object XP

jsms

User's Guide

Table of Contents

1. About this Document

This document introduces the jSMS API and explains how to use it. Some of the topics covered are:

- The architectural model of jSMS.
- The installation of the jSMS binary version.
- Sample applications that use the jSMS library.
- Future directions.
- Copyright information.

1.1. Scope

The main chapters in this document are:

- **The jSMS Model**
  Gives a short overview of the architecture of jSMS

- **Installation**
  Describes the steps necessary to install and configure the product

- **jSMS Services**
  Gives a short description about the SMS Service classes contained in the jSMS API

- **Sample Applications**
  Shows some of the possible applications of the API. Explains how to use the API for sending and receiving Short Messages and Multimedia Messages.

- **Debugging your Application**
  Describes how to enable jSMS logging

- **FAQ**
  Compilation of frequently Asked Questions concerning installation and use of this product

1.2. Audience

This document is intended for Java Programmers starting to develop applications using the jSMS API.

2. The jSMS Model

2.1. The jSMS Software Layers

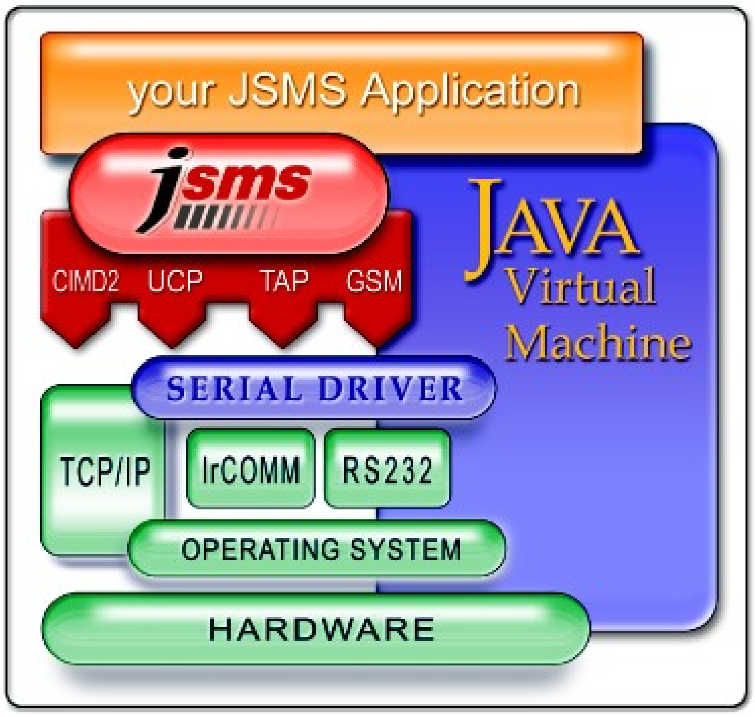


Figure 1: The jSMS Layers

Figure 1 illustrates the jSMS layers. The lowest levels are the hardware and the operating system. The next level is a Java virtual machine allowing to run portable Java applications. jSMS requires in most cases serial communication to a SMS device (except if a TCP/IP based solution is applied. The Java communication API (also known as `javax.comm`) defines the interfaces to be used for serial communication. jSMS uses this standard and can be used with any javax.comm compliant implementations. The following list shows some links to **implementations of javax.comm** that can be used with jSMS:

- Sun's reference implementation at http://java.sun.com/products/javacomm/index.html
- Serial I/O driver at http://www.serialio.com/
- RX-TX at http://www.rxtx.org/
- Any other driver that is compliant with javax.comm.

The jSMS layer sits above the JVM and javax.comm and is therefore fully portable to all Java compliant platforms. jSMS does not use any native calls except those through the javax.comm driver. The top layer is your jSMS-enabled application.

2.2. jSMS API versus Java Mail API

Sun created the Java Mail API that is used for sending and receiving mail messages. Sun's API is very flexible in terms of the underlying mail mechanism. However, the universal and abstract design made the Java Mail API quite complex to use and is mostly used for e-mail communications. The required Java classes (or .jar files) for a Java Mail API are quite heavy in terms of required disk space. As a result, if messaging features such as e-mail or SMS are needed in small applications or even embedded devices, most applications implement their own mechanism for sending/receiving e-mails.

jSMS provides a small footprint SMS and mail API that can easily be applied on embedded devices. One may ask when should which API be used. As a general term, if a complex e-mail application should be developed, one better sticks with the Java Mail API defined by Sun. For applications that don't have to send/receive complex messages (e.g. MIME) or especially for SMS applications, the use of jSMS is recommended.

3. Getting started

First of all, download a copy of the jSMS-API from http://www.objectxp.com/. The API ships as as either a gzipped tar-archive (Unix users) or as ZIP-file (Windows users).

3.1. License jSMS

After jSMS is downloaded or purchased, an email **message** is sent to the licensee **containing license properties**. These properties are **required to run jSMS** applications (for the demo version as well as the full version of jSMS). The following table shows the license properties your should receive after jSMS has been downloaded or purchased.

Property	Comment
`proto.license.user`	The user name of the licensee.
`proto.license.company`	The company name of the licensee.
`proto.license.version`	Until which version this key will be valid.
`proto.license.serial`	Serial number of the license.
`proto.license.type`	Trial / Development / Runtime license
`proto.license.sig`	The key itself

3.2. Install jSMS

The following steps are necessary to install jSMS.

Windows-Version

1. Unzip the downloaded file to your target directory (using a tool like WinZIP)
2. Change directory to your target directory
3. Make the necessary changes in the `jsms.conf`file:
   1. Copy the license properties that you **received via email** into the `jsms.conf`file
   2. Change the property `connector.serial.port`to contain the port that you have the GSM device connected to (e.g. `COM1`).
4. Install javax.comm for Windows (http://java.sun.com/products/javacomm/index.html)

Unix-Version

1. Untar the downloaded file to your target directory
2. Change directory to your target directory
3. Make the necessary changes in the jsms.conf file:
   1. Paste the license properties that you received via email into the `jsms.conf`file
   2. Change the property `connector.serial.port`to contain the port that you have the GSM device connected to (e.g. `/dev/ttyS0`).
4. Download and install a javax.comm implementation for Unix

3.3. Testing the Connection to your GSM device

Next, make sure that your Modem/ISDN-Adapter or GSM device is working properly: For devices connected to a serial port , start a terminal emulation (e.g. minicom (Unix) or Hyperterm (Windows)) and connect to the device (use the same port and baud rate you specified in the jSMS configuration file). For GSM devices connected to a terminal server, connect to the device using telnet (`telnet` *<IP-address> <port>*). When connected to the device, type in `AT` followed by the `[Return]` key. If the device does not respond with `OK`, check your cabling and/or port/hardware settings and try again.

