



mBlox MSIP Interface Technical Manual V2.2

Provision of mBlox Services is dependent upon compliance with the specifications set forth herein. The information in this document is subject to change without notice. Although mBlox has taken reasonable steps to ensure the accuracy and completeness of this document, it shall not be liable for any losses whatsoever, whether direct or indirect, including without limitation any loss of profit, loss of use, or loss of data, as a result of any errors or omissions contained herein. The information or statements in this document concerning the specification or performance of mBlox software or hardware systems shall not constitute any binding promise or warranty.

Table of Contents:

Preface 5

 Organisation of this document 5

 Where to get help 5

Section 1: Change History..... 7

 1.1 Current document 7

 1.2 Version History: 7

Section 2: Introduction 8

Section 3: Getting Started 9

 3.1 Creating an instance..... 9

 3.2 Connecting to the mBlox Network 9

 3.3 Maintaining the connection 10

 3.4 Closing the connection 11

 3.5 Configuring the API 11

Section 4: Sending MT traffic 12

 4.1 Methods for sending 12

 4.1.1 Short..... 12

 4.1.2 Detailed..... 13

 4.2 Send return value 14

 4.3 Exceptions in send methods 14

 4.3.1 Profile Exception 14

 4.3.2 Encode Exception..... 14

 4.3.3 Block Exception 15

 4.4 Special parameters..... 16

 4.4.1 Sequence..... 16

 4.4.2 Body 16

 4.4.3 Originator and originatorType 18

 4.4.4 User Data Header (udh)..... 18

 4.4.5 Body encoding 19

 4.5 Block while sending..... 19

 4.5.1 Considerations and disadvantages when using "Blocking" 21

4.6 Binaries	21
4.7 Products and profile	21
4.8 Synchronization	22
4.9 Failover	22
4.10 Retry management	22
4.10.1 Rejects of type destinationBusy	22
4.10.2 Internal timeout and shutdown	23
4.11 Internet connectivity	24
4.12 Failing send calls	24
4.13 Throughput	25
Section 5: Inbound Messages (MO) and other Callbacks	26
5.1 SubmitConf callback	26
5.2 SubmitRej callback	27
5.3 DeliverReq callback (MO)	27
5.4 StatusInd callback	28
5.5 Log callback	29
5.6 System info callback	30
5.7 Synchronization	31
5.8 Inbound speed	31
Section 6: Miscellaneous methods	33
6.1 Activate Notify Up	33
6.2 Is Active	33
6.3 Is Alive	33
6.4 Is Up	33
6.5 Join All	34
6.6 Wait Until Connected	34
Section 7: Logging	35
Section 8: Configurations	36
8.1 Profile-groups	36
8.2 Configuring connect details	37
8.2.1 Connection details in conf.dat file	37
8.2.2 Example conf.dat files:	38

8.2.3 Connect details using set-method	39
Appendix.....	40
Configuration parameters	40
Parameter details	43
Reject codes.....	53
Example of conf.dat	55
Character map	56
Character map	57
Send examples.....	59
Text message with originator	59
Nokia ringtone	59
Notes	60

Preface

This document is designed to take the Client through the MSIP Interface in detail. If it does not answer the Clients question please contact mBlox customer support, (see **'Where to get help'** below).

Organisation of this document

Section 1	The change history of this document
Section 2	Introduction to the MSIP interface
Section 3	Getting started with the MSIP interface
Section 4	Sending MT traffic
Section 5	Inbound Messages (MO) and other Callbacks
Section 6	Miscellaneous Methods
Section 7	Logging
Section 8	Configurations
Appendix	

Where to get help

For any further information or support please visit our website at **www.mblox.com** or contact one of our offices:

United Kingdom: **Address:** 1 Oliver's Yard, 55-71 City Road, London, EC1Y 1HQ
 Tel: +44(0)20 8432 1260
 Fax: +44(0)20 8432 1290

USA: **Address:** 485 East Evelyn Avenue, Sunnyvale, California 94086
 Tel: +1-408-617-3700
 Fax: +1-408-617-3799

Sweden: **Address:** Tegnergaten 3, 1tr, 111 40 Stockholm
 Tel: +46(0)853 480780
 Fax: +46(0)853 480789

France: **Address:** 34 Blvd Haussman, 75009 Paris
 Tel: +33 1 72 71 25 55
 Fax: +33 1 72 71 25 99

Spain: **Address:** Zurbano, 41, 1º planta, 28010 Madrid
 Tel: +34 91 310 98 28
 Fax: +34 91 591 43 51

E-mail: **support@mblox.com**

Section 1: Change History

1.1 Current document

Document Title: mBlox MSIP Interface Technical Manual
Author: Steve Wood / Oskar Stal
Classification: Confidential
Version: 2.0
Date of Issue: 12th November 2006

1.2 Version History:

VERSION No.	DATE	REASON FOR ISSUE
1.2	14 th January 2004	Release Version
1.3	9 th February 2004	Minor changes
1.4	29 th July 2004	Package change and two new error codes
1.5	23 rd November 2004	Add sessionid and PSM\$plex codes
1.6	11 th May 2005	Modified definitions of at, operator and statusReport parameters. Corrected default value of keepAliveInterval. Corrected examples in appendix .
1.7	18 th August 2005	Addition of Tag/value parameter
1.8	14 th October 2005	Addition of contentType, serviceDesc and serviceId parameters.
2.0	12 th November 2006	Update to document presentation and format
2.1	12 th June 2007	Remove all details relating UTF8
2.2	15 th August 2007	Added properties tags (p.28)

Section 2: Introduction

The mBlox MSIP (Java Client API) is used to connect to the mBlox network for all services including Mobile Originated, Mobile Terminated and Premium Rated messaging. It is a Java based API developed by mBlox that is used and executed at the client's site. The API has been designed with ease of use in mind but the implementation requires basic Java skills and some previous experience in working with Java code. This document should be used in conjunction with product description documents.

Briefly the implementation process involves:

- Instantiating classes
- Connecting to the mBlox Network
- Sending and receiving messages

Section 3: Getting Started

Starting the API is a two-stage process

- Create an instance to use
- Opening the connection to the mBlox network

Configuration details are contained within the **conf.dat** file, supplied with the API. Minor changes and additions will be needed to the **conf.dat**, such as adding the correct server names, profiles and password. This information is supplied in the "Welcome to mBlox" e-mail. Other configuration parameters may be changed to suit specific application needs. The **conf.dat** file is described in detail in **section 8**.

3.1 Creating an instance

To start using the API, the application needs to create an instance of the following class.

```
com.mblox.gateway.clientapi.MbloxClient
```

This instance is then used for sending messages and querying the status of the connection throughout the client application's lifetime.

An instance is created using the constructor:

```
public MbloxClient (String confPath, String logPath, boolean logTrace, Listener listener)
```

For full details of the parameters, see the **appendix**. When creating the instance the location of the **conf.dat** file is specified in the **confPath** string. For details about the **logPath** and **logTrace** and the **listener** see the **appendix**.

3.2 Connecting to the mBlox Network

When the instance has been created it is still not "active". The API is started by calling the **connect()** method in order to connect to the mBlox servers.

```
public boolean connect (int millis) throws ConnectException
```

The parameter specifies how long the method will wait before time out if the connection could not be established. If zero is used the method does not time out.

The method returns true if the connection was established and false if not. If false is returned and a finite time out was declared the connection process has stopped. If this happens **connect()** must be called again to obtain a connection.

If '-1' is supplied the connection process starts and the method returns false immediately (this can be discarded) but continues to attempt to connect until **close()** is called. The connection status can then be established using methods described in **sections 5.4** and **5.6**.

This is part of the core functionality of the API and mBlox recommend that maintaining the connection be performed by the API and not by the client's application.

We do not recommend using a connection strategy that continuously re-connects, as there is a constraint on how many connections can be made (the client does not need to connect to more than 1 or 2 sockets) and such an approach may give rise to unnecessary errors.

The **ConnectException** is thrown if the Client API could not start the connection process. This could be caused by incorrect or missing parameters or it could be that the Client API is already connected. If this exception is thrown the connection process has not been started. The message within the exception contains details on why the Client API could not start the connection process.

Calling the **connect()** method launches a number of threads. These threads are all alive until the **close()** method is called. The **isAlive()** method can be used to identify live threads. (See **section 6.3**)

3.3 Maintaining the connection

The Client API maintains the connection to the mBlox servers automatically, when possible.

If the connection is found to be down at any time whilst the API is running, a reconnect of the Client API should NOT be triggered.

The API will automatically reconnect as soon as the cause of the lost connection is gone (i.e. network disturbances). The Client application may need to change its behaviour if the connection is lost.

3.4 Closing the connection

The `close()` method should only be called before shutting down the entire application. When this is done the Client API closes all connections and then stops all the threads that were started in the `connect()` call.

