Metrics to measure the response time of the SMPP server, helping you analyze its performance.

Local Test

Performance testing of Inetlab.SMPP on the local machine with logging disabled shows the following result:

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

The following code demonstrates 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
{
    internal class Program
    {
```

```
        private 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*/ };
                await server.StartAsync();

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

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

                    int testResult = await RunTest(client, 50000);

                    Console.WriteLine("Performance: " + testResult + " 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.SubmitAsync(
                    SMS.ForSubmit()
                        .From("111")
                        .To("222")
                        .Coding(DataCodings.UCS2)
                        .Text("test")));
            }

            await Task.WhenAll(tasks);

            watch.Stop();

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

        }
    }
}
```

It's important to note that the performance test should always be started without an attached debugger, and the application should be built with the Release configuration.

The performance result obtained in the local test represents the maximal throughput achievable on the machine under the given conditions.

SMPP Server FAQ (with sample app)

How to send message to the connected client

In following code target client is selected and <xref:Inetlab.SMPP.PDU.DeliverSm> message is sent to this client.

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

    SmppServerClient client = FindClient(systemId);

    await client.DeliverAsync(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.DeliverAsync(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 the server receives <xref:Inetlab.SMPP.Common.CommandSet.SubmitSm> or <xref:Inetlab.SMPP.Common.CommandSet.SubmitMulti> PDU, it generates a corresponding response and sets MessageId.

You can change the MessageId property in <xref:Inetlab.SMPP.SmppServer.evClientSubmitSm> and <xref:Inetlab.SMPP.SmppServer.evClientSubmitMulti> event handlers.

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

Sample program link for the SMPP Server

Reade more about creating SMPP-server and Connect (with sample app).

SMPP Address

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

Address is a 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

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

Short number:

address: 55555, TON: 3, NPI: 0

Alphanumeric string:

address: MyService, TON: 5, NPI: 0

SSL/TLS Connection

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

For <xref:Inetlab.SMPP.SmppServer> class you can set server certificate and supported SSL/TLS protocols.

For <xref:Inetlab.SMPP.SmppClient> class you can specify supported SSL/TLS protocols, and optional client certificate for authentication.

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

    await server.StartAsync();

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

        var subject = client.ClientCertificate.Subject;

        var certificate = client.ClientCertificate as X509Certificate2;
        if (certificate != null)
        {
            bool isValid = certificate.Verify();
        }

    };

    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.ConnectAsync("localhost", 7777))
        {
            BindResp bindResp = await client.BindAsync("username", "password");

            if (bindResp.Header.Status == CommandStatus.ESME_ROK)
            {
                var submitResp = await client.SubmitAsync(
                    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.DisconnectAsync();
        }
    }
}
```

SubmitMulti. Send message to multiple destinations

The <xref: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 the method <xref:Inetlab.SMPP.Builders.IDestinationAddressBuilder%601.To%2A>

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

this can be done from the 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 <xref: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.SubmitAsync(SMS.ForSubmitMulti()
    .ServiceType("test")
    .Text("Test Test")
    .From("MyService")
    .ToDestinations(destList)
);
```

When <xref:Inetlab.SMPP.PDU.SubmitMultiResp> response received, it means SMPP server stored message for further delivery to recipients.

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

MMS notifications

MMS Notification

After creating a MMS content and placing it on a web server, you can tell MMS compatible mobile phone to retrieve the content.

```
var smList = MMS.Notification(new MMSNotification
{
    TransactionId = "NOK5CyhldCAWF",
    From = "+79171234567/TYPE=PLMN",
    MessageClass = MMSMessageClass.Personal,
    MessageSize = 858,
    Expiry = TimeSpan.FromDays(30),
    ContentLocation = "http://test.com/resources/NOK5CyhldCAWFygAxJ"
})
    .TransactionId(0x41);

await _client.SubmitAsync(smList);
```

MMS Delivery

When MMS Content was received from another MMS client, you can notify sender that receipient retrieved the content.

