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

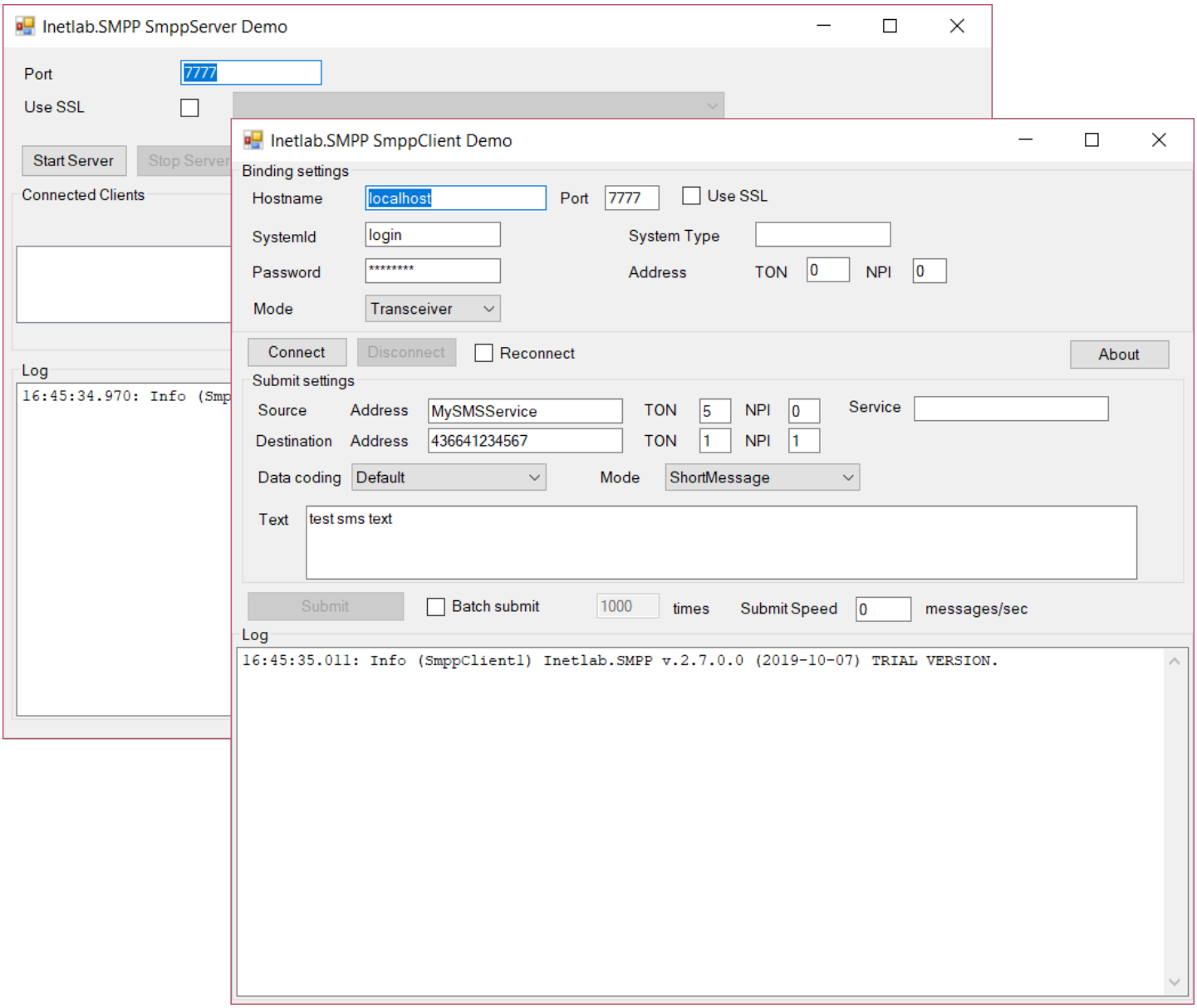
Get samples

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.