This method call returns before the connection is really closed. The call just starts the process. Thus it also returns before all the threads have stopped. The `joinAll()` method can be used to check that all threads have stopped. (See [section 6.5](#))

It is important to keep the Client API connection open for as long as the application is alive. A strategy that connects and closes the Client API for each sent message is not recommended and will provide a poor performance and unnecessary data traffic on the connection.

3.5 Configuring the API

There are many parameters that can be set before starting the Client API. However, in most cases the standard values provided in the `conf.dat` file are sufficient and only the password and server lines require modification. Clients with very specific requirements might be interested in changing other values to optimize the Client API for specific application needs.



It is strongly recommended that clients discuss any changes with the mBlox support team before changing configurations that they may not fully understand.

All configuration values can be changed or inspected using `get / set` methods as well as via the `conf.dat`, but as most of the configurations can be set using the `conf.dat` file it is strongly recommended not to use any `set` methods and to use the `conf.dat` file instead.

The `set` methods can only be used before `connect()` has been called. If configuration changes are required after the application has started, `close()` should be used to disconnect the Client API, the settings then changed and `connect()` method used to restart.

Section 4: Sending MT traffic

This section describes in detail how to send outbound (Mobile Terminated) messages.

4.1 Methods for sending

There are two different methods that can be used to send an MT message:

- Short
- Detailed

4.1.1 Short

This is the simplest way to send messages and it is the recommended method if it meets the client's requirements. This method allows clients to send plain text messages with dynamic originators (alpha numeric, network short codes, etc.), but it does not support sending more advanced types of messaging such as binary data and does not support delivery receipts (via **StatusInds**). If binary messaging is required (or if there is other functionality needed that is not provided in the short method), then the **sendSMSDetailed()** method must be used.

Method Header: Short

```
public int sendSMSShort (    String destination,  
                           String body,  
                           int profile,  
                           String originator,  
                           int originatorType,  
                           int blocking,  
                           String billingRef  
                           throws EncodeException,  
                           ProfileException, BlockException {
```

For details about each parameters meaning, see the **appendix**. For details about the exceptions, see **section 4.3**. For details about the return value see **section 4.2**. All parameters can be set to **null** to be omitted or '-1' for default behaviour.

4.1.2 Detailed

This method supports all SMS message types supported by the mBlox network and product set.

Method Header: Detailed

```
public int sendSMSDetailed (    String destination,  
                             String body,  
                             int profile,  
                             String originator,  
                             int originatorType,  
                             int sequence,  
                             int smsClass,  
                             boolean statusReport,  
                             int pid,  
                             int expirationTime,  
                             int reply,  
                             String bodyEncoding,  
                             String UDH,  
                             int blocking,  
                             String billingRef,  
                             int multipart,  
                             int operator,  
                             String tariff,  
                             String sessionId,  
                             Properties tags,  
                             String serviceDesc,  
                             int contentType,  
                             int serviceId  
    ) throws EncodeException, ProfileException, BlockException {
```

For details about the parameters and their meanings see the **appendix**, for details about the exceptions see **section 4.3** and for details about the return value see **section 4.2**.

All parameters can be left out using **null**, **false** or **'-1'** as it sets the parameter to its default value. The acceptable values for the various parameters will depend on the product being used.

4.2 Send return value

The return value of all **sendSMS()** methods is an integer. If the value is '-1' it means that the Client API could not put the message into the mBlox network and that the message will never reach the mBlox servers, unless the client retries sending the message. Clients are advised to have some kind of retry strategy for when the method calls return '-1', because it might happen from time to time when there are network problems. This is discussed further in **sections 4.11** and **4.12**.

If not '-1', the **sequence** value for that message is returned. The **sequence** is the specific identifier used for this particular message up until the mBlox servers have accepted the message. The identifier is then replaced with a specific mBlox unique identifier (the **msgReference**). For more details about the **sequence** see **section 4.4.1**. This replacement is notified to the client in the **SubmitConf()** callback, see **section 5.1**.

4.3 Exceptions in send methods

The **sendSMS()** methods can throw three different exceptions (**profile**, **encoding** and **blocking exceptions**). Note that for some exceptions the message will be sent anyway and for some not. Further details about the exception can be obtained by using:

```
theThrownException.getMessage()
```

4.3.1 Profile Exception

The configuration does not support the value supplied in the parameter **profile**. The profile being used must be declared in the **conf.dat** and configured in the mBlox network. The message has not been sent when this exception is thrown.

4.3.2 Encode Exception

The encoding of the **body** or **originator** field is not acceptable. The message of the exception should be checked for further details. The message has not been sent when this exception is thrown.

4.3.3 Block Exception

The block exception is only thrown when the blocking functionality is used. The exception contains a reason-code that should be examined. Following is a description of the codes. The possible reason codes are found as public-static-final parameters within the exception, for example:

```
com.mblox.gateway.clientapi.BlockException.ALREADY_BLOCKING
```

Reason code	Message sent?	Description
TIME_OUT	Yes	The Client API has waited the time supplied by the user and the message has still not been confirmed or rejected by the mBlox servers.
TOO_MANY_BLOCKING	No	There are too many threads blocking at the same time. The configuration parameter maxBlockingSequences must be changed to match the possible amount of threads that might block at the same time.
ALREADY_BLOCKING	No	The client is already blocking on this sequence with another thread. The use of sequences must be unique.
CLOSED	No	The Client API is closing down or already closed.
UNKNOWN	Yes	An unknown error occurred while waiting for the confirmation or rejection.
RESULT_NOT_AVAILABLE	No	Only thrown by the method <code>getBlockSendResult()</code> when used with a sequence for which there is no result. Either <code>getBlockSendResult()</code> was called too early and it is still blocking or timed out or a sequence is being used that was never sent in the first place.

4.4 Special parameters

Some of the parameters used when sending messages demand further detailed descriptions. A complete list of parameters can be found in the **appendix**.

4.4.1 Sequence

The **sequence** is a unique identifier for a message. This value identifies the message until the mBlox servers accepts the message. At that point the identifier is switched for an mBlox unique identifier generated by the mBlox servers. This new identifier is supplied in the **SubmitConf()** callback (see **section 5.1**).



It is strongly recommended to use the value '-1' as the sequence parameter in the sendSMS() methods. The Client API then generates a sequence that is returned from the sendSMS() call if the message was successfully sent.

If the clients supply their own **sequence**, it is important that it is unique. One suggestion is to have an increased counter and then append the four least significant digits from

```
System.currentTimeMillis()
```

The sequence must be a positive number below the maximum value of an integer.

4.4.2 Body

Some characters in the body must be specially encoded. The Client API, depending on the settings in the **conf.dat** file, can do this automatically.



If the configuration parameter encodeTextBody is declared as false, no characters should be encoded. For example the message body could be: "This is an SMS with a smiley :-)"

If **encodeTextBody** is set to **true** the following characters **must** be encoded: comma, full-stop, colon, semi-colon, line-feed, carriage-return and equals. This is done by writing the hexadecimal ASCII character, with a colon to identify it as an escape encoded character, for example, ":3A" where 3A is the hexadecimal ASCII value for a colon. The example above would become "This is an SMS with a smiley :3A-)"

If **encodeTextBody** is **true** any character may be encoded as “:XX” where XX is the hexadecimal ASCII value for that character.



It is recommended to have encodeTextBody set to false.

For binary messages the body must always be supplied in the same format regardless of the configuration. It should be supplied as “:A1:A2:A3” etc, where A1,A2,A3 represents hexadecimal values for the bytes sent in the message.

If configuration parameter **encodeUnicodeBody** is **false**, no characters should be Unicode encoded. For example: “This is an SMS with a smiley :-).” If **encodeUnicodeBody** is **true** all characters **must** be Unicode encoded. A character is encoded as “:XXXX” where XXXX represent the hexadecimal Unicode value for that character.

In a Java string all characters are represented by Unicode anyway and presenting the string to the method with **encodeUnicodeBody** set to true will correctly send Unicode. The example above would be:

```
“:0054:0068:0069:0073:0020:0069:0073:0020:0061:006e:0020:0053:004d:0053:0020:0077:0069:0074:0068:0020:0061:0020:0073:006d:0069:006c:0065:0079:0020:003a:002d:0029”
```

It is recommended to have **encodeUnicodeBody** set to **false**.