```
var smList = MMS.Delivery(new MMSDelivery
{
    MessageId = "041310322550000000004",
    To = "+79171234567/TYPE=PLMN",
    Date = new DateTime(2020, 4, 13, 12, 32, 40),
    Status = MMSStatus.Retrieved
})
    .TransactionId(0x42);

await _client.SubmitAsync(smList);
```

WAP Push

WAP PUSH enables the delivery of multimedia content to mobile devices. A hyperlink is delivered to the mobile device of the recipient as a Service Indication message. The recipient mobile device will automatically prompt the recipient to download the content on the hyperlink and display it over a GPRS/WAP connection.

```
byte transactionId = 1;

var responses = await _client.SubmitAsync(SMS.ForWapPush()
    .TransactionId(transactionId)
    .From("1111").To("60166609999")
    .Title("Google").Url("http://www.google.com")
);
```

Multiple Client binds to SMPP servers

The SmppClient represents a single session to the SMPP server. When you need to bind to multiple SMPP servers or when you need to establish several sessions to a SMPP server, you have to create several instances of SmppClient class and attach event handler methods to all instances.

```
public class SmppRouterSample
{
    private readonly List<SmppClient> _sessions = new List<SmppClient>();

    /// <summary>
    /// The SMPP sessions
    /// </summary>
    public IReadOnlyList<SmppClient> Sessions => _sessions;


    /// <summary>
    /// Add new session with SMPP provider
    /// </summary>
    /// <param name="endPoint">The endpoint of the SMPP provider.</param>
    /// <param name="systemId">The System ID for the Bind command.</param>
    /// <param name="password">The Password for the Bind command.</param>
    /// <param name="systemType">The type of ESME system</param>
    public async Task AddSession(EndPoint endPoint, string systemId, string password, string systemType =
null)
    {
        SmppClient client = new SmppClient();
        client.ConnectionRecovery = true;
        client.EnquireLinkInterval = TimeSpan.FromSeconds(30);
        client.SystemType = systemType;

        AttachEvents(client);
        await client.RetryUntilConnectedAsync(endPoint, TimeSpan.FromSeconds(5));

        BindResp resp = await client.BindAsync(systemId, password);
        if (resp.Header.Status != CommandStatus.ESME_ROK)
        {
            throw new InvalidOperationException($"Cannot bind to {endPoint}. Bind Status:
{resp.Header.Status} ");
        }

        _sessions.Add(client);
    }

    /// <summary>
    /// Send SubmitSm PDU to the selected session
    /// </summary>
    /// <param name="submitSm">The SUBMIT_SM PDU</param>
    /// <param name="sessionSelector">The function to select a SMPP session.</param>
    /// <returns></returns>
    public Task<SubmitSmResp> SubmitAsync(SubmitSm submitSm, Func<SmppClient,bool> sessionSelector)
    {
        SmppClient client = Sessions.FirstOrDefault(sessionSelector);

        if (client == null)
        {
            throw new InvalidOperationException("The SMPP session is not found for the message");
        }

        return client.SubmitAsync(submitSm);
    }


    /// <summary>
```

```
    /// Send SubmitSm PDU to the selected session
    /// </summary>
    /// <param name="builder">The SMS builder</param>
    /// <param name="sessionSelector">The function to select a SMPP session.</param>
    public Task<SubmitSmResp[]> SubmitAsync(IBuilder<SubmitSm> builder, Func<SmppClient, bool>
sessionSelector)
    {
        SmppClient client = Sessions.FirstOrDefault(sessionSelector);

        if (client == null)
        {
            throw new InvalidOperationException("The SMPP session is not found for the message");
        }

        return client.SubmitAsync(builder);
    }


    private void AttachEvents(SmppClient smppClient)
    {
        smppClient.evDeliverSm += OnDeliverSm;
        smppClient.evDataSm += OnDataSm;
        smppClient.evUnBind += OnUnbind;
    }