Finally you can **test your jSMS installation** by running the `sms.sh` (Unix) or `sms.bat` (Windows) script located in the *<install-dir>*`/bin` directory. Starting this script with the argument "`-h`" displays usage information:

```
$ ./sms.sh -h
Usage: SMSService [-p protocol] [-c cfg] [-vVhasr] [-n number] [-m msg]
 Options:
   -p protocol to use (gsm [default], tap, ucp, cimd2, smpp)
   -c use specified config file
   -v verbose messages
   -V display version information
   -h display this help message
   -a wait for incoming messages (start receiving)
   -n recipient
   -o originator (sender)
   -m message to send
   -r read messages stored on device (GSM only)
   -s request status report
   -e encoding: gsm (default), binary (8bit), ucs2 (Unicode)
```

Now send an SMS to a mobile phone:

```
$ ./sms.sh -v -n "+41123456789" -m "Hello Mobile World"
Reading configuration...
Using SmsService implementation 'com.objectxp.msg.GsmSmsService'
Initializing SMS Device...
Sending message to '+41123456789'...
Shutting down...
$
```

4. jSMS Services

jSMS supports various transport facilities for sending (and receiving) Short Messages (SMS) and Multimedia Messages (MMS). Currently, the following protocols are implemented:

- **GSM** 03.38, 03.40 & 07.05 (GSM Devices with a built-in Modem)
- **UCP** (Universal Computer Protocol)
- **CIMD2** (Computer Interface to Message Distribution)
- **SMPP** (Short Message Peer-to-Peer Protocol)
- **TAP/IXO** (Paging Protocol)
- **MM1** (Multimedia Messaging using WAP/WAP PUSH and GPRS)
- **MM7** (Multimedia Messaging for Value Added Service Providers)

Additionally, jSMS also contains a small footprint SMTP client for sending e-mails.

4.1. Supported Message Types

jSMS supports the following message types:

- **7-Bit** Text: The message will be encoded as packed 7-bit data. jSMS automatically converts Messages from ISO-8859-1 to the 7-Bit GSM-Alphabet. The maximum length of a 7-bit message is 160 characters.
- **UCS2**: Unicode message, uses 2 bytes per character. The maximum length of the message is 70 characters.
- **Binary Messages**: A total of 140 bytes of binary data can be sent / received in a short message.
- **Nokia SmartMessages**: jSMS includes classes for sending and receiving Business Cards, Agenda Entries, Ring tones, Picture Messages and Operator Logos according to Nokia's SmartMessaging 3.0 specification.
- **EMS Messages**: The API supports sending and receiving Extended Messaging System (EMS) message containing pictures, animations, sound and formated text elements. An EMS Message can also be used for sending and receiving content longer than 160 characters (text) or 140 bytes (data).
- **OTA Messages**: jSMS supports the Over The Air-technology (OTA) for sending Bookmarks, Browser settings and Service Indications.
- **MMS Notification:** The MMS Notification advises the jSMS user about a MMS Message which is ready to be picked up at the MMS-Center. The MMS Notification is delivered using WAP-Push (WAP over SMS)
- **MMS Messages**: Support for SMIL- and `multipart/mixed`messages.

4.2. The SmsService Interface

The `SmsService`interface defines the methods available for sending and receiving Messages.

4.3. Windowing for Applications

Windowing for applications enables jSMS to initiate more than one operation before receiving a response from the SMSC. This increases the message throughput. `SmsService`implementations that support Windowing implement the `WindowingService`interface. For Windowing to work, your SMSC operator must enable windowing for your account.

4.4. SmsService Implementations

jSMS currently provides the following `SmsService`implementations:

- **GsmSmsService** (for GSM devices supporting GSM 03.38, 03.40 & 07.05)
- **TapSmsService** (Implementation of the TAP/IXO protocol)
- **UcpSmsService** (Universal Computer Protocol)
- **Cimd2SmsService** (CIMD2 - Computer Interface to Message Distribution)
- **SmppSmsService** (SMPP - Short Message Peer-to-Peer Protocol)

4.4.1. GsmSmsService

This class may be used to send and receive GSM Short Messages (SMS) using a GSM mobile device (e.g. a Mobile Phone). The device may be attached to the serial port or to a TCP/IP capable terminal server.

4.4.2. TapSmsService

This class may be used to send Short Messages (SMS) / Pager messages to a mobile recipient using TAP/IXO.

The message will be sent through a TAP Gateway (SMSC). The Gateway may be reached by either a modem or ISDN connection.

TAP/IXO only supports 7-Bit encoded messages. All characters in a message above ASCII(127) will therefore be truncated to a dot (.). **Notice**: Receiving Messages through TAP is not supported.

4.4.3. UcpSmsService

This class may be used to send (and receive) Short Messages (SMS) to a mobile recipient using UCP (Universal Computer Protocol).

The message will be sent through a SMS Center (SMSC) reachable by either a modem/ISDN connection or TCP/IP.

Receiving Short messages via UCP is only supported for TCP/IP connections. UCP over TCP/IP usually requires a "large volume account" at the SMSC of your Mobile Network Provider. Contact your local provider for further information.

Certain UCP specific message properties (like deferred delivery) are provided by the class `UcpMessage`. Use this class instead of `SmsMessage` to access those UCP specific features.

4.4.4. Cimd2SmsService

This class may be used to send (and receive) Short Messages (SMS) to a mobile recipient using Nokia's CIMD2 protocol (Computer Interface to Message Distribution).

The message will be sent through a SMS Center (SMSC) reachable by TCP/IP.

Like UCP, CIMD2 requires a "large volume account" at the SMSC of your Mobile Network Provider. Contact your local provider for further information.

Certain CIMD2 specific message properties (like tariff class) are provided by the class `Cimd2Message`. Use this class instead of `SmsMessage` to access those CIMD2 specific features.

4.4.5. SmppSmsService

This class may be used to send (and receive) Short Messages (SMS) to a mobile recipient using the SMPP protocol.

