

mGw2 Client for Java User Guide

CONTENTS

1	Introduction	3
2	Service Managers	3
2.1	Manager Types	3
2.2	Manager Common Properties	3
2.3	Manager Instantiation	4
2.3.1	Constructor arguments	4
2.3.2	Setting Properties	4
2.3.3	Mixed	5
3	Messaging Services	6
3.1	Sending Messages	6
3.1.1	Sending SMS message	7
3.1.2	Sending MMS Message	7
3.1.3	Sending Logo Message	7
3.1.4	Sending RingTone Message	8
3.1.5	Sending SMPP Message	8
3.1.6	Sending WAP Push SI Message	8
3.1.7	Extensions for Sending Messages	9
3.2	Acquire Messages Delivery Status	9
3.3	Retrieve Messages (Pull Model)	10
3.4	Receive Messages (Push Model)	12
3.4.1	Install AXIS Message Receiver	12
3.4.2	Registering Message Listener	14
4	Charging Services	15
4.1	Service Limitations	15
4.2	Charging Examples	15
4.2.1	Charge Amount	15
4.2.2	Reserve, Commit or Release Charge	15
5	Location Services	17
5.1	Finding User Location	17
6	Manager Exceptions	18
7	Logging	18
8	Additional Notes	18

1 INTRODUCTION

Message Gateway (mGw) provides various functionalities through its remoting services. These services use standardized or custom protocols for communication. Examples of such protocols are SOAP for web services, SMPP and many more. They can become too complex and cumbersome to use. mGw client is user friendly library for communicating with mGw remoting services.

This document describes mGw2 client v1.0.6.

2 SERVICE MANAGERS

2.1 Manager Types

Different services are u via `Managers`. There are three `Managers`, each for every service exposed by mGw remoting interfaces:

- Message Manager – sending and receiving messages
- Charging Manager – charging funds to user
- Location Manager – finding user location

Manager Interfaces and Implementations

Interface	Implementation	Used for
<code>MessageManager2</code>	<code>MessageManager2Impl</code>	Sending and receiving messages
<code>ChargingManager2</code>	<code>ChargingManager2Impl</code>	Charging user with some amount
<code>LocationManager2</code>	<code>LocationManager2Impl</code>	Finding user location

2.2 Manager Common Properties

Every type of `Manager` uses SOAP to communicate with mGw. There are some common properties that each manager must set before it can communicate with mGw.

- **mGw service URL**

To be able to communicate with mGw, manager must specify URL on which various mGw services are exposed. This URL is actually a prefix to URL at which services are exposed. Port name is not included in this URL, but appended automatically by the client depending on the type of request and type of message.

- **Authentication**

To call mGw service, manager must authenticate itself. Manager authenticates itself with username/password pair. Every manager can set default username/password pair used for every call. There is also copy of every call in which new username/password pair can be specified overriding default values.

- **Timeouts**

Since every call is in effect network communication there are plenty of reasons why mGw can't be reached (no network connection, firewall blocking traffic etc). Every manager has timeout property which defines maximum time in milliseconds waiting for mGw to send response, after which call is abandoned and exception is thrown.

- **Receive threading model**

When manager receives message from mGw it has to send message to registered listeners for processing. Since processing can take long time there are two possibilities. To spawn a new thread and return response to mGw immediately or to wait until processing is done and then return response to mGw. Both possibilities have their strengths and weaknesses. It is up to developer to decide which approach is better for him. The default operation is to spawn new thread for processing and returning response to mGw immediately.

Common Manager Properties

Property	Description
serviceURL	mGw service URL
username	mGw service authentication username
password	mGw service authentication password
timeout	connection timeout in milliseconds (default is 30,000)
threadedListener	new received message spawn new thread for processing (default is true)

NOTE: To obtain connection parameters (URL, username/password pair) contact your mGw administrator or your mobile operator contact person.

2.3 Manager Instantiation

Before any service call can be made appropriate Manager must be instantiated. There are several ways Manager can be instantiated.