    private void DetachEvents(SmppClient smppClient)
    {
        smppClient.evDeliverSm -= OnDeliverSm;
        smppClient.evDataSm -= OnDataSm;
        smppClient.evUnBind -= OnUnbind;
    }

    private void OnDataSm(object sender, DataSm data)
    {
        SmppClient client = (SmppClient)sender;
        Console.WriteLine($"DATA_SM received from the session {client.RemoteEndPoint}/{client.SystemID}");
    }


    private void OnDeliverSm(object sender, DeliverSm deliverSm)
    {
        SmppClient client = (SmppClient)sender;

        if (deliverSm.MessageType == MessageTypes.SMSCDeliveryReceipt)
        {
            Console.WriteLine($"Delivery Receipt received from the session
{client.RemoteEndPoint}/{client.SystemID}");
        }
        else
        {
            Console.WriteLine($"Incoming SMS received from the session
{client.RemoteEndPoint}/{client.SystemID}");
        }
    }

    private void OnUnbind(object sender, UnBind data)
    {
        SmppClient client = (SmppClient)sender;

        Console.WriteLine($"Unbind the session {client.RemoteEndPoint}/{client.SystemID}");

        DetachEvents(client);

        _sessions.Remove(client);
    }
```

```
}
```

The sample class can be used like this code:

```
SmppRouterSample router = new SmppRouterSample();
await router.AddSession(new DnsEndPoint("smpp.server.net", 7777), "test", "test");


SubmitSmResp[] resp = await router.SubmitAsync(SMS.ForSubmit().From("SERVICE").To("123456789").Text("my
message"), c=>c.SystemID == "test");
```

Implementing USSD (Unstructured Supplementary Service Data)

Overview

The USSD session can be initiated by the Mobile Station (MS) or an External Short Message Entity (ESME). The USSD messages create a real-time connection during a USSD session. The connection remains open, allowing a two-way exchange of a sequence of data.

This makes USSD more responsive than services that use SMS.

USSD can be used to provide:

- enhance mobile marketing capabilities
- menu-based information services
- interactive data services
- mobile-money services
- location-based content services
- callback service (to reduce phone charges while roaming)
- configuring the phone on the network

Requirements

USSD over SMPP solution is always vendor specific and requires service description from your SMPP provider or mobile network operator. We can help you to implement this with Inetlab.SMPP library, but you need to send us this description.

Types of USSD messages

- Set up a session: Begin message
- Continue a session: Continue message
- End a session: End message
- Abort a session: Abort message

Types of USSD operations

The USSD session involves the following operations:

Request: to request a session

- If the session is initiated by an MS, the MS can only send a Request message in the first message. In the subsequent message exchange, the MS can only respond to the Request or Notify message from an ESME.
- If the session is initiated by an ESME, the Request message can be sent by only the ESME, while the MS can only respond to the message.

Notify: to notify of a session

The Notify message can be sent by only an ESME. The Notify message differs from the Request message in that an MS responds to a Notify message automatically, but the response to a Request message can only be done manually. For example, send a response character string.

Response: to respond to a session

The response to a Request or Notify message can be sent by the MS or an ESME. When the ESME sends a Response message, it indicates the end of a session.

Release: to release a session

When an ESME ends a session initiated by itself, the operation type can only be Release.

Getting Help

To get support please contact us at InetLab website contact page or via our forum.

Migration from v1.x to 2.x

How to solve some compile issues:

SmppClient

1) The events `evBindComplete`, `evSubmitComplete`, `evQueryComplete` have been deprecated. You should use async await pattern with the methods <xref:Inetlab.SMPP.SmppClient.BindAsync*>, <xref:Inetlab.SMPP.SmppClient.SubmitAsync*>, <xref:Inetlab.SMPP.SmppClient.QueryAsync*>.