The message will be sent through a SMPP SMS Center (SMSC) reachable by TCP/IP.

Like UCP and CIMD2, SMPP usually requires a "large volume account" at the SMSC of your Mobile Network Provider. Contact your local provider for further information.

4.5. Creating a SMS Service Object

Before SMS messages can be sent or received, an `SmsService` object must be created and initialized. The following steps are required to construct a SMS service object:

1. Create a `java.util.Properties` or a `java.io.File` object that holds configuration data
2. Instantiate the appropriate Service Implementation (e.g. `GsmSmsService`)
3. Call the `init` method providing the Properties or File object as argument.

4.6. Initializing the Service Object

After you have created a `SmsService` object suiting your transport facility (GSM, UCP, CIMD2, SMPP, TAP/IXO), the service has to be initialized with the appropriate configuration properties. This can be achieved by passing a `java.util.Properties` object containing the configuration to the service using the `init(java.util.Properties)` method. You may also store your jSMS properties in a file and let the `SmsService` implementation read it upon initialization. Use the `init(java.io.File)` method for this purpose. When calling the `init` method without any arguments, jSMS tries to locate and load the properties by searching for a file called `jsms.conf` in your `PATH` and `CLASSPATH`. Consult the API-Documentation for further information.

For any of the `SmsService` classes, the initialization properties **must contain** at least one valid jSMS license key. A license key consists at least of the properties `user`, `company`, `serial`, `version`, `type` and signature (`sig`). You should have received a jSMS license key after registering for a trial version or purchasing the full version of jSMS.

Each `SmsService` Implementation (e.g. `Cimd2SmsService`) supports it's own set of configuration properties. A detailed description of the properties supported by each `SmsService` can be found in the Java API documentation that comes with the jSMS distribution.

4.6.1. Example for Instantiating a GsmSmsService Object

A valid properties object can be created either from a file or by setting the attributes manually. Thus, an example reading from a file would look like this:

```
java.io.File config = new java.io.File("/path/to/your/jsms.conf");
SmsService service = new GsmSmsService();
service.init(config);
```

The configuration file `jsms.conf` should look similar to the following example:

```
gsm.license.company=your_company
gsm.license.user=your_name
gsm.license.serial=your_serial
gsm.license.version=2.x
gsm.license.type=TRIAL LICENCE
gsm.license.sig=your_key

sms.gsm.connector=SERIAL
connector.serial.port=COM1
connector.serial.bps=19200
```

The configuration properties may also be **embedded** inside the application. An example would look like this:

```
java.util.Properties props = new java.util.Properties();

props.put("sms.gsm.connector","SERIAL");
props.put("connector.serial.port","COM1");
props.put("connector.serial.bps","19200");

props.put("gsm.license.company","your_company");
props.put("gsm.license.user","your_name");
props.put("gsm.license.serial","your_serial");
props.put("gsm.license.version","2.x");
props.put("gsm.license.type","TRIAL LICENSE");
props.put("gsm.license.sig","your_key");


SmsService service = new GsmSmsService();
service.init(props);
```

5. Multimedia Message Service (MMS)

jSMS 2.x supports sending and receiving of Multimedia Messages (MMS). Currently, jSMS provides a implementation for the following protocol(s):

- **MM1** (WSP/WTP using WAP-PUSH and a PPP connection)
- **MM7** (Multimedia Messaging for Value Added Service Providers (VASP))

5.1. MMSService

The `MMSService` interface defines the methods to send and receive a Multimedia Message (MMS) to/from a MMS-Proxy/Relay.

Even if SMS and MMS look very similar for the user, they base on different technologies.

With **MM1**, the content of a MMS is not transported using the GSM SMS technology but over a PPP (Point-to-Point) connection. To send a MMS, a PPP connection is established and the MMS is submitted to the MMS-Proxy/Relay using a WAP POST request (UDP). For receiving a MMS, the MMS-Proxy/Relay sends a notification to the client using WAP-PUSH (WAP over SMS). This Notification contains the location where the MMS may be retrieved. The actual fetch of the incoming MMS is done with a WAP GET request (UDP, using a PPP connection over GPRS).

MM7 uses SOAP (Simple Object Access Protocol) for exchanging multimedia messages between a Value Added Service Provider (VASP) and the MM7 Proxy/Relay. SOAP messages are transferred over HTTP(S).

To construct a `MMSService` object, the `MMSServiceFactory` (see below) must be used.

5.2. MMSServiceFactory

Use this class to create `MMSService` objects. The factory depends on user-provided configuration properties for its operation. The factory offers two methods for acquiring an `MMSService` implementation: `getService(String name)` for acquiring a MMSService object by name and `getDefaultService()` which returns the default service. Please note that the factory creates single instances for each service, therefore calling `getService` multiple times for the same service name always returns the same object.

The three main properties that are required to create a MMSService are the Service name, the name of the Protocol and the name of the class implementing the Transport interface. Each Protocol and Transport-Implementation expects its own additional properties. Here are the properties used by the factory to create a MMSService:

Property	Description
[mms.default.service]	The name of the default service. If this property is set, the MMSServiceFactory will use the prefix "mms.[name]." when looking for the service configuration. If no default service is specified, the factory will use the prefix "mms."
[mms.protocol.name]	MMS Protocol to use: ● MM1 ● MM7
[mms.transport.class]	MMS Transport implementation to use (See the API docs of package com.objectxp.mms.transport for a list of supported transports)

5.3. MMSListener

The MMSListener interface is used by jSMS to pass incoming Messages, Notifications and Read/Delivery-Reports to your application. The interface declares four methods to process incoming messages and reports:

- handleIncomingMessage
- handleReadReport
- handleDeliveryReport
- handleNotification

In order to receive incoming messages and reports, your application must implement this interface and register the implementation with the MMSService by invoking the setListener method on the MMSService.

When a message or report has been successfully processed by your application, you should return MMSResponseStatus.SUCCESS. In case of a failure or if you want to reject an incoming message/report, return one of the predefined MMSResponseStatus objects or create a customized status. If your implementation throws a RuntimeException or returns a null value, this is treated like returning a MMSResponseStatus.SERVICE_ERROR.

Since the jSMS **MM1** implementation (MM1Service) relies on a SmsService for receiving MM1 notifications and reports, a connection between the SmsService and the MM1Service can be established. For this purpose, the MM1Service implements the MessageEventListener interface and therefore can be registered as a listener for events emitted by the SmsService. In case of an incoming MM1 notification or report, MM1Service intercepts the message event and forwards it to the registered MMSListener by invoking the corresponding handler method.