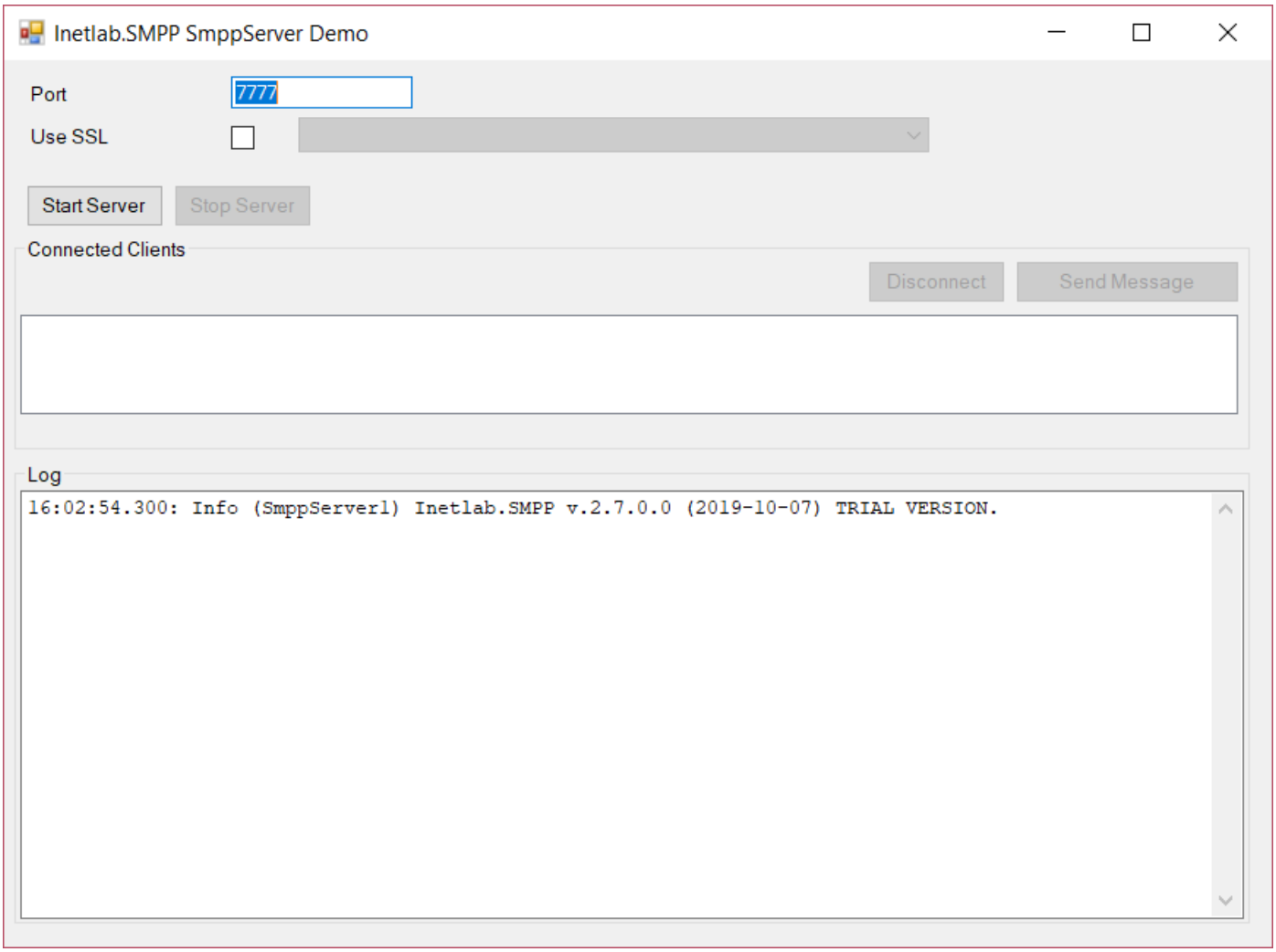
Start Demo Apps

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

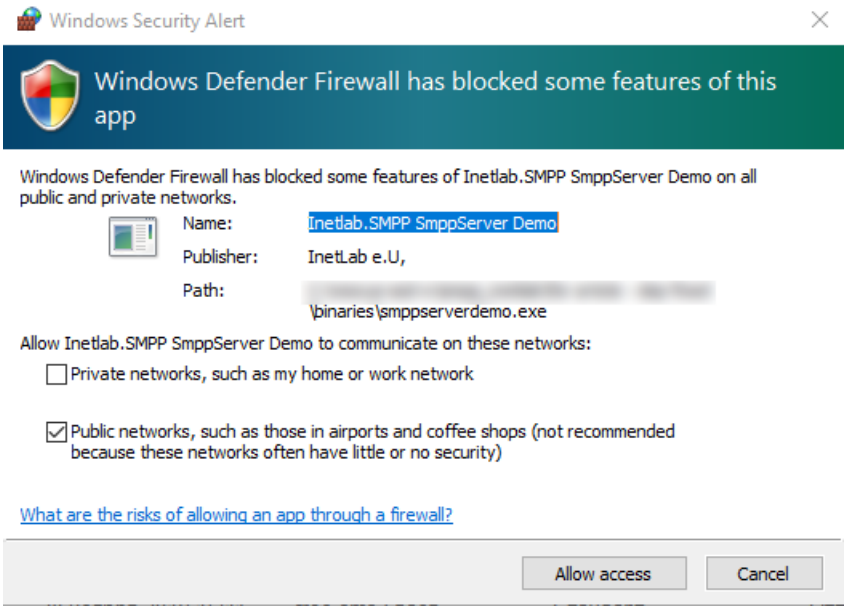
Unpack ZIP and run file run_demo.bat in “smpp-samples-master” folder. In a console window appeared you might see the question – respond with “Y” for starting samples compilation. After this (and further launches) of run_demo.bat you should see two demo-applications started: "Inetlab.SMPP SmppServer Demo"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