4.4.3 Originator and originatorType

This parameter defines what originator the SMS will have upon terminated on the handset. The format varies depending on the **originatorType**. For further instructions regarding how to use the **originator** parameter please see the **appendix**.

originatorType	Type	Description
0	Numeric	Only numeric digits are allowed in the originator parameter. The terminal will add a plus sign to the originator and the terminal can reply to this msisdn. For example '46709001122'. Maximum length is 15 digits.
3	Shortcode	Only numeric digits are allowed in the originator parameter. The terminal will not add anything. The terminal can reply to this number, as long as it is a valid shortcode.
5	Alphanumeric	Letters and numeric digits are allowed (see appendix for details). The terminal cannot usually reply to this originator . (Some handsets may allow replies if the originator consists only of numeric digits). Maximum length is 11 characters. If configuration parameter encodeTextBody is false , all characters are supplied without any encoding. If it is true all illegal characters must be encoded as described for the body parameter in section 4.4.2 .

OriginatorType default is 0.

4.4.4 User Data Header (udh)

The **udh** parameter is often used when sending binary messages (**bodyEncoding** set to "Data"). It always consist of 8-bit bytes and is supplied as ":A1:A2:A2" where A1,A2,A3 are the hexadecimal value of those bytes.



For further details on how to format UDH's refer to the vendor specific documentation for the type of binary content being sent.

4.4.5 Body encoding

This parameter is used to specify which type of content is being sent in the message. Below are the possible values:

Value	Explanation
"Data"	Used when sending binary messages
"Unicode"	Used when sending Unicode messages
"Text"	Used when sending ordinary text messages
Null	Default is "Text", so this will result in a text message

4.5 Block while sending

This is special functionality that can be used to simplify the development process. It does however have drawbacks in the final solution so this method is not recommended for competent Java programmers.

The `sendSMS()` method will block until the message has been confirmed or rejected by the mBlox servers. After this block is released the result of the sending is available by doing a method call to the **Client API**. This can be very convenient if a web application or a test application is being developed.

The parameter **blocking** is normally supplied as '-1'. If a different value is supplied the `sendSMS()` methods will block for that amount of milliseconds waiting for a confirmation or rejection. Use of zero will make the block time infinite.



Note that the method will not block for exactly that amount of time. It can in some situations block for 20-40 seconds even though the supplied time was much shorter.

When the method does not return '-1' and does not throw any exceptions the result is available. It is obtained by calling the `getBlockSendResult()` method.

For example:

```

int sequence = myMbloxClient.sendSMSShort('0046709001122',
                                         "Hello",
                                         -1,
                                         null,
                                         -1,
                                         60000, //blocking
                                         null);

if (sequence != -1) {
    BlockSendResult myResult;
    myResult = myMbloxClient.getBlockSendResult(sequence);
}

```

In this example the `sendSMSShort()` method will wait 60 seconds for a confirmation or rejection. Synchronization is not a problem as many threads can do the operations in the example above at the same time. The result instance obtained by the `getBlockSendResult()` call includes the following parameters.

Parameter	Description
Accepted	Boolean value. It is true if the mBlox servers confirmed the message and false if the message was rejected. A rejected message will never reach the handset.
rejectCode	Only supplied if accepted is false . Identifies the reason for rejection, see appendix .
rejectMessage	Only supplied if accepted is false . Lexical description of why the message was rejected.
msgReference	Only supplied if accepted is true . This is the new mBlox unique identifier for this message. If StatusInds arrive later, they will reference the original message using this parameter.
Retry	The possible values are defined as public-static-final parameters in the BlockSendResult class named RETRY_YES and RETRY_NO . It is only supplied if accepted is false and indicates whether this rejected message should be retried or not. See section 5.7 for further details.

4.5.1 Considerations and disadvantages when using “Blocking”

To obtain a high throughput several threads are needed to simultaneously call the `sendSMS()` method. Without the use of blocking one thread is more than enough to obtain maximum throughput.

Each thread that does the `sendSMS()` calls can be blocked for quite some time. If sending without blocking the `sendSMS()` will never block for long, thus leaving the thread to do other operations while waiting for a lingering confirmation / rejection response.

When blocked sending is used together with MO or StatusInds the application has to be multi threaded. In other words the application must be able to handle incoming MO and StatusInds at the same time as being blocked in the `sendSMS()` method. If not deadlocks occur, poor performance will result and so the use of blocking in this way is not recommended.

4.6 Binaries

mBlox provides clients access to a range of **transparent** MT messaging products . For detailed information on how to send binary messages please refer to the vendor specific instructions that can be obtained by the relevant handset manufacturers for the type of content to be sent.

Binary messages can only be sent using the `sendSMSDetailed()` method. It is done by supplying parameter **bodyEncoding** as “Data” and the **body** and **udh** parameter as “:A1:A2” etc, where A1,A2 represents the hexadecimal values for the bytes. For further details see **section 4.4.2**.

4.7 Products and profile

The parameter **profile** is used to determine the mBlox MT Product being used, so each Product has a corresponding Product/Profile mapping.

ProfileIDs are unique for every client and are provided along with connection details. Use of **profileID** allows several different products to exist on one account with one Client API setup.

4.8 Synchronization

Several threads can send at the same time, both with and without blocking, so the application does not have to concern itself with synchronization when sending traffic.

4.9 Failover

The Client API contains several failover strategies. There is a configuration parameter named **loadShare** and if it is **true** the traffic is alternated between the primary and secondary server. If it is **false** the primary server is used as much as possible. **True** is recommended.

The Client API also measures the number of **destinationBusy** rejects. These rejects are sent when the destination for the messages is unavailable (typically this only happens on low quality products). If one server sends a high percentage of these rejects, the Client API will try to avoid that server for a while. This is configured in the parameters **destBusyBadQuota** and **destBusyTestSet**. The first is a number from 0-100 which defines what percent of all incoming confirms and rejects should be **destinationBusy** rejects.

The second parameter defines what size set of incoming confirms and rejects should this be measured on. In other words the percentage is measured on the last **destBusyTestSet** amount of confirms and rejects.

The configuration parameter **failoverOnRejects** defines a list of **rejectCodes** (see the **appendix**). If any of these reject codes arrive the Client API will temporarily route less traffic to that server.

4.10 Retry management

The Client API handles retry management. Messages that have not been confirmed / rejected within a certain amount of seconds (configured in parameter **windowRetryInterval**) will automatically be retried.

4.10.1 Rejects of type **destinationBusy**

Using a low end product such as mBlox Economy the destination for the messages might become temporarily unavailable. In these situations the mBlox server returns a reject of type **destinationBusy**. This means that the message could not be accepted and it is recommended to try again later. These rejects are identified by the **retry** value that is set to 1 for **destinationBusy** rejects.

Messages that receive this type of reject will be retried automatically by the Client API, so these rejects will never reach the application through the `SubmitRej()` callback (see **section 5.2**). The messages will be retried rapidly, thus the configuration parameter `windowRetryInterval` is not used here. The automatic retry of `destinationBusy` rejects can be turned off by the call `setResendOnBusy(false)`. If this is done the rejects will instead hit the `SubmitRej()` callback with the `retry` parameter set to one (see **section 5.2**).

4.10.2 Internal timeout and shutdown

Some rejects are generated within the Client API. They all have a `rejectCode` between 100 and 200. There are three different reasons why the Client API would generate a rejection:

RejectCode	Description
100	The message timed out within the Client API. The Client API has probably tried to resend the message many times without any success. The configuration parameter <code>windowMaxReservationTime</code> defines when the message times out.
101	These rejects are created when the Client API is shut down <code>close()</code> is called. If there are messages that the Client API is trying to resend without any success, they are rejected like this when the <code>close()</code> method is called.
102	There is a repository within the Client API used for messages that have not successfully been delivered yet. If this repository grows full messages will be rejected with this <code>rejectCode</code> . If this happens often, the configuration parameter <code>repositorySize</code> should be changed

4.11 Internet connectivity

It is very important to have a stable connection to the mBlox servers; a reliable internet connection is required to achieve this. With poor internet connectivity sending will be slow and the `sendSMS()` methods will often return '-1'.

Ping-times from Client API to the mBlox server should be around 50 milliseconds maximum and the server where the Client API is running must have a dedicated IP address.

The Client API contains functionality to maintain stable traffic flow even during poor network conditions, but if network connectivity is impaired, this will probably still result in a lower throughput.

If network connectivity problems are experienced a somewhat more sophisticated sending strategy is recommended. For example:

```
for (inti=0;j<10;j++){
    if (sendSMS(..)!=1){
        break;
    }
    Thread.sleep(10000);
}
```

A similar effect is obtained by raising the value of the `windowBlockFor` parameter; if set to 120000 a similar effect is obtained. The `sendSMS()` method will then block for a longer time waiting for a chance to get the message on the network.