```
public async Task SubmitBatchAsync(List<SubmitSm> batch)
{
    IEnumerable<SubmitSmResp> responses = await _client.SubmitAsync(batch).ConfigureAwait(false);
}
```

or handle the response with ContinueWith

```
_client.SubmitAsync(submitSm).ContinueWith(t =>
{

    if (t.IsFaulted)
    {
        var error = t.Exception;
    }
    else
    {
        //get SUBMIT_SM_RESP
        SubmitSmResp resp = t.Result;

    }

});
```

2) Missing `BatchMonitor` class. Use instead <xref:Inetlab.SMPP.SmppClient.SubmitAsync(System.Collections.Generic.IEnumerable{Inetlab.SMPP.PDU.SubmitSm})>. It waits until all responses for the batch are received.

> **TIP**
>
> Awaiting for each single SubmtiSmResp can reduce the *performance*. You can create a batch of SubmitSm PDUs and send all of them with one <xref:Inetlab.SMPP.SmppClient.SubmitAsync(System.Collections.Generic.IEnumerable{Inetlab.SMPP.PDU.SubmitSm})> call.

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 <xref:Inetlab.SMPP.PDU.SmppPDU.Header> property of the 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 <xref:Inetlab.SMPP.PDU.SubmitSm.SourceAddress> of type <xref:Inetlab.SMPP.Common.SmeAddress>

- `DestAddrTon`, `DestAddrNpi`, `DestAddr` replaced with <xref:Inetlab.SMPP.PDU.SubmitSm.DestinationAddress> of type <xref:Inetlab.SMPP.Common.SmeAddress>
- `UserDataPdu` replaced with <xref:Inetlab.SMPP.PDU.SubmitSm.UserData>
- `Optional` replaced with <xref:Inetlab.SMPP.PDU.SubmitSm.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 = ReadFromDataReader();

SubmitSm submitSm = pduData.Deserialize<SubmitSm>(_client.EncodingMapper);
```

Report a Bug

To report a bug please contact us at InetLab website contact page or via our forum.

Changelog

[2.9.32] - 2024-01-26

Changed

- The extension method SmppClient.SubmitWithRepeatAsync can now queue a batch of SubmitSm requests regardless of the SMPP session state. The queued requests will be sent as soon as the SMPP session is established or restored.

[2.9.31] - 2024-01-16

Fixed

- The DisconnectAsync method in the SmppClient is experiencing a hanging issue within the .NET Framework.

[2.9.30] - 2023-10-27

Added

- Add extension method GetMessageBytes for SmppPdu classes.

Fixed

- Recover the features of SendQueueLimit and ReceivedRequestQueueLimit. The thread should be blocked to prevent excessive memory consumption.

[2.9.29] - 2023-09-12

Added

- National language shift tables for Portuguese language

[2.9.28] - 2023-08-02

Added

- support of .NET 6.0
- new telemetry feature for monitoring and analysis of SMPP operations. Only for applications targeting .NET 6.0 or later.

Fixed

- SubmitAsync may experience a hang when SendSpeedLimit is utilized and the session becomes disconnected.
- Memory leak occurs when suppressing the response in the request handler and then sending a new response using the SendResponseAsync method.

[2.9.27] - 2023-04-03

Improved

- don't block threads when using SendQueueLimit or ReceivedRequestQueueLimit

Fixed

- throw exception with SocketError.Timeout instead of SocketError.OperationAborted from ConnectAsync when connection timeout

[2.9.26] - 2022-11-25

Improved

- process responses faster and prevent SMPPCLIENT_RCVTIMEOUT

Changed

- when ReceivedRequestQueueLimit is specified, a free slot for the request will be added when response is sent

[2.9.25] - 2022-11-13

Improved

- RetryUntilBindAsync extension method. Retries bind when ESME_RALYBND is received.

Fixed

- Prevent ObjectDisposedException when request is being sent and SMPP session is closed. SMPPCLIENT_NOCONN response status is returned instead.