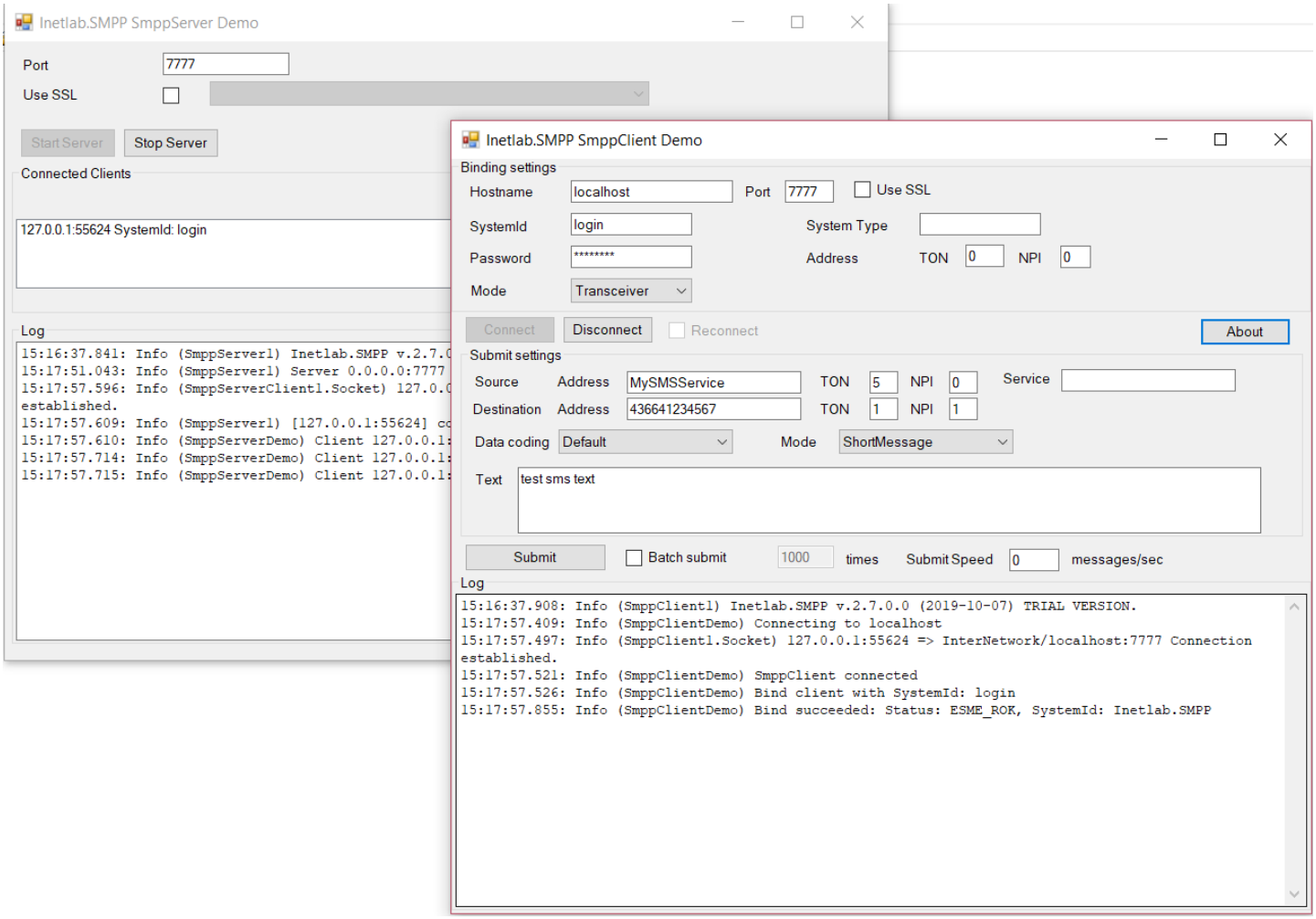
Showcases

Connect "Inetlab.SMPP SmppClient Demo" to "Inetlab.SMPP 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:

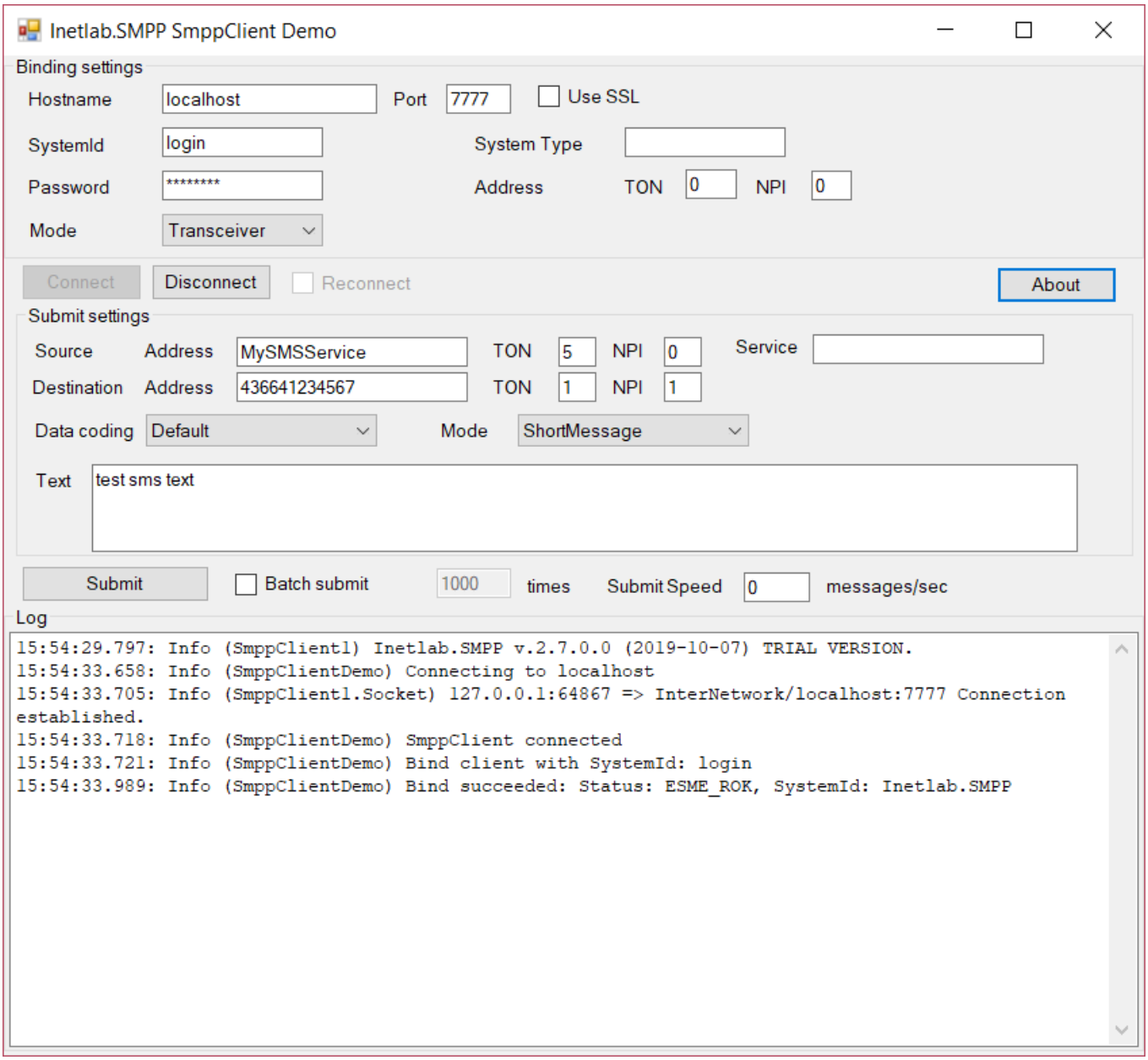
[timestamp]: Info (SmppServerDemo) Client 127.0.0.1:53233 sends message From:MySMSService, To:436641234567, Text: это тестовое sms[TRIAL] [timestamp]: Info (SmppServerDemo) SMS Received: это тестовое sms[TRIAL]