4.12 Failing send calls

The send call fails when '-1' is returned from the `sendSMS()` method call. This happens when the Client API could not put the message on the network for one reason or another. The three most common reasons for '-1' are:

- The Client API is currently not connected the way needed to deliver this message
- Traffic is moving slowly across the network. The Client API always blocks while waiting for an available slot, but after some time it will stop waiting and return '-1'

- The mBlox servers have returned large numbers of destinationBusy rejects. Those rejects exhaust the Client API's sending capabilities, since the Client API continuously retries them. This happens when the destination for the message gets unavailable for longer periods

In this context it is interesting to look at the **windowBlockFor** configuration parameter. It defines how long the **sendSMS()** call waits for an available send slot. High value means that '-1' will be returned less frequently, but the **sendSMS()** method will block for longer.

4.13 Throughput

There are four main factors limiting the available throughput:

- The speed of the application
- The quality of the Internet connection between mBlox and the servers hosting the Client API
- Maximum allowed throughput on the account (the rate at which mBlox accept messages). This is configurable by mBlox by product
- Maximum available throughput to the destination, (the rate at which mBlox are sending messages out to the GSM network). This might vary with product and destination

The Client API has been thoroughly tested and can submit traffic fast enough to not be a limiting factor in throughput. Throughput problems are likely to be caused by one of the above four factors, not the API itself.

If blocking is being used while sending messages several threads will be needed to obtain maximum throughput (see **section 4.5**).

If throughput problems are experienced the most likely causes are.

- Internet connectivity between the application server and the mBlox servers is poor
- Callback methods are not being processed fast enough (see **section 5.8**). All communication is connected in some respect, so if a **StatusInd()** callback is not processed fast enough this may also slow down the SMS traffic to the mBlox servers

Section 5: Inbound Messages (MO) and other Callbacks

The Client API will send various types of information back to the application triggered by different events (such as receiving an MO). This is done through the use of the **listener**. A class should be created that implements the interface **com.mblox.gateway.clientapi.Listener**. The class will have to include several predefined methods that will be called by the Client API upon certain events.

One instance of the class is supplied to the Client API when creating the **MbloxClient** instance. Listener is not essential for operation of the Client API. **Null** can be supplied instead of a listener in the **MbloxClient** constructor; however no callbacks will be received in this case (i.e. no MOs or Delivery Notifications).

If blocking send is used and the send result retrieved with the **getBlockSendResult()**, this could however be a satisfactory approach. The listener will be essential for implementation of certain products; those which feature MO or Delivery Notifications. To receive MOs or StatusInds (Delivery Notifications) a listener instance must be created and supplied in the **MbloxClient** constructor.

Details about all parameters are found in the **appendix**.

5.1 SubmitConf callback

The method is called each time a message is confirmed by the mBlox servers.

```
public void submitConf(      int sequence,  
                          String msgReference,  
                          long at,  
                          int operator)
```

This means that the mBlox servers have accepted the message and will deliver it into the network. At this time the unique identifier (**sequence** parameter) for this message is switched for a new mBlox unique identifier (**msgReference** parameter). From now on this message will only be referenced by the **msgReference** value. The **operator** parameter contains the identifier for the operator to which the message was submitted. If no identifier is provided, this contains the value '-1'.

5.2 SubmitRej callback

The method is called each time a message is rejected by the mBlox servers.

```
public void submitRej(    int sequence,  
                        String reason,  
                        int rejectCode,  
                        long at,  
                        String msgFrom,  
                        String destination,  
                        int retry)
```

This means that the mBlox servers have rejected the message and will not try to deliver it into the network. The reason for rejection is supplied as parameters **reason** and **rejectCode**. The different values of the **rejectCode** parameter are listed in **appendix**. If **retry** is 1 this is a **destinationBusy** reject and is treated appropriately.

5.3 DeliverReq callback (MO)

This method is called each time a mobile originated (Inbound) message arrives.

```
public boolean deliverReq (    String destination,  
                            String originator,  
                            String body,  
                            long at,  
                            long sequence,  
                            String msgReference,  
                            int deliverer,  
                            int operator,  
                            String tariff,  
                            String bodyEncoding,  
                            String UDH,  
                            String sessionId,  
                            Properties tags)
```

The **sequence** is the mBlox unique reference for this message. If messages are sent more than once because of network problems, the Client API will filter them out (using **sequence**), thus duplicates will never generate this callback. In other words the client will never retrieve two calls to this method with the same **sequence** value (unless the **DeliverReq()** callback returns **false** or generates an exception, see below). This also means that if two messages arrive that seem to be the same, but have different **sequence** values, they are considered to be different messages by the mBlox servers and should be treated as such.

This method returns a boolean value. This value should be **true** if delivery was successful. If the return value is **false**, this message will not be confirmed by the Client API and will be resent within one minute by the mBlox servers. The same will happen if the **DeliverReq()** methods throw an exception back to the Client API. When this happens the same message with the same unique **sequence** will appear as a **DeliverReq()** callback more than once. It will also soon stop all incoming traffic (StatusInd and MO) until it can be successfully confirmed.

The **msgReference** is a special reference for this message generated by the originating SMSC. This reference is not always supplied.

5.4 StatusInd callback

This method is called each time a status indication arrives. A status indication is a message containing the current status for an SMS, for example "delivered", which means that the SMS has reached the handset. For a complete list of available StatusInds please see the **appendix** and parameter **status**.

```
public boolean statusInd(
    String msgReference,
    long at,
    String originator,
    String status,
    int reason,
    int occurrenceTime,
    String clientRef,
    long sequence,
    Properties tags)
```

The **msgReference** parameter tells the client which SMS this StatusInd relates to. It is the same unique identifier that was supplied as **msgReference** in the **SubmitConf()** callback.

The **sequence** parameter is the unique identifier for the specific status indication message, thus has nothing to do with the **sequence** of the originating SMS. If messages are sent more than once because of network problems, the Client API will filter them out, thus they will never generate this callback. In other words the client will never retrieve two calls to this method with the same **sequence** value (unless the **StatusInd()** callback returns **false** or generates an exception, see below). This also means that if two messages arrive that seem to be the same, but have different **sequence** values, they are considered to be different messages by the mBlox servers and should be treated as such.

This method returns a Boolean value. This value should be true if delivery was successful.

If the return value is **false**, this message will not be confirmed by the Client API to the mBlox server, so the mBlox server will re-send the message within one minute, and continue to do so until it receives the relevant confirmation. The same will happen if the **statusInd()** methods throw an exception back to the Client API. When this happens the same message with the same unique **sequence** will appear as a **StatusInd()** callback more than once. Note that it will also quite soon stop all incoming traffic (StatusInd and MO) until it can be successfully confirmed.

5.5 Log callback

This method is called each time a row is logged to any of the log-files. The method is called regardless of whether the log-files are turned on or off.

```
public void log (          int log,
                    String message)
```

The **log** parameter identifies the originating log-file. The values for the log parameter are found as public-static-final values in the **MbloxCient** class. The second parameter declares the actual information to log.

Value	Description
LOG_TRACE	Logs detailed information about what is going on within the Client API.
LOG_FULLIO	All traffic that is sent back and forth between the Client API and the mBlox servers.
LOG_ERROR	Contains error messages.

Using this callback the client can create specialised logging strategies. However, this method will be called very often and there may be ~30 calls per SMS so it is not advisable to save a row in a database for each call, or perform other time consuming activities with the results. This of course depends on what kind of throughput is needed and how fast the application is.

5.6 System info callback

This method is called when the Client API has general information about its own state.

```
public void systemStatusInfo (      String message,
                                int  messageCode,
                                int  value,
                                long at)
```

The **messageCode** states what kind of system information this is. The possible values are found as public-static-final parameters in **MbloxClient** with the names as stated in the table below. The **message** parameter contains a description of what happened.

MessageCode	Description
SYSTEMINFO_MEMORY	The Client API has encountered a memory problem. Restart the application and check it for memory leakages.
SYSTEMINFO_UP	The connection to the mBlox servers has gone up with a certain profile-group. The value contains the index of the profile-group whose connection has gone up. This callback will only be performed when the "notify-up" functionality is turned on.
SYSTEMINFO_THROTTLE	The mBlox servers generated a throttle-message. It is advisable that the traffic speed be lowered on this particular destination operator. The value is the sequence of the original SMS that generated this throttle-message.
SYSTEMINFO_THREAD_CREATED	A thread has been created and started within the Client API. The message parameter contains the "name" of the thread. This can be used for thread management within the application.
SYSTEMINFO_THREAD_DESTROYED	A thread has been stopped and destroyed within the Client API. The message parameter contains the "name" of the thread (matches the originating thread-created call). This can be used for thread management within the application.

5.7 Synchronization

It is very important to realize that all callback methods can be called at the same time by many different threads. The class that implements the **Listener** interface must act accordingly. If the client is unfamiliar with the area of synchronization it is advisable to add the word “synchronized” to all callback method declarations within the listener class.