[2.9.24] - 2022-10-03

Fixed

- ResponseTimeout set after bind is not saved for the later session created from the SmppClient instance.

[2.9.23] - 2022-09-28

Added

- ActualEndpoint property in SmppServer class. Gets the actual endpoint after server start. ### Fixed
- ResponseTimeout not respected if set after connect

[2.9.22] - 2022-08-29

Fixed

- GsmPackedEncoding. Use CR control as a padding filler.

[2.9.21] - 2022-08-18

Fixed

- DeliverSm with Receipt encoded with UCS2 data_coding in the message_payload

[2.9.20] - 2022-08-10

Fixed

- Memory leak around TCP connections

[2.9.19] - 2022-06-06

Changed

- Initiate TLS connection only when SmppConnectionOptions.Ssl property is defined.

[2.9.18] - 2022-05-19

Added

- SmppOutbindClient class that can establish SMPP session with ESME using Outbind command.
- SmppOutbindServerClient class, that passed down to the evClientOutBind event in the SmppServer class.
- SmppOutbindServerClient has evDeliverSm event that can be used to receive outstanding messages from SMSC.

[2.9.17] - 2022-04-12

Added

- Extension method SmppClient.RetryUntilBindAsync

[2.9.16] - 2022-01-20

Added

- Response time metrics

Fixed

- ReceivedRequestQueueLimit doesn't work
- sometimes SubmitAsync hangs when the server drops the connection

[2.9.15] - 2022-01-11

Changed

- SmppServer is now sealed class
- Remove unused CancellationToken parameter from SmppServer.StopAsync method
- default SmppClientBase.ReceiveTaskScheduler to TaskScheduler.Default

Improved

- Performance for developer version

Fixed

- several warnings in the code

[2.9.14] - 2021-08-31

Fixed

- NullReferenceException warning on accepting connection when server disconnects the newly connected client.
- SmppClientBase.Dispose doesn't free internal objects when connection is already closed.

[2.9.13] - 2021-07-30

Fixed

- endless repetition of sending in the SubmitWithRepeatAsync extension method

[2.9.12] - 2021-07-01

Fixed

- unnecessary delay on success submit in the SubmitWithRepeatAsync extension method
- The metric client.Metrics.Sent.Requests.PerSecond reports incorrect results.

[2.9.11] - 2021-06-24

Fixed

- When client disconnects the PDU packets that already received from network stream must be processed.
- Better logging for disconnect routine
- When client receives corrupted response PDU with readable SMPP Header and non-ok status, return this response in corresponding method and write the warning into the log.

[2.9.10] - 2021-06-12

Fixed

- SmppClient. memory leak in SMPP connection.
- SmppClient. Deadlock on disconnection in Windows Forms application for in .NET Framework.

[2.9.9] - 2021-05-13

Fixed

- Performance. Default buffer size was 4096 bytes. This significantly reduces the throughput. Now default buffer size is 65536 for send and receive.
- SmppServer. SendBufferSize and ReceiveBufferSize properties changed in SmppServerClient in evClientConnected event handler are not applied to Socket buffer size.
- SmppServer. SmppServer may hang when stopped.
- SmppServer. Missing binds after failed accept connection.
- SmppClient. When remote side drops the connection, the response status SMPPCLIENT_RCVTIMEOUT maybe returned instead of SMPPCLIENT_NOCONN.
- SmppClientBase. Don't disconnect when remote side doesn't response on our EnquiryLink request, but send any other request PDU.

[2.9.8] - 2021-04-28

Fixed

- wrong User Data Length with 16-Bit concatenation

[2.9.7] - 2021-04-12

Fixed

- Developer License. Unable to cast object of type 'System.Security.Cryptography.RSACng' to type 'System.Security.Cryptography.RSACryptoServiceProvider'

[2.9.6] - 2021-04-09

Added

- Tag property to SmppClientBase class to associate any data with the client.