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.

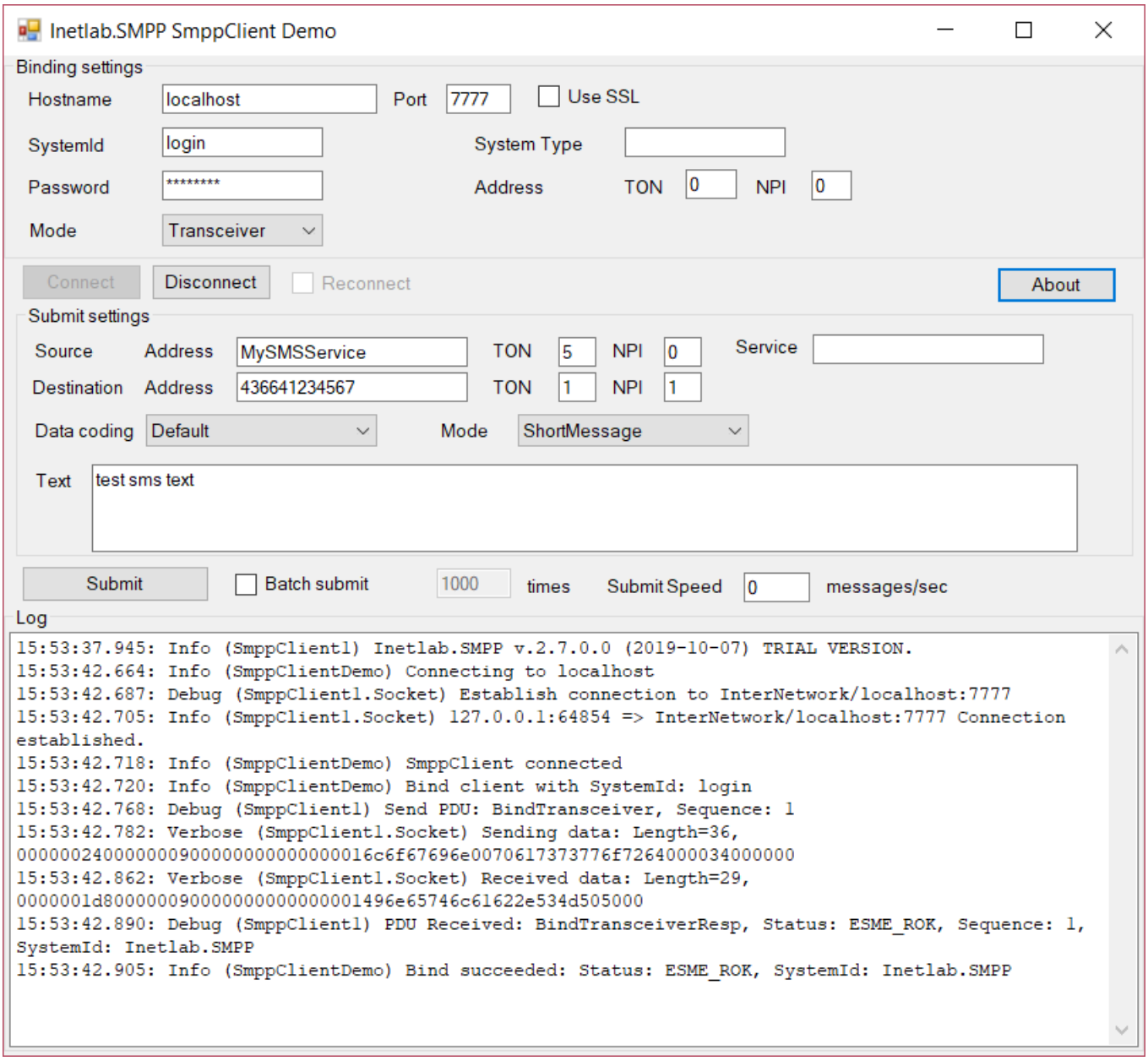
You can read more about text encoding in the article Map Encoding.

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").