For example:

```
public synchronized void submitRej(...) {  
    // client application code  
    ...  
}
```

5.8 Inbound speed

The speed at which callback methods are executed is crucial for the outbound throughput. This might seem odd, but since all traffic uses the same sockets to talk to the same servers, both the inbound and outbound speeds are closely related.

The single most common reason why outbound traffic is slow is that the listener callbacks are not executed quickly enough. For instance, if a database is queried for each incoming StatusInd, chances are quite high that the MT throughput will suffer after a period of intense sending.

If traffic is to be sent at high throughput for a long period (for example 5 messages per second for 10 minutes) it is advisable to save the StatusInds to a text file and then have a separate process that stores them in the database, or to make sure the database write is fast enough to handle this.

As an example, to send 10 messages per seconds and also receive StatusInds there is a need to process the callback methods within an average ~20 milliseconds (depending on round-trip times).

If traffic throughput suddenly drops, one strategy would be to delay processing the StatusInds, and then process all StatusInds when the burst of SMS traffic has finished.

Returning **false** immediately in the **StatusInd()** and **DeliverReq()** callbacks would do this while the send batch executes. Once the send is complete, stop returning **false** and start processing the StatusInds. This will not lose any information by since all messages and notifications are stored in the mBlox server queues until they are successfully confirmed (the callback method returned **true**). There are some things to consider before using this strategy:

- StatusInds only have a lifetime of four hours in the mBlox server queues, so the confirmation must be within this time
- This will also delay MOs mapped to the same username since they are placed in the same outbound queue in the mBlox servers
- The queues in the mBlox servers have a limited size so there is a risk queues may fill and messages be lost

Section 6: Miscellaneous methods

6.1 Activate Notify Up

If the connection is down for a certain profile-group (for details about profile-groups) the Client API can notify once that connection is back up again. Calling this method does this:

```
public boolean activateNotifyUp (int profileIdx)
```



Note that the notification-service has only been turned on if true was returned. The service cannot be turned on if the connection is up.

When the service has been turned on a call to the `systemStatusInfo()` method will be done once the supplied profile-group is back up again. Note that once that call has been made, the notification-service is turned off again.

6.2 Is Active

This method checks that the Client API is in a state where it can use its connectivity maintenance functionality correctly.

```
public boolean isActive ()
```

In reality **false** is only returned before `connect()` is called and after `close()` is called.

6.3 Is Alive

Returns **true** if the Client API has any threads running.

```
public boolean isActive ()
```

6.4 Is Up

Returns **true** if the Client API is correctly connected to the specified profile-group (for details about profile-groups). Use -1 to check if all configured profile-groups are up.

```
public boolean isUp (int profileIdx)
```

6.5 Join All

Blocks until all threads within the Client API have stopped. It does cause any thread to stop so this must be done by calling the **close()** method.

```
public void joinAll (long millis)
```

It will block for the amount of milliseconds supplied. Zero blocks forever. This method is often used in conjunction with **close()** and **isAlive()**, for example:

```
myMbloxCliient.close();  
myMbloxCliient.joinAll(120000);  
If (myMbloxCliient.isAlive()) {  
    System.out.println ("Warning the ClientAPI threads did not stop after 2  
    minutes");  
} else {  
    System.out.println ("ClientAPI successfully closed");  
}
```

6.6 Wait Until Connected

This method blocks until a certain profile-group is connected and ready for traffic.

```
public boolean waitUntilConnected (int profileIdx, int millis)
```

It blocks for the maximum amount of milliseconds supplied. This is a target value, so it may not be blocked for exactly the amount supplied, but close to it. Use zero to block forever. Supply **'-1'** as **profileIdx** to wait for all configured profile-groups to be connected. If it returns **true** the profile-group is connected (or profile-groups if **'-1'** was supplied). It returns **false** if the waiting timed out.

Section 7: Logging

There are three different log-files called the **trace-log**, **fullIO-log** and **error-log**. They can all be turned on and off using set methods. The methods are called **setErrorLogging()**, **setTraceLogging()**, and **setFullOLogging()**.

The **trace-log** can also be turned off in the **MbloxCient** constructor – the reason for this is that otherwise the Client API starts to write to the log-files before the client can turn it off.

It is strongly recommended to have all log-files turned on at all times since they are vital if problems do occur, and mBlox Support may require the log files to match against mBlox server logs for the session.

The files will be located in the folder specified in the **MbloxCient** constructor. The path to the log-files will be **logPath+logname** where **logPath** is defined in the constructor and the log-names are "**fullIO**", "**error**" and "**trace**". For example, if **logPath** is **"/usr/local/clientapi/"** the **fullIO-log** will be in folder **"/usr/local/clientapi/fullIO"**.

The logs are written in files named **"2003-03-20.txt"** (the current day's date), so that logs are rotated every day. These files can be large, so it is advisable to monitor the disc space where the Client API is running, and to archive older log files.

The three logs contain the following information:

- **fullIO-log** :Contains all traffic that is sent back and forth from the Client API to the mBlox server (the full communication)
- **trace-log** :Contains detailed information on activity within the Client API. It is a very useful source in the event of error or unexpected behaviour.
- **error-log** : When errors occur they are written in the **trace-log**. Together with that error there is an id-number that refer to a stack-trace. That stack-trace can then be found in the **error-log**. The **error-log** will only write stack-traces at a maximum rate of 1/s. If errors occur more often then that they will have the id-number written as '-1' within the **trace-log**. The **error-log** only contains those stack-traces.

Section 8: Configurations

This chapter describes in detail some of the available parameters in the configuration file. For full details about all parameters, see the **appendix**.

8.1 Profile-groups

The connection to the mBlox servers can be split into several profile-group connections. Each profile-group is handled separately with respect to functionality for maintaining the connection. Each of these profile-groups can connect to different mBlox accounts and different servers.

For each profile-group a list of profiles is specified that comprise the profile-group (see 0 for details about profiles). One profile may not be shared between different profile-groups. Adding the profile '-1' to a profile-group's list of profiles specifies it as the default profile-group.

When a message is sent through the Client API the **profile** parameter supplied in the call is used to route the message into the correct profile-group. If the supplied **profile** parameter is unknown, the profile-group with profile '-1' is used.

Each profile-group has an index value. This starts at zero for the first specified group and is numerically sequential for other groups. This index is used each time a specific profile-group is referenced within the Client API.

The profiles that comprise a given profile group are declared in an entry with the following format.

profile_X 1001,1003,1004

Where X is the index value of the profile group and the numbers are comma-delimited profileids.

8.2 Configuring connect details

IP-numbers are specified in a tree structure. The top nodes are the profile-groups. Each profile-group has one or more servers. Each server has one or more IP-numbers. For each server the client specifies a destination port.

There are two ways to specify the connect details, either by the `conf.dat` file, or by using the `selfProfileSocketGroupSettings()` method.

8.2.1 Connection details in `conf.dat` file

All IP-numbers are supplied as rows in the `conf.dat` file in the format

profile_X_ip_Y_server_Z server-address

Where X is the profile-group index and Y is the IP-number index for that server, and Z is the server index for that profile-group.

No indexes may be skipped; all sequential numeric values to the maximum must be used. For example: if there exist a server indexed 3 for a profile-group A, there must be servers indexed 0-2 for A also.

Each specified server must have a row that specifies the destination port for the connection in the format

port_profile_X_server_Z port

Where X is the profile-group index and Z is the server index for that profile-group.

Each specified profile-group has one or more profile associated with it (as described above). This information is used by the Client API to route the messages to a profile-group. All profiles used that are not explicitly defined in the `conf.dat` file will be routed to the profile-group with profile '-1' configured.

For each profile-group the values **clientName**, **applicationName** and **password** must be specified. This can be used to make different profile-groups connect to different mBlox accounts.

8.2.2 Example conf.dat files:

```
# profile-group index 0
profile_0 -1
clientName_0 loginName
applicationName_0 myApplication
password_0 verySecret
port_profile_0_server_0 1000
profile_0_ip_0_server_0 serverone.mblox.com
profile_0_ip_1_server_0 secondary-route-serverone.mblox.com
port_profile_0_server_1 1000
profile_0_ip_0_server_1 servertwo.mblox.com
profile_0_ip_1_server_1 secondary-route-servertwo.mblox.com
# profile-group index 1
clientName_1 loginName
applicationName_1 myApplication
password_1 verySecret
profile_1 1033,1034,1035
port_profile_1_server_0 1010
profile_1_ip_0_server_0 serverone.mblox.com
profile_1_ip_1_server_0 secondary-route-serverone.mblox.com
port_profile_1_server_1 1010
profile_1_ip_0_server_1 servertwo.mblox.com
profile_1_ip_1_server_1 secondary-route-servertwo.mblox.com
```

This is a simple configuration for two profile-groups that connect to two separate accounts (one on each port).