5.4. Configuring the MM1Service

The `MM1Service`uses a WSP/WTP connection to communicate with the MMS-Proxy/ Relay. WSP (Wireless Session Protocol) is comparable to the HTTP Protocol but adjusted for wireless connections. WTP (Wireless Transport Protocol) is the equivalent of the IP Protocol. The underlying transport layer is PPP (Point-to-Point-Protocol).

To receive a MMS, the MMS-Proxy/Relay sends a `MMSNotification`to the client encapsulated in a SMS (WAP-PUSH). The `MM1Service`fetches the MMS using the WSP/WTP connection (WAP-GET).

To send a MMS, the message is also transported over the WSP/WTP connection (WAP-POST).

5.4.1. Hardware

To send a MMS, any GPRS modem attached to your PC can be used. Receiving a MMS will **not** work with a GPRS mobile phone, since the phone intercepts the MMS notification (even if it is not able to handle MMS) - the notification is never passed to jSMS. Therefore you must use a dedicated GSM/GPRS device (e.g. a Siemens MC-35).

To increase the throughput of MMS messages, you can use two GSM/GPRS devices. One device is used to receive MMS notifications via WAP-PUSH (WAP encapsulated in SMS). The second device establishes a permanent PPP connection to the MMS-Proxy/Relay and is used to send and receive MMS.

5.4.2. Common configuration

The MMS settings must be provided by your MMS provider. Usually you can find the settings required for connecting to the MMS-Proxy/Relay at the website of your provider. The same settings that are used to configure MMS on your mobile phone can be used to configure jSMS.

Adjust the following properties in the configuration file *bin/jsms.conf*:

Property	Description
`[mms.protocol.mm1.mmsc.url]`	URL of the MMS-Relay-Server e.g.: `http://mymmsc/mms/`
`[mms.protocol.mm1.wapgateway]`	IP address (and port) of the WAP gateway. Format: `IP-address:`*`port`* If the port has not been specified, it defaults to `9201`.
`[mms.protocol.mm1.report.allowed]`	Indicates whether or not sending of delivery report is allowed. Default is `true`. Set it to `false`if you do not want to send a report to the sender after you fetched the MMS.

`[mms.transport.class]`	Define the Transport layer which should be used for mms.
	
	GPRS:
	`com.objectxp.mms.transport.PPPDialup`
	
	Direct TCP/IP connection:
	`com.objectxp.mms.transport.DirectConnection`

`[mms.transport.ppp.timeout]`	Number of seconds to wait until the GPRS connection is established. Defaults to `20` seconds.
`[mms.transport.ppp.os]`	Operating System. If this property is unset, the OS gets detected automatically. Set this property to `WIN` for a system which is using a RAS dialer. Set this property to `LINUX`, `UNIX` or `OSX` if your system uses a BSD PPP daemon.

If you plan to receive MMS, you have to configure the GSM section as well. **Please note** that the RAS- and BSD PPP dial-up mechanism does not unblock PIN-code protected SIM-Cards. To activate a PIN-protected SIM-Card, either open a `GsmSmsService`first and close it or add the PIN-Code of your SIM card to the Modem Initialization scripts (`AT+CPIN="`*`<yourPIN>`*`"`).

5.4.3. Windows installation and configuration

On Windows, jSMS uses RAS (Remote Access Services) to establish a PPP connection to the MMSC. The following steps must be carried out on your Windows system:

1. Copy the `jdunxp.dll`(For Windows >= XP) or `jdun2k.dll`(For Windows 2000) or `jdun98.dll`(For Windows 98 or ME) to your Library Path, e.g. to `C:\Windows\System32`.

2. Edit your Modem Settings:
   To establish a GPRS connection, the modem has to be initialized with an additional AT- command that sets the GPRS access point. Go to: Control Panel ▸ Phone and Modem Options ▸ Modems, select your modem and click on "Edit Properties ▸ Advanced"

   Add the AT-command below as extra initialization command:

   ```
   AT+CGDCONT=1,"IP","<accessPointName>"
   ```

   (replace `<accessPointName>` with the access point name of your bearer)

3. Modify the following properties in your jSMS configuration:

Property	Description
`[mms.transport.ppp.ras.user]`	The user name your bearer requires.
`[mms.transport.ppp.ras.password]`	The password used to login to your bearers MMS network.
`[mms.transport.ppp.ras.modem]`	Name of installed GPRS modem. This must **exactly match** the name shown in the List of modems in *Control Panel ▸Phone and Modem Options*
`[Name of the PPP Interface]`	Name of the PPP Interface.

To test your RAS configuration, run the class `mms.SendExample` located in the `examples` directory of the jSMS distribution.

5.4.4. BSD-PPP daemon configuration

jSMS supports Unix systems having the BSD PPP daemon installed. The API has been tested on GNU/Linux- and Solaris 10 (x86), but should also work on other Unixes providing a BSD PPP daemon.

Make sure that the user who is running jSMS is **authorized to run** the PPP daemon (pppd). Unless that user is root, this is usually achieved by adding the user to a certain group (e.g. `dialup` or `uucp`) and setting the SUID-bit on the `pppd` executable. Consult your System Administrator or the Documentation of your OS for more information about how to allow non-root users to establish PPP connections.

To establish a PPP-connection to your MMS-Provider, jSMS invokes the PPP-Daemon (pppd) with the arguments "`call` *`peer`*", where *`peer`* corresponds to the property `mms.transport.ppp.bsd.peerfile` of your `MMSServiceFactory` configuration. Please note that a peers-file must be present for each MMS-Provider.

Installing the peers-file(s)

jSMS comes with templates of a PPP peers-file (the templates are called `mms`) for both the GNU/Linux- and Solaris OS. These templates are located within the `peers/` directory of the jSMS distribution. For other Unixes, start with one of the templates and tweak it to match your PPPD syntax.

For each MMSC-Provider, place a copy of the peers-file template into the peers-directory of your system (usually `/etc/ppp/peers/`), name it after your MMS-Provider (e.g. `vodafone`) and edit the copy using your favorite editor. The template contains extensive comments about each setting that has to be adjusted. Settings that have to be changed include the serial-port configuration (name, baud rate, etc.), the name of your GPRS-Access point, and possibly a username/password combination (if required by the MMS-Provider).