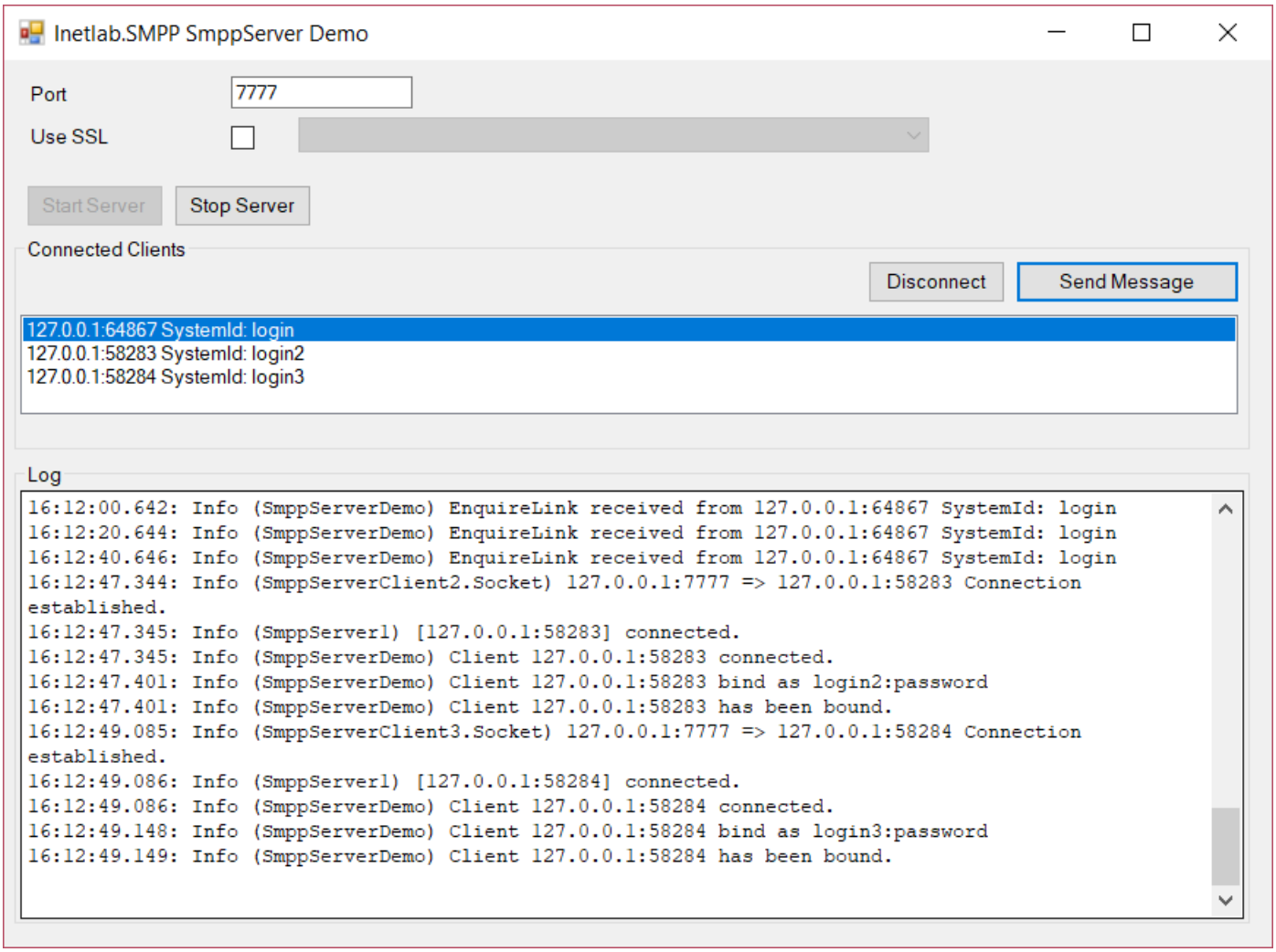
To change logging level it is necessary to change logging settings in the source code of SMPP-client and compile the project again. For example, by setting "Verbose" level in logger settings (showing much more technical details when “Info”) you will get more information in the logger output. Log-field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