2.3.1 Constructor arguments

Via constructor you can specify URL and username/password pair.

```
MessageManager2 manager = new MessageManager2Impl(url, user, pass);
```

2.3.2 Setting Properties

There is also possibility to create Manager without constructor arguments and set properties after.

```
MessageManager2 manager = new MessageManager2Impl();  
manager.setServiceURL(url);  
manager.setUsername(user);  
manager.setPassword(pass);
```

2.3.3 Mixed

You can mix constructor arguments and setting properties. There is also possibility to override previous settings by setting properties.

```
MessageManager2 manager = new MessageManager2Impl(url);  
manager.setUsername(user);  
manager.setPassword(pass);
```

3 MESSAGING SERVICES

To use messaging services, instantiate `MessageManager2Impl`. This is implementation of `MessageManager2` interface that uses SOAP to call mGw messaging services.

There is also `MessageManager` which is older interface now deprecated and was used for previous version of mGw. If you already have application that uses `MessageManager` it can still be used but you have to warn mGw administrator that you are using legacy interface since legacy interfaces are configured differently on mGw.

Use provided URL and username/password pair to instantiate `MessageManager2Impl`.

```
String url = <provided URL>;
String user = <provided Username>;
String pass = <provided Password>;
MessageManager2 manager = new MessageManager2Impl(url, user, pass);
```

Message manager is configured with single URL and can be reused for complete communication with single server. Multiple message managers can be created in application if communication with multiple servers is required.

3.1 Sending Messages

After manager instantiation, `sendMessage` method is used for sending messages. `sendMessage` method expected parameters are: message to send, array of destination addresses, originator address, priority and charging.

```
manager.sendMessage(
    message, destination, originator, priority, charging);
```

NOTE: See javadoc for more information about `sendMessage` method on `MessageManager2` interface.

Message Sending Interfaces and Implementations

<code>hr.tis.mgw.client.message.MessageManager2</code>	interface
<code>hr.tis.mgw.client.message.MessageManager2Impl</code>	implementation
<code>hr.tis.mgw.client.message.MessageManager</code>	deprecated interface
<code>hr.tis.mgw.client.message.MessageManagerImpl</code>	deprecated implementation

Message Types and Their Use

Message Type	Use
<code>hr.tis.mgw.client.message.SmsMessage</code>	SMS message up to 460 characters
<code>hr.tis.mgw.client.message.MmsMessage</code>	MMS message with various textual and binary content

<code>hr.tis.mgw.client.message. LogoMessage</code>	Black and white logo message
<code>hr.tis.mgw.client.message. RingToneMessage</code>	Ring tone melody in RTTTL format
<code>hr.tis.mgw.client.message. SmppMessage</code>	SMPP message
<code>hr.tis.mgw.client.message.wap. PushSIMessage</code>	Wap Push Service Indication

3.1.1 Sending SMS message

Create new `SmsMessage` containing message text. Priority is ignored for SMS message. For use of charging parameter see Charging section.

```
SmsMessage sms = new SmsMessage("sms message text");
manager.sendMessage(sms, destinations, originator, null, null);
```

3.1.2 Sending MMS Message

Create new `MmsMessage` containing `MmsParts`. `MmsMessage` holds subject of message. `MmsPart` hold textual or binary content. Attach `MmsParts` to `MmsMessage`. While sending specify priority of message. For use of charging parameter see Charging section.

```
byte[] bytes = ... byte array of loaded JPEG image from file ...
MmsMessage mms = new MmsMessage("subject");
MmsPart text = new MmsPart("This is text part of MMS message");
MmsPart image = new MmsPart(bytes, "image/jpeg");
mms.addPart(text);
mms.addPart(image);
manager.sendMessage(mms, destinations, originator, Priority.DEFAULT,
null);
```

Priority Values

Priority	Value
Default	<code>hr.tis.mgw.client.message.Priority.DEFAULT</code>
Low	<code>hr.tis.mgw.client.message.Priority.LOW</code>
Normal	<code>hr.tis.mgw.client.message.Priority.NORMAL</code>
High	<code>hr.tis.mgw.client.message.Priority.HIGH</code>

NOTE: Binary content in this example was JPEG image, but any type of binary content can be used instead.