Adding a route to the MMSC server

Usually, the MMSC cannot be directly reached over the PPP link since the IP address of the PPP interface is not in the same subnet as the MMSC server. This means that a temporary route to the MMSC must be added to the routing table on your system. After the PPP link is terminated, the OS automatically removes this entry. To automatically add this route whenever the PPP link is established, the appropriate `route` command must be added to a script that is invoked by the PPP daemon after the link has been established. Depending on your OS vendor, the route command can either be added to the script `/etc/ppp/ip-up` or put in a separate script and placed in the directory `/etc/ppp/ip-up.d/`. The jSMS distribution contains example `ip-up` scripts for GNU/Linux and Solaris that can be found in the `peers/` directory. Either copy the provided `ip-up` script to the appropriate location on your system or integrate the scripts content to your existing `ip-up` script.

Validating the PPP configuration

After you have set up the peers-file(s) and ip-up script, you should validate the PPP configuration by manually establishing a link to your MMSC-provider(s).

For each provider, invoke the command pppd call *<peer>* (where *<peer>* is probably the name of your MMSC provider) and verify that the connection can be established. Make sure that you invoke pppd **as the same user** that will be running the jSMS-enabled application.

Below is a transcript of successfully establishing a PPP connection on Solaris 10. After the connection has been established (local- and remote addresses displayed), hit CTRL-C to disconnect. Please note that the pppd binary is usually placed in /usr/sbin/ (GNU/Linux) or /usr/bin/ (Solaris).

```
$ /usr/bin/pppd call mymmsc
Serial connection established.
Using interface sppp0
Connect: sppp0 <--> /dev/term/a
local IP address 10.111.10.131
remote IP address 192.168.254.254
^C
Terminating on signal 2.
Connection terminated.
Connect time 0.4 minutes.
Sent 620 bytes (16 packets), received 459 bytes (13 packets).
Serial link disconnected.
$
```

If you run into problems bringing up the PPP link, uncomment the debug directive in the peers-file and possibly also add the option "-V" to the chat script embedded in the peer file. This will give you a lot of debugging hints on *stderr*.

5.5. Configuring the MM7 protocol

MM7 is the protocol between the Multimedia Service Center (MMSC) and a Value Added Service Provider (VASP). It is based on SOAP and uses HTTP(S) as the transport protocol. It can be used by third-party applications to send and receive multimedia messages to/from MMS-capable mobile devices.

Please notice that MM7 requires Java 1.4.2 or later. The MM7 implementation depends on the JavaMail API (V1.4). You will also need the JavaBeans Activation Framework (JAF - V1.1). The jSMS distribution already contains the JavaMail (`lib/javamail.jar`) and JAF (`lib/activation.jar`) libraries. Those libraries must be added to your `CLASSPATH`.

5.5.1. Information required from your MMSC provider

MM7 requires an MM7 account at a MMSC provider. For **outgoing** multimedia messages, your MMSC provider must supply you with the following information:

- URL where the MM7 MMSC listens for message submission requests from your application
- If applicable, a user name and password required for submitting messages to the *URL* above
- If the MMSC operator uses *SSL*, you may require SSL specific information, e.g.:
  - a X509 CA Certificate used for verifying the SSL server
  - a X509 Client Certificate and private key used for authenticating your application at the SSL server
- a VASP ID

5.5.2. Information provided to your MMSC provider

For **incoming** multimedia messages, you must provide the following information to your MMSC provider:

- URL where your application is listening for incoming multimedia messages and reports
- If applicable, a user name and password that the operator must use when submitting requests to the *URL* above
- If your application uses SSL, you may also provide the operator with SSL specific information, e.g.:
  - the X509 CA Certificate which signed your SSL server certificate
  - a X509 Client Certificate and private key that the operator must use for authentication at the *URL* above

5.5.3. Operation Modes

The MM7 protocol implementation provided by jSMS supports two modes of operation: **Standalone** and **Web-Application**. In both modes, jSMS uses `java.net.URL` for outgoing connections.

Standalone Operation

When using jSMS in standalone-mode, an embedded web server is used for reception of MM7 requests from the MMSC operator. jSMS doesn't provide its own web server implementation but uses *Jetty* (*http://www.mortbay.org*) instead. Jetty is **not** bundled with jSMS, therefore you must download **Jetty 6.x** yourself. Add `lib/servlet.jar` and the Jetty JAR files (`jetty-6.x.x.jar`, `jetty-util-6.x.x.jar`) to your `CLASSPATH`.

When configuring the embedded web server, various configuration properties are used to form the final `URL` where the MM7 MMSC can submit messages/report to:

```
(http)://[(user):(pass)@]<hostname>[(:port)][(path)]
```

(http)	`mms.protocol.mm7.in.ssl`
(user)	`mms.protocol.mm7.in.username`
(pass)	`mms.protocol.mm7.in.password`
(:port)	`mms.protocol.mm7.listen`
(path)	`mms.protocol.mm7.in.path`

Assuming your machine is named `myhost.mydomain` and you have set `ssl` to `false`, `listen` to `8080` and `path` to `/in/`, the resulting URL would be **`http://myhost.mydomain:8080/in/`**. This is the URL that you would have to provide to your MM7 MMSC operator. Since no username/password is set, accessing the URL doesn't require *HTTP Basic Authentication*. See *MM7 Configuration Properties* for a list of supported properties.

Web-Application Mode

If you use jSMS within a Java Web Application, the *Web Application Server* (e.g. Tomcat, Jboss) will handle incoming HTTP(s) requests from your MM7 MMSC.

jSMS provides a `Servlet` for processing incoming requests from the MMSC: `MM7ReceiverServlet`. This class is abstract, so the developer must subclass it and implement at least the `getService` method, returning a `MMSService` instance. It is up to the developer on how to construct this `MMSService` instance.