8.2.3 Connect details using set-method

The connect details can also be specified using a set method called

```
public void setProfileSocketGroupSettings (ProfileSocketGroupSetting[] settings)
```

The client supplies an array of **ProfileSocketGroupSetting** instances where each instance represents the connect details for one profile-group. For further details about the content of the **ProfileSocketGroupSetting** see the ProfileSocketGroupSetting.java file. The parameters within the class are:

Parameter	Description
profiles[]	Contains all profiles that belong to the profile-group. This information is used when the Client API routes the message to a profile-group. All profiles used that are not explicitly configured will be routed on the profile-group with profile -1 configured.
noConnections	Number of sockets to use connecting to each server.
ipNumbers	The IP-numbers to use when connecting this profile-group. This vector has one element for each server this group will use. Those elements are vectors that then contain one element for each IP-number. Those elements are strings.
ports[]	Array with the network port numbers to which the servers should connect. Indexed the in the same way as the servers in the ipNumbers vector above.
clientName	See the appendix, (Configuration parameters) .
applicationName	See the appendix, (Configuration parameters) .
password	See the appendix, (Configuration parameters) .

Appendix

Configuration parameters

This chapter describes the contents of the **conf.dat** file. The descriptions also apply to the set / get methods as they use identical parameters. **For all Boolean values, use “true” or “false” in lowercase.** All parameters that are not required can be left out; the default value will be used. The **conf.dat** file reader is case sensitive.

Parameter	Description
applicationName_X	Application name used when logging on to the mBlox servers. Used to distinguish different client applications using the same client account. Consult mBlox support for details about how this parameter can be used if the client is running several applications on one account. Used for profile-group indexed X. Required.
clientName_X	This name is used to log in to the mBlox services. It is supplied as a part of the account information. Used for profile-group indexed X. Required.
connections_X	Number of connections (sockets) for each server in profile-group indexed X. Default 1.
destBusyBadQuota	Number between 0 and 100. When this percentage of the last X number of messages (X is destBusyTestSet below) receive a destinationBusy this server will be avoided for a while. Default is 30.
destBusyTestSet	Number of consecutive messages used when testing if a server sends destinationBusy too often. Default is 50.
encodedTextBody	If true the client application must do all of its encoding and escaping of forbidden characters in the body before sending the body to the Client API (see section 3.4.2.1). Only applies to messages of type text and alphanumeric originators. Default false .
encodedUnicodeBody	Only applies to Unicode messages. If true the client application must encode all characters in the body before sending the body to the Client API (see section 3.4.2.3). If false no character may be encoded. Default is false .

Parameter	Description
failoverOnRejects	Comma separated numbers that specify reject-codes. If a message is rejected with one of these rejectCode values, the Client API will try to avoid this server for a while.
keepAliveInterval	How often keep-alive requests are sent on each socket. In seconds. Default 90.
keepAliveManagerSleep	How often the keep-alive mechanism checks the status to see if any action needs to be taken, such as sending a keep-alive request. In milliseconds. Default 1000.
keepAliveWait	Time before which a keep-alive response must have been received from the mBlox servers. In seconds. Default 30.
loadShare	If true traffic will be load shared on servers instead of just using the primary server. Default true .
maxBlockingSequences	Only applies when using blocking send (see section 4.5). Maximum number of sequences that the sendSMS() methods can block on at one time. Should match the number of threads that could possibly call the sendSMS() methods, if blocking sendSMS() is used (see section 4.5). Default 40.
password_X	Password used to log in for profile-group indexed X. Required.
port_profile_X_server_Z	Port used to connect profile-group with index X for server indexed Z. Must be supplied for each existing combination of X and Y in the IP-numbers below. See section 8.2.1 for details. Required.
profile_X	Profiles mapping to profile-group indexed X. The value is a comma-separated list of profiles that should be sent on this profile-group. See section 8.2.1 for details. Required.
profile_X_ip_Y_server_Z	IP number with index Y for server Z on profile-group index X. Each profile-group index specified in the profile_X parameter above must have an IP with the same index here. The index Z is used to specify the number of the servers in the profile-group. See section 8.2.1 for details. At least one IP per profile-group is required.

Parameter	Description
repositorySize	There is a repository within the Client API where messages can be stored for resending. This parameter defines the size of that store. Default 50.
sequenceCacheSize	The size of all sequence caches. These are used to filter out incoming MO and StatusInds with the same sequence . Default 50.
stayBadFor	Some events trigger the Client API to avoid sending to a server. This parameter specifies how long a server is avoided. In seconds. Default 1800.
windowBlockFor	This value controls how long each sendSMS() method will block while waiting for an available slot to use for transmitting the message to the mBlox servers. The lower the value, the more often the sendSMS() methods will fail (return -1) and the higher the value the longer each sendSMS() will block while waiting for an available slot. Note that this does not exactly define how long the sendSMS() method might block, it influences the length of this period. In milliseconds. Default 30000.
windowManagerSleep	How often the resend mechanism checks the status to see if any action needs to be taken, for example resending a message. In milliseconds. Default 1000.
windowMaxReservationTime	How long an SMS can stay within the retry cycle in the Client API. When the time runs out a call to the SubmitRej() callback is made. In seconds. Default is 86400 (24 hours).
windowRetriesToReconnect	When this amount of consecutive resends have been performed on a server connection, the server connection is reconnected. The reason for resending must be that there was no response from the mBlox servers. Default 5.
windowRetryInterval	The period after which an SMS is retried, if no response was received from the mBlox servers. In seconds. Default 60.
windowSize	Number of sequences in the windows. There is one window for each server connection. This setting can on rare occasions be changed to improve the throughput. Never be changed without consulting the mBlox Support. Default 10

Parameter details

Here are descriptions for all parameters from the `sendSMS()` calls and all callback methods. All parameter values can be left out (sent as '-1', **null** or **false** depending on data-type).

Parameter	Description
at	<p>The time when this event occurred.</p> <p>Absolute time in milliseconds since 0:0:0 1/1/1970 (number containing 13 digits).</p> <p>deliverRec(): The time is supplied by the SMSC representing when the SMS was sent. It is not always supplied by the SMSC. If not it is replaced with the time when the SMS hit the mBlox servers. Note that if this is supplied by the sending SMSC, it will be referenced to the time zone to which the SMSC's real time clock has been set, which may not be UTC.</p> <p>statusInd(): Time when this StatusInd hit the mBlox servers.</p>
billingRef	<p>The client invoice can be potentially split in different sections. This parameter can be used to determine how the traffic will be broken down. Clients can for instance supply "reminder" and "chat" to distinguish between the traffic of two different applications in the invoice. Speak to your account manager to find out whether this option is available</p> <p>The max length is 9 and the only characters allowed are a-z lowercase and digits. If these restrictions are not met, the mBlox servers will reject the message.</p>
blocking	<p>For ordinary sending use '-1'. Use this parameter to block the sendSMS() method until the mBlox servers have confirmed / rejected the message. The value is the number of milliseconds the method should wait, zero means wait forever. Note that the treats the parameter as a target and does not block for exactly this amount of time. See section 4.5 for details.</p>

Parameter	Description
body	<p>The body of the SMS. Should be supplied differently depending on the type of SMS. Type of SMS is defined in the parameter bodyEncoding.</p> <p>Text: If configuration parameter encodedTextBody is true some characters are considered illegal and must be encoded. Then all characters can be coded as ":3A" where 3A is the hexadecimal ASCII value for that char. See section 4.4.2 for details and the appendix for allowed characters.</p> <p>Data: Body is supplied in the format":3A:3B:3C", where 3A,3B,3C is the hexadecimal value for the bytes supplied. See section 4.4.2 for details.</p> <p>Unicode: If configuration parameter encodedUnicodeBody is false, all characters are supplied as a normal String. If true, all characters must be supplied as ":003A:003B:003C". See section 4.4.2 for details.</p> <p>deliverReq: The body is never encoded in any way. All characters are supplied as Java characters within the Java string.</p>
bodyEncoding	<p>Specifies the type of message. "Text" for an ordinary text message, "Data" for binary and "Unicode" for Unicode messages. Default is "Text".</p>
clientRef	<p>Currently not in use.</p>
confPath	<p>Absolute file-path to the conf.dat file. For example "/usr/local/myapp/clientapi/conf.dat" or "c:\myapp\clientapi\conf.dat".</p>