3.1.3 Sending Logo Message

Create new `LogoMessage` containing byte array representing image. Image can be in one of following formats: jpeg, gif or png. The image will be adapted, resized

to logo size and colors removed to black and white. Be careful what image resolution and how many colors it has since after adaptation result can be unrecognizable. Since various handset models have different image encoding, handset encoding format must be specified via format parameter. Priority is ignored for Logo message. For use of charging parameter see Charging section.

```
byte[] bytes = ... byte array of loaded image from file ...
LogoMessage logo = new LogoMessage(bytes, Format.EMS);
manager.sendMessage(logo, destinations, originator, null, null);
```

Format Values

Sms Format
hr.tis.mgw.client.message.Format.EMS
hr.tis.mgw.client.message.Format.SMART_MESSAGING

3.1.4 Sending RingTone Message

Create new `RingToneMessage` containing melody in RTTTL format. Since various handset models have different ring tone encoding, handset encoding format must be specified via format parameter. Priority is ignored for ring tone message. For use of charging parameter see Charging section.

```
String ringtone = <RTTTL melody>;
manager.sendMessage(ringtone, destinations, originator, null, null);
```

3.1.5 Sending SMPP Message

Create new `SmppMessage` containing smpp message as binary byte arrays. SMPP message must be split into appropriate chunks and placed inside binary byte arrays in right order as would be if message is sent via smpp protocol. Originator and destination parameters are ignored. Any value specified will not have effect on message delivery. Originator and destination address are specified in SMPP message content. Also priority is ignored for `SmppMessage` message. For use of charging parameter see Charging section.

```
List content = ... byte[] containing SMPP message ...
SmppMessage smpp = new SmppMessage(content);
manager.sendMessage(smpp, null, null, null, null);
```

3.1.6 Sending WAP Push SI Message

Create new `PushSIMessage` containing text and uri (this is minimum requirements). Priority is ignored for `PushSIMessage` message. For use of charging parameter see Charging section.

```
PushSIMessage push =
    new PushSIMessage("Check this URL", new URI("www.kapsch.net"));
manager.sendMessage(push, destinations, originator, null, null);
```


3.1.7 Extensions for Sending Messages

Client API and ParlayX enables clients to send messages that might be forwarded on to messaging center (SMSC) as multiple messages. This number of messages sent to SMSC might be used for reporting and billing purposes. Also this number might be different then the number of sending requests client perform on mGw.

To resolve this issue, extensions are implemented, both in client API and on web service interface exported by mGw.

Methods take same parameters and type of messages as described before in this chapter. Only difference is in their return type.

```
manager.sendMessageWithResult(  
    message, destination, originator, priority, charging);
```

Method returns `SendResult` structure.

SendResult

Property	Description
messageId	Message ID
count	Number of messages forwarded to single destination, e.g. if long SMS sent via client is split into 3 SMS messages when sending to SMSC, and is sent to 2 destinations, value of this parameter would be 3.

NOTE: Extension might not be supported by some operators, contact mGw administrator to verify this.

3.2 Acquire Messages Delivery Status

To acquire message delivery status `MessageId` obtained after sending message is used as parameter to `getMessageDeliveryStatus` method. Since message can be sent to many destination addresses, for every address there is one corresponding message delivery status. `DeliveryStatusInfo` contains destination address and status of message.

```
SmsMessage sms = new SmsMessage("text");  
MessageId mid =  
    manager.sendMessage(sms, destinations, originator, null, null);  
DeliveryStatusInfo[] infos = manager.getMessageDeliveryStatus(mid);
```

Message Status

Status	Value
Delivered	<code>hr.tis.mgw.client.message.Status.DELIVERED</code>
DeliveryUncertain	<code>hr.tis.mgw.client.message.Status.DELIVERY_UNCERTAIN</code>
DeliveryImpossible	<code>hr.tis.mgw.client.message.Status.DELIVERY_IMPOSSIBLE</code>
MessageWaiting	<code>hr.tis.mgw.client.message.Status.MESSAGE_WAITING</code>

NOTE: See javadoc for more information about `getMessageDeliveryStatus` method on `MessageManager2` interface.

3.3 Retrieve Messages (Pull Model)

Call `receiveSmsMessages` to retrieve SMS messages or `receiveMmsMessages` to retrieve MMS messages. The maximum number of messages received is configured on mGw so to retrieve all messages call appropriate method until no more messages are retrieved. To retrieve messages via methods mentioned previously you have to specify registration identifier. It is used to select route (channel) over which message arrived.