5.5.4. MM7 Configuration Properties

The following configuration properties apply to MM7:

Property	Description
[mms.protocol.name]	Protocol handler. For MM7, this must be set to MM7
[mms.protocol.mm7.vasp.id]	MM7 VASP ID
[mms.protocol.mm7.vas.id]	Value Added Service ID (VASID). **Optional.**
[mms.protocol.mm7.out.url]	URL where the MM7 MMSC listens for message submission requests from your application
[mms.protocol.mm7.out.username]	User name required for submitting messages to the MM7 MMSC. **Optional.**
[mms.protocol.mm7.out.password]	Password required for submitting messages to the MM7 MMSC. **Optional.**

Additional Properties for Standalone Mode

[mms.protocol.mm7.listen]	Set this property to a port or IP-address/port combination if you want to enable the embedded HTTP(S) server for MMS reception. When specifying an IP address, the HTTP(S) server will only listen on the given address. If only a port is set, the server will listen on all network interfaces. **Notice:** you must add Jetty (http://jetty.mortbay.org/) to your CLASSPATH to enable standalone mode.
[mms.protocol.mm7.in.path]	This is the path component of the URL for incoming messages/reports from the MM7 MMSC. If unset, the path defaults to /. **Optional**.
[mms.protocol.mm7.in.ssl]	Turn on HTTPS (SSL) for the embedded HTTP(S) server. **Optional.**
[mms.protocol.mm7.in.ssl.clientauth]	Require certificate based client authentication from the MM7 MMSC. SSL must be enabled. **Optional.**
[mms.protocol.mm7.in.username]	User name hat the MM7 operator must use when delivering messages to your application. **Optional.**
[mms.protocol.mm7.in.password]	Password hat the MM7 operator must use when delivering messages to your application. **Optional.**

5.5.5. SSL/TLS

jSMS can use the SSL/TLS protocol for securing the communication with the MM7 MMSC. The API relies on the Java Secure Socket Extension (JSSE) which is part of Java version 1.4.2 and later. JSSE includes support for SSL/TLS and contains a protocol handler for dealing with HTTPS URLs.

- For sending MM7 messages using HTTPS, jSMS uses the class `java.net.URL`.
- For receiving MM7 messages and reports in Standalone Mode, jSMS creates a Server Socket using `javax.net.ssl.SSLServerSocketFactory`
- When using jSMS with a web application, the Web Application Server is responsible for handling SSL-Requests. For more information about configuring SSL for your web application, consult the documentation of your Web Application Server.

JSSE Configuration

Key material and X509 Certificates that JSSE uses for SSL/TLS are stored in keystores. Information in a keystore can be grouped into two categories: key entries and trusted certificate entries. A key entry consists of a X509 certificate and its private key, and can be used e.g. for running a SSL/TSL server or for authenticating a client against an HTTPS server. A trusted certificate entry can be used for verifying the identity of a communication partner. If a keystore only contains trusted certificate entries, it is called a *truststore*.

If you require a key-entry (for authenticating against the MM7 HTTPS server or for receiving messages and reports over HTTPS in standalone mode), you must create your own keystore.

Java includes a tool called `keytool`that can be used for creating and managing keystores. The location of your keystore(s) must be specified using the Java system properties listed below:

System Property	**Description**
`javax.net.ssl.keyStore`	Location of the keystore
`javax.net.ssl.trustStore`	Location of the truststore. If unset, JSSE will look for $JAVA_HOME/jre/lib/jssecacerts and $JAVA_HOME/jre/lib/cacerts
`javax.net.ssl.keyStorePassword`	Keystore password
`javax.net.ssl.trustStorePassword`	Truststore password

System Properties can either be passed to Java using the option `-D` on the command line (e.g. `java -Djavax.net.ssl.keyStore=$HOME/mykeystore`) or can be set at runtime by calling `System.setProperty(...)`

For more information about customizing JSSE, see this document:

http://java.sun.com/j2se/1.4.2/docs/guide/security/jsse/JSSERefGuide.html

6. Messages

6.1. Message and SmsMessage

The base class for all messages (except MMS) in the jSMS API is called `Message`. A `Message` object can be sent as an SMS (by using one of the `SmsService` implementations) or as an email (using the `SmtpService`). If you want to have more control about the content and attributes of a SMS Message, the class `SmsMessage` should be used. `SmsMessage` extends the base `Message` class and adds additional features like specifying the message's character encoding (alphabet), setting the validity period of the message, requesting a status report, etc.

6.1.1. Sending a SMS Message

After a `SmsService` object has been created and initialized, you can send Short Messages using either the basic `com.objectxp.msg.Message` object or an instance of `com.objectxp.SmsMessage` (for fine grained control over the message):

```
Message msg = new Message();
msg.setRecipient("+41123456789");
msg.setMessage("Hello Mobile World");
service.connect();
service.sendMessage(msg);
```

6.1.2. Receiving a SMS Message

SMS messages can also be received using the SmsService object. To prepare your application for receiving short messages, the following steps have to be done:

1. Implement the interface MessageEventListener. All registered event listeners will be notified by the involved SmsService about incoming messages, outgoing messages, established connections, etc. Consult the jSMS API documentation for more information about the MessageEventListener interface.
2. Register your MessageEventListener implementation with the SmsService by invoking its addMessageEventListener method
3. Instruct the SmsService to start receiving messages by calling connect followed by startReceiving

The following code illustrates the basic steps to implement reception of SMS messages in a jSMS application.

```
import java.io.File;
import com.objectxp.msg.*;

public class MyEventListener implements MessageEventListener
{
    public void handleMessageEvent(MessageEvent event)
    {
        if(event.getType() == MessageEvent.MESSAGE_RECEIVED){
            System.out.println("Received message is:" + event.getMessage());
        }
    }

    public static void main(String args[])
        throws Exception
    {
        File config = new File(”/path/to/your/jsms.conf”);
        SmsService service = new GsmSmsService();
        service.init(config);
        service.addMessageEventListener(new MyEventListener());
        try{
            service.connect();
            service.startReceiving();
            System.out.print("Press any key to stop receiving messages");
            System.in.read();
            service.stopReceiving();
            service.disconnect();
        } finally{
            service.destroy();
        }
    }
}
```

6.2. MultipartMessage

Normally the content of one message is limited to 160 characters (or 140 octets in binary mode). The `MultiPartMessage` interface gives the user the ability to send content longer than the above limits. Two implementations of the `MultiPartMessage` interface are available: `SmartMessage` and `EMSMessage`.