[2.9.5] - 2021-03-18

Added

- SmppClient.SubmitWithRepeatAsync extension method for sending single SubmitSm.
- SmppClient.ConnectAsync(SmppConnectionOptions) method with extended connection options
- SmppConnectionOptions.SinglePDUinTCPPacket allows to send each PDU in single TCP packet
- new async methods in SmppServer class: StartAsync, StopAsync, RunAsync

Fixed

- SmppServerClient.Status is Closed in the event SmppServer.evClientConnected
- InvalidOperationException on SmppServer.Stop

[2.9.4] - 2021-02-06

Fixed

- Incorrect length of ShortMessage for the first part in the concatenated message when using NationalLanguageSingleShift user data header.

[2.9.3] - 2021-02-04

Fixed

- Delivery Receipt parsing

[2.9.2] - 2021-02-01

Fixed

- Incorrect length of ShortMessage for concatenated message part when using NationalLanguageSingleShift user data header.

[2.9.1] - 2021-01-29

Fixed

- SmppServerClient.RemoteEndPoint is null when client connection is accepted.

[2.9.0] - 2021-01-25

Added

- MMS m-notification-ind and m-delivery-ind over SMPP
- WorkersTaskScheduler class creates worker threads to handle received requests for a client.
- ReceiveTaskScheduler property in SmppClientBase class. Received requests can be handled in individual or global TaskScheduller or in standard TaskScheduler.Default. By default client uses WorkersTaskScheduler with 3 worker threads.
- RetryUntilConnectedAsync extension method for SmppClient class that helps to establish connection when a SMPP server is temporary unreachable.
- SubmitBatchAndWaitForDeliveryAsync extension method for SmppClient class that helps to submit a batch of SubmitSm PDUs and receive all delivery receipts.
- SubmitWithRepeatAsync extension method that help to repeat the SubmitSm requests on disconnection or unsuccess reponse status.

Changed

- Target Framework changed from .NET 4.5.2 to .NET 4.6.1
- SmppClientBase.SendQueueLimit limits a number of requests waiting for response. When this limit is exceeded, SubmitAsync method is blocked until queue has a free slot.
- Less GC pressure by reusing reading buffer.
- Improve DeliverReceipt deserialization when received PDU has wrong DataCoding.
- Deliver receipt serializer enriches DeliverSm PDU with optional parameters (MessageState, ReceiptedMessageId, NetworkErrorCode) only when PDU was not received from remote side.
- Throws GenericNackSmppException when GENERICK_NACK response received in Submit methods.
- Throws ObjectDisposedException when trying to call public method on disposed classes SmppClientBase, SmppClient, SmppServerClient
- New implementation of SMPP connection that frees all used resources after disconnect.
- Improved batch submit. Send several PDUs in one TCP packet.

Fixed

- ArgumentNullException in MessageComposer by adding a PDU with empty text
- SmppClientBase.SendSpeedLimit sends more than allowed PDUs when run SubmitAsync in tasks.
- GSMEncoding returns wrong number of bytes for Turkish charset when both NationalLanguageLockingShift and NationalLanguageSingleShift are specified.
- Text splitting. The maximum length of the short_message field is not used if NationalLanguageSingleShift is enabled.

Removed

- evConnected event from SmppClientBase class. SmppClient is connected after call of the method ConnectAsync or on the event evRecoverySucceeded. SmppServer has own method evClientConnected.
- Obsolete properties from PDU classes.
- QueueState class and SmppClientBase.Queue property. Use SmppClientBase.Metrics instead.
- SmppClientBase.WorkerThreads property. Set the SmppClientBase.ReceiveTaskScheduler property to the instance new WorkersTaskScheduler(3).

[2.8.6] - 2020-11-19

Fixed

- CancelSm serialization

[2.8.5] - 2020-11-12

Fixed

- NationalLanguageLockingShift user data header

[2.8.4] - 2020-11-12

Breaking Changes

- license file Inetlab.SMPP.license must contain product name. Generate new license file in your Inetlab Account. ### Fixed
- write license loading errors in license status.