Note that this interface is not meant for implementation of real time services, i.e. invoking these methods every couple of milliseconds is very resource consuming and inefficient method of receiving messages. Push model is more suited for these types of services, because server will notify client immediately when it becomes available.

```
ReceivedSms[] smses = manager.receiveSmsMessages(regId);
```

ReceivedSms Properties

Property	Description
<code>smsMessage</code>	Received sms
<code>senderAddress</code>	SMS message originator address
<code>destinationAddress</code>	SMS message destination address
<code>registrationIdentifier</code>	Route (channel) over which message arrived

```
ReceivedMms[] mmses = manager.receiveMmsMessages(regId);
```

ReceivedMms Properties

Property	Description
<code>mmsMessage</code>	Received MMS
<code>senderAddress</code>	MMS message originator address
<code>destinationAddress</code>	MMS message destination address

priority	MMS message priority
registrationIdentifier	Route (channel) over which message arrived

NOTE: See javadoc for more information about `receiveSmsMessages` and `receiveMmsMessages` methods on `MessageManager2` interface. To find out valid registration identifier contact mGw administrator or your mobile operator contact person.

Messages received this way include regular messages and delivery reports, i.e. delivery reports are encoded as `ReceivedSms` or `ReceivedMms`.

Client can use `MessageHelper` utility class with methods for recognition and creation of `DeliveryReports`.

```
ReceivedSms sms = smses[0];
if (MessageHelper.isDeliveryReport(sms)) {
    DeliveryReport deliveryReport =
        MessageHelper.createDeliveryReport(sms);
    // process delivery report
} else {
    // process sms
}
```

Same processing can be done for `ReceivedMms`.

DeliveryReport

Property	Description
senderAddress	Delivery report sender address (equal to destination address in original MT message)
destinationAddress	Delivery report destination address (short code)
messageId	Message ID (equal to message ID received when sending original MT message)
registrationId	Registration ID.
status	Message delivery status (see 3.2).

3.4 Receive Messages (Push Model)

To receive messages one has to use servlet container and in it define AXIS message receiver. All received messages will be sent to `MessageManager2` which will decide depending on registration to which listener message is to be routed for processing. If registered listener can't be found, message will be silently rejected. Only entry in log (if enabled) will be indication of message rejection. Client has to implement interface `MessageListener2` and register itself to `MessageManager2`.

MessageListener2 methods

Method	Action
<code>smsReceived(ReceivedSms sms)</code>	Called when SMS message is received
<code>mmsReceived(ReceivedMms mms)</code>	Called when MMS message is received
<code>deliveryReportReceived(DeliveryReport report)</code>	Called when delivery report is received
<code>longSmsReceived(ReceivedLongSmsSegment seg)</code>	Called when long SMS segment is received

Delivery reports are always delivered in separate callback, i.e. `smsReceived` and `mmsReceived` always receive regular messages.

NOTE: `longSmsReceived` is ParlayX extension and might not be supported by operator.

ReceivedLongSmsSegment

Property	Description
<code>registrationIdentifier</code>	Registration identifier.
<code>text</code>	Message text (single segment in concatenation).
<code>senderAddress</code>	Sender address.
<code>destinationAddress</code>	Destination address.
<code>sarMsgRefNum</code>	SAR (segmentation and assembly) message reference – uniquely identifies concatenation for given sender.
<code>sarTotalSegments</code>	Total number of SAR segments for given concatenation.
<code>sarSegmentSeqNum</code>	Index of current segment in concatenation. Starts with 1, maximum value 255.

NOTE: See Retrieve Messages chapter for descriptions of `ReceivedSms` and `ReceivedMms` structures or consult javadoc.