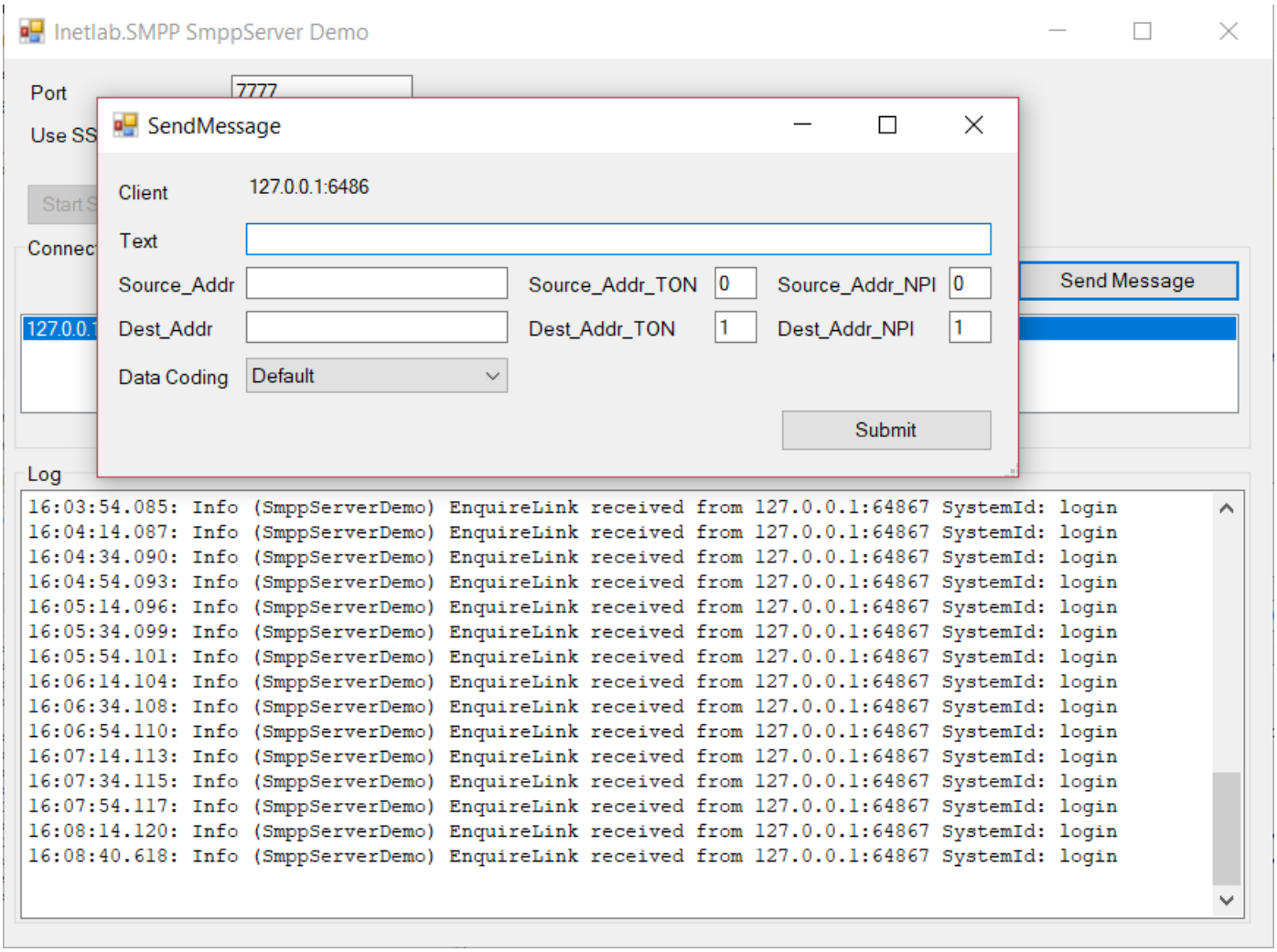
You can read more logging and logging levels in the article Creating local and global Logger.

Send a message from server to client selected

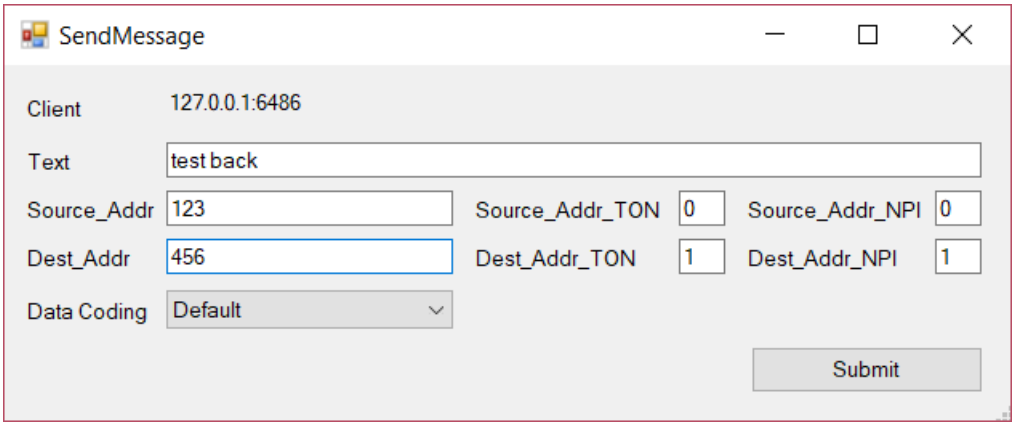
There is a list of all SMPP-clients connected and their respective logins in the “Connected Clients” field of "Inetlab.SMPP SmppServer Demo" window.



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 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 BindAsync. In BindAsync method you can also specify ConnectionMode. When calling BindAsync the third parameter (Connection Mode) is optional and, if not specified, by default it is Transceiver.

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

Calls to methods ConnectAsync and 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 BindAsync execution, in particular the server response and status.

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

ESME_ROK status confirms successful execution of authentication command and means you can proceed with sending and/or receiving messages. The SmppClient object will change its Status to Bound.

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

Connection recovery

Connection recovery can be activated with ConnectionRecovery property.

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

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

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

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

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

The delay time between recovery attempts can be changed with the property ConnectionRecoveryDelay. Default is 2 minutes.

Note

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

Create and send messages

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

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

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<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 Inetlab.SMPP.PDU messages and set properties not supported by SMS class this method is very useful.

Actual message transmission is performed by calling method SubmitAsync and passing either an argument of `SubmitSm PDU` either arrays/collections using method overloads.

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

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

Please note, the order of 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 SubmitSm on server side produces server response containing status `response.Header.Status = ESME_ROK`.

It is possible that several SubmitSmResp will produce a response with error status:

- SMPPCLIENT_NOCONN - connection failed during sending attempt.
- SMPPCLIENT_UNBOUND - you are probably trying to send commands via SmppClient without authentication (Bind).
- SMPPCLIENT_RCVTIMEOUT - response to request is not arrived during certain time.

Receive messages

An SMPP-server sends SMS to SMPP-client by using command DeliverSm. It may contain inbound SMS as well as delivery report.

There is an event evDeliverSm in class SmppClient. The event rises on 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 MessageComposer is used for that.

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

Track message sending and delivery

Track sending

The SMPP-server generates unique `MessageId` for each `SubmitSm PDU` received from SMPP-client. In case of a multipart message (long message split into parts) each part is assigned with unique `MessageId` generated by the server.