6.2.1. SmartMessage

jSMS includes classes for sending and receiving Nokia Smart Messages. The following Smart Messages are supported

- Operator logo
- Picture message
- Calling line identification icon (CLI Icon)
- Ring tone
- VCard (Business card)
- VCalendar

6.2.2. EMSMessage

jSMS supports sending and receiving EMS messages. With EMS you can send more then just simple text messages. The EMS package in jSMS gives you the ability to send and receive formated text, pictures, animations and sounds. Several content elements can be inserted in one message. jSMS supports the following EMS elements:

- Animations
- Text and formated text
- Sounds (predefined and user defined)
- Pictures (predefined and user defined)
- User prompt indicator

An `EMSMessage` can also be used for sending and receiving user defined content such as text longer than 160 characters or data longer than 140 octets.

6.2.3. Sending a multipart message

Sending a EMS or Smart message is as easy as sending a text message. You do not have to care about the message length and how to split it. Just create the message and send it.

```
EMSMessage msg = new EMSMessage();
//add content
msg.add(new EMSText("Hello",EMSTextFormat.BOLD));
msg.add(EMSAnimation.WOW);
msg.add(new EMSText("World",EMSTextFormat.LARGE));
//set recipient
msg.setRecipient("+41791234567");
//send it
service.sendMessage(msg);
```

6.2.4. Receiving a multipart message

Depending on the content length of a SmartMessage or EMSMessage, jSMS splits such messages in multiple fragments before sending them over the wire. jSMS also provides a class that will automatically reassemble such fragments into a single message on reception. To use this feature, add an instance of MultiPartReceiver to your service as a MessageEventListener. Your own MessageEventListener implementation can then be added to this MultiPartReceiver. All events, except for MessageEvent.MESSAGE_RECEIVED, will be dispatched directly to the registered listeners of the MultiPartReceiver. In case of a MESSAGE_RECEIVED event, the MultiPartReceiver checks if the message received is a multipart message or a single-part message. Single-part messages will be dispatched directly to underlying listeners without further processing. If the received message is part of a concatenated SMS, the message is placed into an internal memory cache. After all parts of a message have arrived, those parts are reassembled into one single MultiPartMessage (EMS - or SmartMessage).

```
//Create the SmsService (Replace GsmSmsService with the SmsService
//Implementation of your choice).
SmsService service = new GsmSmsService();

//new event listener
MessageEventListener listener = new MyMessageEventListener();

//Construct a multipart receiver
MultiPartReceiver receiver = new MultiPartReceiver(TIMEOUT, MAX_ENTRIES, listener);

try{
    //get the service ready to listen for incoming messages
    service.init(config);
    //add the multipart receiver as listener
    service.addMessageEventListener(receiver);
    service.connect();
    service.startReceiving();

    //manualy interupt the receiver
    System.out.print("Press any key to stop receiving messages");
    System.in.read();

    //stop listening for incoming sms's
    service.stopReceiving();
    service.disconnect();
} finally{
    service.destroy();
    receiver.destroy();
}
```

6.3. OTA Messages

The Messages in the OTA package (Over The Air) provide the means for handling WAP-Push messages. The SMS payload (user data) for OTA messages holds a binary encoded XML document (WBXML) with a specific MIME type depending on the type of OTA message.

6.3.1. BrowserSetting

To be able to access services such as WAP, GPRS, MMS or email, a mobile phone must be configured accordingly. Instead of manually entering those settings on the mobile phone, the configuration can also be “pushed” to the mobile phone using the appropriate OTA “Browser Setting” message. Depending on the subsystem to configure, one of the following classes can be used:

- `GPRSBrowserSetting`
- `GsmCsdBrowerSetting`
- `GsmSmsBrowerSetting`
- `GsmUssdBrowerSetting`
- `Is126CsdBrowerSetting`
- `CSDBrowserSettings`

6.3.2. Bookmark

This class may be used to send OTA Browser bookmarks using WAP PUSH over SMS. Browser bookmarks are used to provide handsets with bookmarks of any kind that can be used for browsing. A `Bookmark` consists of a Name and a URL.

6.3.3. Service Indication / Service Loading

The `ServiceIndication` class represents a OTA Service Indication (SI). The SI provides a way to inform a user that an event has occurred and indicate a URL that can be loaded in order to react to that event. This is done by sending a SMS to the client that informs the recipient about the event, and a URL from where the appropriate service can be loaded. For example, the message could state that "A new mail has arrived", including the URL of the Web-Email interface.

The `ServiceLoading` class represents the OTA Service Loading content type (SL). A SL message causes a WAP browser on a mobile phone to load and execute a URL. If appropriate, the mobile phone loads this URL without any user intervention.

6.3.4. Sending an OTA Message

Note: To use the OTA package, a SAX parser implementing the Java API for XML Processing (JAXP) must be accessible:

- Starting with Java Version 1.4, an implementation of the JAXP-API is already included in the JRE.
- For previous Java versions, you must add a SAX parser and the JAXP API classes to your `CLASSPATH`. We recommend using the Apache Xerces XML Parser which is freely available at http://xml.apache.org/. Add both `xercesImpl.jar` and `xml-apis.jar` to your application `CLASSPATH`.

```
//Create and initialize the SmsService (Replace GsmSmsService with
//the SmsService Implementation of your choice).
SmsService service = new GsmSmsService();
service.init(config);

//Create Bookmark
Bookmark msg = new Bookmark("objectXP","www.objectxp.com");
msg.setRecipient(receiver);
msg.setSender(sender);

//connect, send message and disconnect
try{
    service.connect();
    service.sendMessage(msg);
    service.disconnect();
} finally{
    service.destroy();
}
```

6.3.5. Receiving OTA Messages

Although OTA messages can be received with jSMS, they will not be automatically converted to the appropriate OTA object. This means that incoming OTA messages will be delivered as regular `SmsMessage` objects. Parsing of those messages is therefore left to the application.

6.4. Multimedia Messages

6.4.1. MMSNotification

The `MMSNotification` extends the `SmsMessage` class. It also implements the `MultiPartMessage` interface. A `MMSNotification` is sent from the **MM1** MMSC to the recipient of a Multimedia Message to notify the recipient that a MMS is ready to be retrieved. The Notification is encapsulated in a SMS message (WAP PUSH). After reception of a `MMSNotification`, the application must establish a connection to the MMSC using the appropriate `MMSService` object and retrieve the MMS using the `fetch` method offered by `MMSService`.

6.4.2. MMSMessage

A `MMSMessage` is a container holding one or multiple multimedia parts, such as text, image, sound and video. It doesn't contain a definition on how the target device should display the media parts. A possible application for this class is to send a Ring tone to a MMS-capable mobile phone.