3.4.1 Install AXIS Message Receiver

First you have to choose servlet container. In following text Apache Tomcat version 5 will be assumed, but the same should apply to any other servlet

container. In web application which will receive messages add to web.xml file following:

```
<listener>
    <listener-class>
        org.apache.axis.transport.http.AxisHTTPSessionListener
    </listener-class>
</listener>

<servlet>
    <servlet-name>AxisServlet</servlet-name>
    <display-name>Apache-Axis Servlet</display-name>
    <servlet-class>
        org.apache.axis.transport.http.AxisServlet
    </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>

<session-config>
    <session-timeout>30</session-timeout>
</session-config>

<mime-mapping>
    <extension>wsdl</extension>
    <mime-type>text/xml</mime-type>
</mime-mapping>

<mime-mapping>
    <extension>xsd</extension>
    <mime-type>text/xml</mime-type>
</mime-mapping>
```

It will register AXIS servlet to listen on /services servlet path for incoming messages. Copy server-config.wsdd file to WEB-INF directory of your web application. File can be found in resource directory of mGw client library. You can also find web.xml example file in the same directory. Copy content of mGw library lib directory to yours web application WEB-INF/lib directory. All dependencies needed by mGw client can be found in this directory.

Start your servlet container and check if installation was successful. Open `http://<host>:<port>/<context>/services/SmsNotificationPort` URL in your browser and you should see text on screen indicating AXIS service.

3.4.2 Registering Message Listener

Assuming you have implemented `MessageListener2` interface the next step is to register your message listener. Listener is registered via `registerMessageListener` method on `MessageManager2` interface. There are two methods for registering listeners, one that use default username/password pair for authentication and one where you can specify username/password pair for authentication.

It is important to understand how messages are received and when and why username/password pair is used for authentication.

- **receiving SMS message**

The whole message containing text is received when request from mGw is accepted on client side.

- **receiving MMS message**

Only message reference is received when request from mGw is accepted on client side. To get content of MMS message client library must call mGw service with received reference to retrieve message content. When this call is made, username/password pair is used for authentication. Note that if MMS contains only single content part and this part is text, then text is pushed from server in notification and additional request from client to server to retrieve content is not required.

To register listener you have to specify registration identifier that represents route (channel) over which message arrived. To receive messages delivered via any route (channel) use constant `MessageManager2.REG_ID_ANY`.

```
String regId = MessageManager2.REG_ID_ANY;
MessageListener2 listener = new YourMessageListenerImplementation();
manager.registerMessageListener(regId, listener);
```

Multiple `MessageListener2` instances can be registered with single `MessageManager2Impl`, with different registration ID. This way, client can use separate listener for messages with different short code.

Note that all parameters provided in notification callbacks (`ReceivedSms`, `ReceivedMms`, `DeliveryReport` and `ReceivedLongSmsSegment`) also contain `registrationId` value. This way, single listener can be used to receive all messages and be processed by partner in some arbitrary way.

Multiple `MessageManager2Impl` instances can be created in application, for instance if partner wants to communicate with multiple servers, hosted at different OPCOs.

Note that single listener can be registered multiple times with single or multiple `MessageManager2Impl` instances, but each time with different `registrationId`. Client API will use registration ID value to route the request to appropriate listener. Registering another listener with same registration ID overwrites previous registration and new listener will receive callbacks for this registration ID. Registration ID is not scoped per message manager instance but should be unique in client API (i.e. in general case it's not possible to receive notifications from different operators with same registration ID).

NOTE: See javadoc for more information about `MessageListener2` interface and `receiveMmsMessages` methods on `MessageManager2` interface. To find out valid registration identifier contact mGw administrator or your mobile operator contact person.

4 CHARGING SERVICES

To use charging services, instantiate `ChargingManager2Impl`. This is implementation of `ChargingManager2` interface that uses SOAP to invoke mGw charging services.

Use provided URL and username/password pair to instantiate `ChargingManager2Impl`.

```
String url = <provided URL>;
String user = <provided Username>;
String pass = <provided Password>;
ChargingManager2 manager = new ChargingManager2Impl(url, user, pass);
```

4.1 Service Limitations

`ChargingManager2` interface provides many methods for charging funds to user. Not all methods have to be supported by your mobile operator. If unsupported method is called, mGw will throw exception that will be propagated to client.

NOTE: To find out which methods are supported contact your mGw administrator or your mobile operator contact person.

4.2 Charging Examples

In following examples only commonly used methods by mobile operators will be shown. Thus only amount charging will be shown. Some parameters to methods can be used differently than mentioned, depending on mobile operator billing system capabilities.