Parameter	Description
contentType	<p>Tag used to identify the type of content being sent. This is required for certain products. Please refer to the relevant product documentation to see if this is required and consult the mBlox Content Type Guide for the list of allowed values.</p> <p>This is an integer value. If not included -1 is submitted as default.</p>
deliverer	<p>Specifies from which deliverer this MO was delivered to the mBlox servers. This may be used for premium clients to know from which originating operator the MO came. Please consult the product description.</p> <p>The list of possible values will be supplied with the account details.</p>
destination	<p>Destination msisdh supplied as a String. Always in format "0046709001 122", with two zeros before the country code. If this value is left out in the sendSMS() methods, the message will be rejected by the mBlox servers.</p> <p>deliverRef(): The number the that MO message was sent to, for example which short-code.</p> <p>submitRef(): Destination of the originating SMS.</p>
expirationTime	<p>Used to determine how long the message is stored within the SMSC waiting for delivery. If the message could not be delivered before this time (for example, the handset is turned off) the message expires and the client notified through a StatusInd (if the application use status-reporting).</p> <p>Absolute time in seconds since 0:0:0 1/1/1970 UTC (number containing 10 digits).</p> <p>Each product has a maximum expiry time. If an expirationTime is supplied that is beyond the product setting, the mBlox servers will change the expirationTime and forward the message to the network anyway.</p>
listener	<p>The client must develop a class that implements the interface called <code>com..mblox..gateway.clientapi.Listener</code>. This class will have predefined methods (defined in the interface) that will be called by the Client API upon certain events. This parameter should be one instance of the class. The Client API will call the method in this instance.</p>

Parameter	Description
log	From which log-file this message originates.
logPath	Absolute file-path to where log-files location. For example "/usr/local/myapp/clientapi/logs/" or "c:\myapp\clientapi\logs\".
logTrace	True if the trace-log should be used.
message	systemStatusInfo() : Description of why the callback method was called. log() : Includes the string message that should be logged.
messageCode	Identifies the type of systemStatusInfo() call. All possible values are found as public-static-final parameters within the MbloxCient class.
mqFrom	The username used when the Client API logs into the socket on which the originating message was sent.
msgReference	Unique reference on the SMS. submitConf() : This is the unique reference on the sent message as specified by the mBlox servers. All StatusInds that arrives will reference to this message using this id. statusInd() : This parameter matches the reference in the originating MT SMS. deliverReq() : This is a reference on this message as supplied by the originating SMSC. This value is seldom supplied.
multiPart	This parameter is currently not in use. -1 should be used.
occurenceTime	When this event actually occurred as supplied by the SMSC. Absolute time in seconds since 0:0:0 1/1/1970 UTC (number containing 10 digits).

Parameter	Description
originator	<p>String value that may be left out. Defines from where the SMS came as viewed on the handset.</p> <p>sendSMS(): Supplied differently depending on the parameter originatorType. If originatorType is '0' or '-1', the originator is supplied as "46709001122". A plus sign will be added by the handset. If the originatorType is '3', the originator is supplied as "72300" (Network Shortcode format). No plus sign will be added by the handset. Finally if the originatorType is '5' the originator is supplied as "hello" (Alphanumeric). In this case character encoding works the same as for the body, in that if parameter encodedTextBody is false, all illegal characters must be encoded in the format "[:3B" where 3B is the hexadecimal ASCII value for that character. Maximum length is 11 for "alphanumeric" and 15 for "numeric and Shortcode" See appendix for allowed characters.</p> <p>statusInd(): The destination to which the originating SMS was sent.</p> <p>deliverReq(): The MSISDN number of the handset from which the MO was sent. Note that the format is not explicitly defined. It is transparently passed on from the operator. It might vary depending on which operator the MO came from.</p>
originatorType	<p>The type of the originator parameter. 0 is "numeric", 3 is "short-code" and 5 is "alphanumeric". Default is "numeric".</p>

Parameter	Description
operator	<p>sendSMSO: The destination network / operator. Some MT products allow the option of bypassing mBlox's routing engine by explicitly declaring the destination operator. This field may also be used with Premium SMS to identify the network being sent to. Consult product documentation for full usage details and the list of relevant operator id values.</p> <p>deliverReqO: The originating network / operator. It is an important parameter for certain Premium SMS implementations. Consult product documentation for full usage details and the list of relevant operator id values.</p> <p>submitConfO: The network / operator to which the message was submitted. It is an important parameter for certain Premium SMS implementations. Consult product documentation for full usage details and the list of relevant operator id values. If no operator id is provided the value of this parameter will be '-1'.</p>
pid	Use -1 if this parameter is unfamiliar. This is the protocol_identifier GSM parameter. Read GMS0340 for details.
profile	Different products are configured on a single mBlox account by use of multiple profiles. This parameter specifies which product should be used to send the message. The different possible values are supplied along with the account details. The default value '-1' can be used and in this case the lowest numerical profile configured will be used. It is extremely important to use profiles accurately. Some Premium SMS products use profile to determine destination operator. In this case incorrect use can result in non-charging events. The product documentation will explain whether the client should use this parameter for Premium SMS.

Parameter	Description
reason	<p>submitRej(): String that gives a detailed explanation to why a message was rejected by the mBlox servers (see the appendix – Reject codes). There are frequent additions to this list.</p> <p>statusnd(): Numeric value that corresponds to the status parameter.</p> <p>1 – Buffered, phone related (e.g. handset switched off)</p> <p>2 – Buffered, operator unknown (e.g. congestion, HLR fault)</p> <p>3 – Acked, message has been accepted by an SMSC for deliverance.</p> <p>4 – Delivered, message has successfully been delivered to the handset.</p> <p>5 – Failed, the message could not be delivered to the handset.</p> <p>6 – Unknown, the sending SMSC did not supply any information about the success of the deliverance.</p> <p>These codes may be added to in the future.</p>
rejectCode	Number that identifies the reason for rejecting the message. See the appendix – Reject codes .
reply	This parameter is currently not in use. Use –1.
retry	0 means that the rejection is permanent and the message send should never be attempted again. If 1 the message could be accepted later and should be retried.

Parameter	Description
sequence	<p>Identification number for one specific communication transaction between the client application and the mBlox servers. Used for MT, MO and StatusInds.</p> <p>sendSMS(): Unique reference for this SMS up until when the mBlox servers have accepted the message. Must be a positive number between zero and the maximum value of an integer.</p> <p>submitConf(): Will include the sequence as supplied when this message was sent. This is how the submitconf and the submission can be matched. If StatusInds are received later for this message the msgReference must be used to match these up. This is how the StatusInds will refer to this message.</p> <p>submitRej(): The same as for SubmitConf () above. However this time there will not be any StatusInds coming later, since the message was rejected.</p> <p>statusInd(): Unique identifier for this specific StatusInd message. Thus has nothing to do with the sequence supplied in the originating sendSMS() call.</p> <p>deliverReq(): mBlox unique identifier for this MO.</p>
serviceDesc	<p>Used to provide a brief description of the service being provided, which is visible to the end user e.g. on their mobile phone bill. This is required on certain premium products. Please refer to product implementation guidelines for information on what is required. Alphanumeric string of max ten characters.</p>
serviceId	<p>Used on premium products to identify the service or campaign that is associated with the message. Please refer to implementation guidelines for the product specific requirements.</p> <p>An integer identifying the service. Default value of -1 if not included.</p>

Parameter	Description
sessionId	<p>Used with premium services to specify the ID of the session. This value can be received in an incoming MO and should then be included in the outgoing MT for the same session.</p> <p>When this parameter is supplied in the MO, the originator value might be empty. Then the corresponding MT should have an empty destination value.</p>
smsClass	<p>Use -1 if this parameter is unfamiliar. 1 is default. 0 can be used to obtain a "flash to screen" message. This value will be used to set the 'class' bits of the DataCodingScheme GSM parameter. Read the GSM0338 specification for details.</p>
status	<p>String that describes the current status of the message. It could either be a "final" or "intermediate" status.</p> <p>"final" means that the message history is closed and no further transactions will occur relating to it. There will always be one "final" StatusInd for every sent message (if status-reporting is on for that message). Possible values are:</p> <p>"delivered": The message was successfully delivered to the handset.</p> <p>"failed": The message could not be delivered to the handset.</p> <p>"unknown": The SMSC used did not supply any information about the success of the delivery.</p> <p>"Intermediate" status reports may occur many times before a "final" state is reached. They report the current status of the message.</p>
status	<p>"acked": Message has been confirmed for delivery by an SMSC.</p> <p>"buffered": Message is buffered. There could be many reasons, for example "handset out of reach". The SMSC will retry the message.</p>
statusReport	<p>Indicates whether status reporting is requested for this message. Note that not all products support status reporting. If this parameter is false, no StatusInds will arrive for this message. Note that when using the sendSMSShort() method, statusReport is set automatically to false.</p>

Parameter	Description
tags	Tags are used to carry specialized parameters which are specific to certain product implementations and are therefore not covered by the predefined parameters in this table. The client should consult the relevant product documentation to determine whether the use of tags are required and to obtain the valid tag/value pairs.
tariff	Used with premium services to specify the tariff on a message. Possible values are supplied with the account/product information.
udh	User data header, often used when sending binary messages (bodyEncoding set to "Data"). This is needed for instance when sending ringtones, logos etc. The parameter is sent as ":A1:A2" where A1,A2 represents the hexadecimal value for the bytes.
value	Value that is supplied differently depending on the messageCode.