[2.8.3] - 2020-11-04

Fixed

- failed to verify license on macos

[2.8.2] - 2020-09-01

Fixed

- inaccurate SendSpeedLimit on the machine with high-resolution performance counter.

[2.8.1] - 2020-06-16

Fixed

- SendSpeedLimit and result speed has significant difference.
- Send Queue can be blocked in some edge cases.
- Reconnect doesn't start when ThreadPool is overloaded
- Race condition in MessageComposer causes NullReferenceException

[2.8.0] - 2020-04-02

Added:

- SendResponseAsync method in SmppClientBase class. Response sending can be prevented in a event handler by changing it to null. req.Response = null;
- Extension method CanBeEncoded to validate an Encoding for given text message
- MessageComposer with persistence storage interface to save message parts in external database instead of memory.

Fixed:

- NullReferenceException in the event evFullMessageReceived of MessageComposer class
- Setup Project: Unable to update the dependencies of the project. The dependencies for the object 'Inetlab.SMPP.dll' cannot

be determined.

[2.7.1] - 2020-01-20

Changed

- move InterfaceVersion property to the SmppClientBase class

Fixed:

- GenericNack PDU has not been sent when wrong PDU header is received.

[2.7.0] - 2019-12-11

Added

- Added Metrics property for SmppClientBase class.
- Support of 16 bit concatenation parameters in SMS builder classes.

Changed

- ReceiveSpeedLimit with rate limiting. Measure PDU count for defined time unit instead of interval between PDUs.
- rename async-methods according to the dotnet naming conventions
- InactivityTimeout starts when SmppServerClient is connected and EnquireLinkInterval is not defined for this client.
- Generate long number MessageId for SubmitSmResp and SubmitMultiResp. According to SMPP Protocol MessageId should contain only digits.
- Timeout timer in MessageComposer restarts when next segment of the message is received.

Fixed

- Submit hangs after unexpected disconnect.
- Exception by changing send or receive buffer size.
- SmppTime.Format for relative time.
- OverflowException in GSMEncoding.

Removed

- Support of .NET Standard 1.4
- InactivityTimeout from SmppClient

[2.6.14] - 2019-09-20

Fixed

- SmppTime.Format for relative time.
- exception in demo applications

[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

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.

I 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.

LICENSE TO USE SOFTWARE.

We provide the following types of licenses under the terms and conditions as set forth below. Each type of License must be obtained or purchased separately.

1. DEVELOPER LICENSE. If you purchase this type of License, we grant you a non-exclusive, limited Developer License under the following terms and conditions.

   With this type of License the Software can be used by individual developers or smaller teams of up to 3 developers. With this type of License the Software can be used to develop an unlimited number of applications.

   This type of License includes the provision of support and software updates from us, over a one-year period.

2. SOURCE CODE LICENSE. If you purchase this type of License, we grant you a non-exclusive, limited Source Code License under the following terms and conditions.

   With this type of License, Source Code can be used by companies and an unlimited number of developers within the company. With this type of License the Source Code can be used to develop an unlimited number of applications.

   This type of License includes full C# source code for the Software. You are allowed to make custom modification to satisfy your specific needs.

   This type of License includes the provision of priority support and updates from us, over a one-year period.

   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.

3. TRIAL LICENSE. Licensee may use the Software for a Trial and for such purposes the InetLab e.U. grants Licensee a free, limited, non-exclusive Trial License under the following terms and conditions.

   This type of License is for individual developers only and the Software may be used only for private evaluation and testing of functionality. The Software may not be published in any internet nor intranet project until an appropriate license is purchased.

II 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

3. 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.

4. 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.

III 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.

IV 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.

V 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.

VI 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.

VII 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.

VIII 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.

IX 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.

X 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.

XI 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.