The client receives information about MessageId as `PDU SubmitSmResp` response from SMPP-server. The fact of issuing MessageId for message/part means server accepted it for further processing. You can read more about ways of collecting server responses SubmitSmResp in the article Create and sendmessasge.

Track delivery

During message creation it is necessary to set property "registered delivery" in order to get delivery reports after sending. The field `MessageId` is present in each delivery report and matches `MessageId` of the original message/part send before. Matching of delivery reports to original messages is possible by using MessageId and looking for identical values. This way is possible to track delivery progress.

There is a simple handler for evDeliverSm event in the following example. It extracts `MessageId` and status from received delivery report DeliverSm. The next thing to do is to search this `MessageId` in messages sent before.

...

```
_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));
_server.Start();
```

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 SmppServerClient class and raises the event evClientConnected for each TCP/IP client connected.

All clients connected are added automatically to the collection ConnectedClients of SmppServer object.

You may implement any preliminary checks in the event-handler subscribed to 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 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 BindResp to the SMPP-client containing ESME_RBINDFAIL status.

It is common to implement various authentication rules, login checks and other security checks in the event-handler subscribed to 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 InactivityTimeout with default value of 2 minutes for SmppServerClient instances. The SmppServer closes connection to clients based on this timer. It is possible to disable InactivityTimeout by assigning it value **TimeSpan.Zero**.

Example of setting 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);
};

_server.Start();
```

InactivityTimeout is possible to set inside event-handler for 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 SmppServerClient or SmppClient using property EnquireLinkInterval. The 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.Start();
_server.evClientBind += (s, client, bind) => {
    client.EnquireLinkInterval = TimeSpan.FromSeconds(15);
};
```

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

An automatic connection check is started only after successful **Bind**. Client without **Bind** considered inactive and will be disconnected if InactivityTimeout was set.

Please note, values InactivityTimeout and EnquireLinkInterval are to be set for each instance of SmppServerClient created i.e. for each client connected.

The event handler for event evClientDisconnected is called when client disconnects.

Receive messages

To receive messages it is necessary to create an event handler for evClientSubmitSm event. The evClientSubmitSm event is raised each time packet with 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 SubmitSmResp packet and put the 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 SubmitSmResp object.

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

If there is no event handler attached to evClientSubmitSm event, the server sends to client the response SubmitSmResp containing status 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 evClientSubmitSm event.

Send messages

The SMPP-server creates SmppServerClient object automatically for each SMPP-client connected. Calling method DeliverAsync of the SmppServerClient object sends a message to the respective SMPP-client.

To start, it is necessary to choose SmppServerClient instance from the list available at ConnectedClients property. You may use any 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 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".

```
static void 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);

    _server.Start();

    Console.ReadLine();
}

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 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 (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 SmppServer to be able to receive SMPP commands on a TCP port and several SmppClient instances to send message to other SMPP servers (SMSC, Provider).

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

This SMSC MessageId should also be replaced in DeliverSm (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 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 Sequence number to the database. Possible good idea to save entire PDU.
- Send SubmitSmResp to client with his Sequence number and MessageId generated on your server side.
- In another process/thread send this SubmitSm PDU to some SMPP provider.
- Change SubmitSm Sequence number to the next sequence number for the SmppClient that connected to that SMPP provider.
- Receive Provider's MessageId in SubmitSmResp
- Store Provider's MessageId and Sequence number to the same database table as for client's 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 DeliverSm comes from the provider and contains "DeliveryReceipt", you should do the following steps:

- Get Provider's MessageId from delivery Receipt.
- Find client's MessageId and corresponding SMPP user.
- Replace Provider's MessageId in DeliverSm PDU with client's MessageId
- Send "DeliveryReceipt" to the SmppServerClient that belongs to SMPP user.
- If there is no active connection with the client, place 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"

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 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

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.

Throttling error

SMSC can limit number of submitted PDU for SMPP account. When allowed message limit is exceeded, server returns status ESME_RTHROTTLED.

To avoid throttling error you can specify a number of messages per second in SmppClient. For this purpose you can define 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;
```

Common Tools: 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 ILog and ILogFactory interfaces. You can implement this interface with any kind of logging framework for your solution.

For example:

- NLog
- Log4Net

The library provides ConsoleLogFactory and FileLogFactory classes.

When the application starts you need to register global ILogFactory for the library.

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

or you can set Logger property when you create instances of SmppClient, SmppServerClient or 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 ILog and ILogFactory interfaces:

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

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

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

public class ConsoleLogger : ILog
{
    private readonly LogLevel _minLevel;

    public string Name { get; private set; }

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

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

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

```
        int threadId = Environment.CurrentManagedThreadId;

        string text = message;



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


        Console.WriteLine(sb.ToString());

    }



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

            case LogLevel.Error:
                return "ERROR";

            case LogLevel.Warning:
                return "WARN ";

            case LogLevel.Info:
                return "INFO ";

            case LogLevel.Debug:
                return "DEBUG";

            case LogLevel.Verbose:
                return "TRACE";

        }

        return "";
    }



}
```

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

Common Tools: Special Events and Metics

Special events

You can use special events in base class SmppClientBase for tracking PDUs:

- with the event evPduReceiving you can monitor all incoming PDUs.
- event evPduSending is invoked before sending the PDU to network.

Metrics

To monitor SmppClient or SmppServerClient performance you can use metrics for send and receive queues.

Queue propery of type QueueState provides the following parameters:

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

Common Tools: Wireshark

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

SMPP related Wiki article is here.

FAQ

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 SubmitSm class: UnicodeFlashSMS, DefaultFlashSMS

How can I set sequence number before sending PDU

SMS Builder has Create method that returns SubmitSm list with sequence numbers set to 0.