Reject codes

This table contains the values returned as rejectCode in the **submitRej()** call:

Value	Description
3	The prefix of the destination number is blacklisted
6	The mBlox server is currently too busy.
7	You are not allowed to use this value in the reply parameter.
8	Syntax error, the parameters are not specified correctly.
10-12	General problem.
13	You cannot send using this profile .
14	The username used when you logged in is faulty.
15	You are not allowed to send binary messages using this profile .
16-31	There is currently no available route for this message.
32	Error while trying to use a special functionality named "locally forced".
33	Error while trying to use a special functionality called "originator indexing". The originator for this index could not be found.
34	The available destination queues are full.
35	The client account is currently blocked on this server.
36	You supplied an illegal value in the billingRef parameter.
37	You are not allowed to send messages to this destination operator.
38	There is currently too much incoming traffic on this destination operator.
39	This sequence has been used earlier among the X last sent messages on this server. X is most likely 50, but it can vary.
42-43	Premium services only. The destination cannot be found for your supplied values. Verify the values used in the originator , tariff and operator fields.
44	There is no available route that supports the requested message features.
45	This product does not support the supplied features. Please verify your use of the profile parameter.

Value	Description
46	The text content of this message is prohibited on this product.
47	The number portability operator lookup failed.
48	The operator parameter is required when a MT is sent through the PsmsPlex application.
49	The MT could not be routed in the PsmsPlex application.
50	An unknown exception encountered when handling tags.
51	The tag is not configured in the Database.
52	The name of the tag is not valid.
53	The value of the tag is not valid.
54	The tag is not allowed for this destination operator.
55	Syntax error in tag/value pair.
56	Too many tags are submitted in the message.
57	A tag is duplicated.
58	Invalid ServiceDesc. Do not retry.
59	Default ServiceDesc not configured. Do not retry.
60	Invalid ContentType. Do not retry.
61	Default ContentType not configured. Do not retry.
62	ContentType not configured for Operator. Do not retry.
63	Invalid ContentType. Do not retry.
64	Default ServiceId not configured. Do not retry.
65	ServiceId not configured for Operator. Do not retry.

Example of conf.dat

```
# version 2.0
#-----

# Profile Connection Groups
profile_0 -1
connections_0 1
clientName_0 loginName
applicationName_0 myApplication
password_0 verySecret
profile_0_ip_0_server_0 qos1.mblox.com
profile_0_ip_1_server_0 qos1-sec.mblox.com
profile_0_ip_0_server_1 qos2.mblox.com
profile_0_ip_1_server_1 qos2-sec.mblox.com

profile_1 1033,1034,1035
connctions_1 1
clientName_1 loginName
applicationName_1 myApplication
password_1 verySecret
profile_1_ip_0_server_0 psms2.mblox.com
profile_1_ip_1_server_0 psms2-sec.mblox.com

port_profile_0_server_0 6001
port_profile_0_server_1 6001
port_profile_1_server_0 5001

# Window Settings
windowSize 10
windowRetryInterval 60
windowMaxReservationTime 33200
windowRetriesToReconnect 5
windowManagerSleep 1000
windowBlockFor 5000

# Miscellaneous Settings
keepAliveInterval 90
keepAliveWait 30
keepAliveManagerSleep 1000
sequenceCacheSize 50
maxBlockingSequences 40
loadShare true
stayBadFor 1800
destBusyTestSet 50
destBusyBadQuota 30
repositorySize 25
failoverOnRejects 35
```

```
# Text Encoding
encodedTextBody false
encodedUnicodeBody false
```

Character map

On most products ordinary text messages can only include the characters within the IA5 character map. These can be encoded in the way described in **section 4.4.5**. Characters outside of the IA5 set may be supported on some products, please consult the product documentation. Transparent Unicode sending is only available on some products. Sending in Unicode limits the number of characters that can be sent in a message.

An originator of type "alphanumeric" may include a-z, A-Z, digits and special characters !"#%&'()*+,-./?<>. The space character is not allowed. As there is no industry standard originator-encoding format for all handsets and SMSC's, there is no guarantee that other special characters than the ones above will be displayed correctly. All characters in the map below may be supplied to the mBlox servers, but we can only guarantee that the ones listed above are displayed correctly in the handset.

IA5-value
Character
Unicode-value

IA5 character Map

0000 @ 0040	0010 □ 0394	0020 <sp> 0020	0030 0 0030	0040 i 00a1	0050 P 0050	0060 ž 00bf	0070 p 0070
0001 £ 00a3	0011 _ 005f	0021 ! 0021	0031 1 0031	0041 A 0041	0051 Q 0051	0061 a 0061	0071 q 0071
0002 \$ 0024	0012 □ 03a6	0022 " 0022	0032 2 0032	0042 B 0042	0052 R 0052	0062 b 0062	0072 r 0072
0003 ¥ 00a5	0013 □ 0393	0023 # 0023	0033 3 0033	0043 C 0043	0053 S 0053	0063 c 0063	0073 s 0073
0004 è 00e8	0014 □ 039b	0024 à 00a4	0034 4 0034	0044 D 0044	0054 T 0054	0064 d 0064	0074 t 0074
0005 é 00e9	0015 □ 03a9	0025 % 0025	0035 5 0035	0045 E 0045	0055 U 0055	0065 e 0065	0075 u 0075
0006 ù 00f9	0016 □ 03a0	0026 & 0026	0036 6 0036	0046 F 0046	0056 V 0056	0066 f 0066	0076 v 0076
0007 ì 00ec	0017 □ 03a8	0027 ' 0027	0037 7 0037	0047 G 0047	0057 W 0057	0067 g 0067	0077 w 0077
0008 ò 00f2	0018 □ 03a3	0028 (0028	0038 8 0038	0048 H 0048	0058 X 0058	0068 h 0068	0078 x 0078

IA5 character Map (continued)

0009 Ç 00c7	0019 □ 0398	0029) 0029	0039 9 0039	0049 l 0049	0059 Y 0059	0069 i 0069	0079 y 0079
000a <lf> 000a	001a □ 039e	002a * 002a	003a : 003a	004a J 004a	005a Z 005a	006a j 006a	007a z 007a
000b ∅ 00d8	001b <esc> 001b	002b + 002b	003b ; 003b	004b K 004b	005b Ä 00c4	006b k 006b	007b ä 00e4
000c ø 00f8	001c Æ 00c6	002c , 002c	003c < 003c	004c L 004c	005c Ö 00d6	006c l 006c	007c ö 00f6
000d <cr> 000d	001d œ 00e6	002d - 002d	003d = 003d	004d M 004d	005d Ñ 00d1	006d m 006d	007d ñ 00f1
000e Å 00c5	001e ß 00df	002e . 002e	003e > 003e	004e N 004e	005e Ü 00dc	006e n 006e	007e ü 00fc
000f å 00e5	001f É 00c9	002f / 002f	003f ¿ 003f	004f O 004f	005f § 00a7	006f o 006f	007f à 00e0

Send examples

Text message with originator

```
myMbloxCient.sendSMSDetailed("0046709001122","The cat in the hat", -1, "Example",5,-1,-1,false,-1,-1,-1,null,null,120000,null,-1,-1,null,null,null);
```

Nokia ringtone

```
String udh=":06:05:04:15:81:15:81";
```

```
String body;
```

```
body = ":02:4A:3A:5D:25:B9:91:A5:85:B9:84:04:00:B9:28:92:A8:20";
```

```
body += ":C2:CC:30:C2:0C:51:10:24:83:49:A0:83:0A:B0:B0:08:24";
```

```
body += ":C2:08:30:D3:0E:30:83:14:58:09:20:83:48:20:C3:8C:51";
```

```
body += ":12:25:44:C8:95:15:22:4A:A0:83:0B:30:C3:08:31:44:40:92";
```

```
body += ":08:51:34:24:83:14:55:89:51:60:A4:C2:08:30:C3:14:55:29:20";
```

```
body += ":C5:13:42:48:30:C3:14:55:29:20:C5:13:42:48:30:C3:14:59:29";
```

```
body += ":20:C5:15:42:48:31:44:D8:95:11:22:48:20:00";
```

```
myMbloxCient.sendSMSDetailed("0046709001122",body, -1,"Example",5,-1,-1,false,-1,-1,-1,"Data",udh,120000,null,-1,-1,null,null,null);
```

Notes

Please use this space to record any important information or further questions you have on the MSIP Interface.

Notes (continued)...