4.2.1 Charge Amount

This is most simple example that will charge funds to user. Additional parameters (`billingText` and `referenceCode`) may be left empty if not desired by mobile operator or may be used for different purpose again depending on mobile operator.

```
String user = <user>;
BigDecimal amount = <amount>;
String billingText = <text to appear on bill>;
String referenceCode = <unique request identifier>;
manager.chargeAmount(user, amount, billingText, referenceCode);
```

4.2.2 Reserve, Commit or Release Charge

More complex example involving amount reservation, sending message and commit or release of reservation depending on successful message delivery. Additional parameters have same restrictions or use as in simple charge amount example.

```
String user = <user>;
BigDecimal amount = <amount>;
String billingText = <text to appear on bill>;
```

```
String referenceCode = <unique request identifier>;  
Charging charging =  
    chargingManager.reserveAmount(user, amount, billingText);  
String[] destination = new String[] {user};  
String originator = <originator>;  
SmsMessage sms = new SmsMessage("sms message text");  
MessageId mid =  
    messageManager.sendMessage(  
        sms, destination, originator, null, charging);
```

NOTE: Notice usage of charging parameter when sending message. This will link charging information with sent message.

```
chargingManager.chargeAmountReservation(  
    charging, amount, billingText, referenceCode);
```

Or if message was not sent successfully.

```
chargingManager.releaseAmountReservation(charging);
```

For description of all others methods on ChargingManager2 interfaces please consult javadoc.

5 LOCATION SERVICES

To use location services, instantiate `LocationManager2Impl`. This is implementation of `LocationManager2` interface that uses SOAP to call mGw location services.

Use provided URL and username/password pair to instantiate `LocationManager2Impl`.

```
String url = <provided URL>;
String user = <provided Username>;
String pass = <provided Password>;
LocationManager2 manager = new LocationManager2Impl(url, user, pass);
```

5.1 Finding User Location

To find user location, call `getLocation` method. Requester parameter can be left empty. Effect of accuracy values on result depends on mobile operator infrastructure. Result of method call is location which contains user longitude, latitude, accuracy of result and time of result.

```
String user = <user>;
String requester = <requester>;
Accuracy accuracy = Accuracy.MEDIUM;
Location location = manager.getLocation(user, requester, accuracy);
```

Accuracy Values

Accuracy	Value
Low accuracy	<code>hr.tis.mgw.client.location.Accuracy.LOW</code>
Medium accuracy	<code>hr.tis.mgw.client.location.Accuracy.MEDIUM</code>
High accuracy	<code>hr.tis.mgw.client.location.Accuracy.HIGH</code>

Location Properties

Property	Description
<code>longitude</code>	User location longitude
<code>latitude</code>	User location latitude
<code>accuracy</code>	Accuracy of result values
<code>time</code>	Time when measurement was made

6 MANAGER EXCEPTIONS

Every method in every manager throws exceptions. There is one root generic exception `GatewayException` which covers all different exceptions thrown. This way only one exception needs to be caught. If desired sub exceptions can be caught explicitly. To view all possible exceptions look at the `hr.tis.mgw.client.exceptions` package in javadoc. The exceptions correspond to ParlayX exceptions. For understanding of exceptions types and usage read ParlayX for web services manual.

Usually Encountered Exceptions

Exception	Description
<code>MServiceException</code>	Thrown when mGw is unable to process request. See exception message for reason.
<code>MPolicyException</code>	Thrown when authentication has failed. Check username/password pair.
<code>MRemoteException</code>	Thrown when remote call to mGw failed (network problem).
<code>GatewayException</code>	Generic root exception.

7 LOGGING

Logging library used is Apache Jakarta Commons-Logging (<http://jakarta.apache.org/commons/logging>). Library is actually a wrapper around logging framework such as JDK's `java.util.logging` or Apache Log4J (<http://logging.apache.org/log4j>). Throughout the library logging is used extensively so be careful on level of logging used. We recommend `INFO` level since library was tested thoroughly and there is no need for you to test it again. It is also recommended to disable AXIS logging since it produce are lot of debug. In Log4J to set recommended logging level for library use following entries:

```
log4j.logger.hr.tis.mgw.client = INFO
log4j.logger.org.apache.axis = FATAL
```

8 ADDITIONAL NOTES

In library distribution you can find working samples that covers common usage of library. Please read README files in samples directory for instructions on how to use samples.