You can assign the next number from the SequenceGenerator and pass this PDU list to Submit(SubmitSm[]) method.

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

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

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

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 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 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

Receipt format

Often you want to get delivery status for an SMS message. SMPP protocol provides the ability to request a delivery receipt in PDU submitted. 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")
);
```

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

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

This text format is represented in the library as Receipt class.

It has the following properties:

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

Submitted - Number of short messages originally submitted. This is only relevant when the original message was submitted to a distribution list within 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:

STATE	DESCRIPTION
Delivered	Message is delivered to the destination
Expired	Message validity period has expired
Deleted	Message has been deleted

STATE	DESCRIPTION
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 DeliverSmResp with status ESME_RX_T_APPN to SMPP server when 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 SubmitSm class or SubmitMulti class.

Before sending SubmitSm or SubmitMulti PDU you need to assign 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 Sequence in the database. For one *message.Id* you need to store several Sequence.

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

When you receive a delivery receipt in the event evDeliverSm, the server sends same 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 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

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 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);
```

EnquireLinkInterval specifies the time to wait after the last PDU exchange before sending the command. EnquireLink request won't be sent when client and server are sending PDUs.

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

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

From string variable

Open your license file with any text editor and copy and paste the content into the string variable in your code. Set license before using Inetlab.SMPP classes in your code:

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

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

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 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.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 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 SmppClient instance, you can define which Encoding will be used for specified DataCodings.

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

By default SmppClient has the following 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 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 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 MessageComposer class.

MessageComposer supports all types of PDUs: SubmitSm, SubmitMulti, DeliverSm.

You should invoke AddMessage<TSmppMessage>(TSmppMessage) method in each event handler for PDU received. MessageComposer saves PDU in memory and waits for the last segment of the message text and raises evFullMessageReceived event.

When PDU has no concatenation parameters this event will be raised right after calling AddMessage<TSmppMessage>(TSmppMessage) method.

When MessageComposer didn't receive last segment for a long time it raises 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(string.Format("Incomplete message received from {0}", pdu.SourceAddress));
}

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

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 (with sample app)

Production

The speed ultimately is determined by:

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

With well tuned system you can reach approximately 500 messages per second.

Tuning

You can try to play with following optimization parameters:

Change number of threads that process received messages (default is 3):

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

Change receive or send buffer size for the TCP socket:

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

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

Local Test

Inetlab.SMPP performance check on the local machine with logging disabled shows the following result:

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

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

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

            StartApp().ConfigureAwait(false);

            Console.ReadLine();
        }
```

```
        public static async Task StartApp()
        {

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

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

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

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

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

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

            Stopwatch watch = Stopwatch.StartNew();

            for (int i = 0; i < messagesNumber; i++)
            {
                tasks.Add(client.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);

        }
    }
}
```

SMPP Server FAQ (with sample app)

How to send message to the connected client

In following code target client is selected and 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 SubmitSm or SubmitMulti PDU, it generates a corresponding response and sets MessageId.

You can change the MessageId property in evClientSubmitSm and evClientSubmitMulti event handlers.

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

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 SmppServer class you can set server certificate and supported SSL/TLS protocols.

For 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.EnabledSslProtocols = SslProtocols.Tls12;
    server.ServerCertificate = new X509Certificate2("server_certificate.p12", "cert_password");

    server.Start();

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

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

        if (await client.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 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 To

```
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 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 SubmitMultiResp response received, it means SMPP server stored message for further delivery to recipients.

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

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. Missing `BatchMonitor` class. Use instead `client.Submit(IEnumerable<SubmitSm> batch)`. It waits when all responses will be received for a batch.

2. Events `evBindComplete`, `evSubmitComplete`, `evQueryComplete` were depricated. It is possible to use async await pattern or ContinueWith for corresponding `Bind`, `Submit`, `Query` methods.

3. `_client.AddressRange`, `_client.AddrNpi` and `_client.AddrTon` must be specified as `_client.EsmeAddress = new SmeAddress(AddressRange, AddrTon, AddrNpi);`

4. Sequence number and Command Status are moved to Header property of PDU.

- `data.Status` replaced with `data.Header.Status`,
- `data.Sequence` replaced with `data.Header.Sequence`

1. `client.GetMessageText` is moved to `client.EncodingMapper.GetMessageText`

2. PDU Properties

- `SourceAddrTon`, `SourceAddrNpi`, `SourceAddr` replaced with `SourceAddress` of type `SmeAddress`
- `DestAddrTon`, `DestAddrNpi`, `DestAddr` replaced with `DestinationAddress` of type `SmeAddress`
- `UserDataPdu` replaced with `UserData`
- `Optional` replaced with `Parameters`

1. Property `MessageText` in `SubmitSm`, `SubmitMulti`, `DeliverSm`, `DataSm`, `ReplaceSm` classes is deprecated. Use the method `pdu.GetMessageText(client.EncodingMapper)`.

2. Method `SmppClientBase.MapEncoding` moved to `SmppClientBase.EncodingMapper.MapEncoding`

SmppServer

1. Namespace for `SmppServerClient` class changed to `Inetlab.SMPP`.

2. IPEndPoint of the server must be specified in `SmppServer` constructor, instead of `Start` method of this class.

Serialization

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

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

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

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

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

Report a Bug

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

Changelog

[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.

SOFTWARE LICENSE

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

1. LICENSE TO USE SOFTWARE.

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

2. DISTRIBUTION / REDISTRIBUTABLE CODE

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

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

- Inetlab.*.dll

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

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

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

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

1. ADDITIONAL RIGHTS AND LIMITATIONS

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

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

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

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

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

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

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

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

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

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

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

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