6.4.3. SMILMessage

The class `SMILMessage` is also a container holding multimedia parts but additionally contains a SMIL (Synchronized Multimedia Integration Language) document that describes how the multimedia-parts should be displayed on the target device. The SMIL document contains references to the media parts and defines the chronological and graphical order of those parts.

6.4.4. Sending a SMIL Message

Sending a SMIL Message is nearly as simple as sending a common SMS. Instead of adding some text as Message content, the location of a SMIL document is specified.

```
//Create a MMS Service factory
File config = new File("/path/to/jsms.conf");
MMSServiceFactory factory = MMSServiceFactory.createFactory(config);
MMSService mmsService = factory.getDefaultService();

//load a SMIL document from the given URL
URL url = new URL("file:///path/to/your/smil.xml");
MMSMessage msg = new SMILMessage(url);

msg.addTO(new MMSAddress(MMSAddress.TYPE_PLMN, "555123456"));

//send the MMS
try{
    mmsService.connect();
    mmsService.send(msg);
    System.out.println("Message sent succesfully.");
}catch(Exception e){
    System.err.println("could not send Message:"+e);
    e.printStackTrace();
} finally{
    mmsService.disconnect();
}
```

6.4.5. Receiving a multimedia message over MM1

When a **MM1** MMSC has to deliver a MMS, it first sends a notification to the recipient to indicate that a Multimedia Message is ready to be retrieved. This notification is sent using WAP-PUSH (SMS). At a later stage, the MMS can be retrieved by passing the `MMSNotification` to the `fetch` method of the `MMSService`.

To receive MMS notifications, a `SmsService` must be used. Since MMS notifications might not fit into a single SMS, the `MultiPartReceiver` class must be used for reception. See *6.2.4 - Receiving a multipart message* on how to receive multipart messages. The example below only shows how to fetch a MMS message after having received a MMS notification.

```
public void handleMessageEvent(MessageEvent event)
{
    if(event.getType()!=MessageEvent.MESSAGE_RECEIVED ){
        //We are only interested in incoming messages
        return;
    }

    Message msg = event.getMessage();

    if(msg !=null && (msg instanceof MMSNotification)){
        MMSNotification notification = (MMSNotification)msg;
        //Disconnect the SMS Service
        if(service!=null){
            try{
                service.stopReceiving();
                service.disconnect();
            }catch (MessageException e) {
                System.err.println("disconnection SmsService failed.");
                e.printStackTrace();
            }finally{
                service.destroy();
            }
        }

        MMSService mmsService=null;
        try{
            //Get the default MMS Service (The MMSServiceFactory has been created
            //at an earlier stage)
            mmsService= factory.getDefaultService();
            //Connect to the MMSC
            mmsService.connect();
            //Fetch the MMS
            MMSMessage mms = mmsService.fetch(notification);
            //Show incoming message
            System.out.println("MMS Fetched:\n"+mms.toString());
        }catch (Exception e) {
            e.printStackTrace();
        }finally{
            //Disconnect from MMSC
            if(mmsService!=null){
            try{
                mmsService.disconnect();
            }catch (Exception e) {
                System.err.println("Could not disconnect mmsService");
                e.printStackTrace();
            }
        }
    }

    //restart the GSM Service and listen for more incoming messages
    try{
        startGsmService();
    }catch (Exception e) {
        e.printStackTrace();
    }
}
```

6.4.6. Receiving a multimedia message over MM7

Incoming messages and reports are delivered to your application through the MMSListener interface. Your application must implement this interface and register the implementation with the MMSService (by calling the setListener method)

Reception in Standalone Mode

If you use jSMS in Standalone Mode, you must configure the port (and possibly network interface) where jSMS listens for incoming messages and reports from the MM7 MMSC.

As soon as you call connect on the MMSService, jSMS will start to listen for incoming HTTP(S) requests. Incoming requests are transformed into the corresponding Java object (e.g. MMSMessage, MMSReadReport) and then passed to the MMSListener registered with the MMSService. If no listener is registered, jSMS refuses the request from the MM7 SMSC by answering with HTTP status code 500.

Reception with a web application

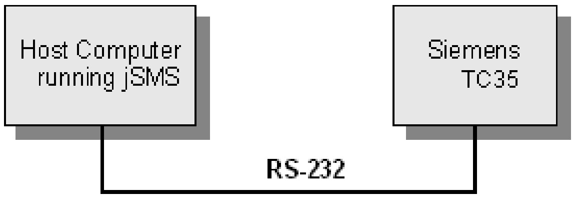
When using jSMS in a web application, the Web Application Server will listen for incoming HTTP(S) requests from your MM7 provider and will forward such requests to your application. In order to receive MMS messages and reports, your application must extend the abstract class MM7ReceiverServlet and provide at least the method getService - returning an instance of a MMSService. The Web Application Server must then be instructed to forward all requests to a certain Path (e.g. /VASP/*) to this servlet. The MM7ReceiverServlet provides a default implementation of the doPost method, parsing the incoming MM7 request and dispatching incoming messages and reports to a MMSListener. The MMSListener must be registered with the MMSService returned by getService. If the request to process is not a MM7 request, the MM7ReceiverServlet will throw a ServletException. To prevent this, MM7ReceiverServlet provides the method isMM7Request that can be used to determine if an incoming request is from a MM7 MMSC. To use this method, you may override the doPost method, call isMM7Request and only pass processing to MM7ReceiverServlet in case of a MM7 request:

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.objectxp.mms.MMSService;
import com.objectxp.mms.protocol.MM7ReceiverServlet;

public class MyMM7Servlet extends MM7ReceiverServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        if(!isMM7Request(request)) {
            // Handle non-MM7 requests here:
        } else {
            // Pass the request to jSMS:
            super.doPost(request, response);
        }
    }

    public MMSService getService() {
        ...
        return service;
    }
}
```

7. Sample jSMS Applications

The directory *examples* within the jSMS distribution contains various examples showing the different aspects of using the jSMS API.

7.1. Send SMS messages using a GSM Device

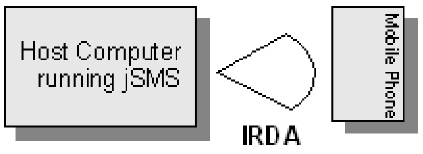


The GSM Device (e.g. your Mobile Phone) is connected **directly to a serial port** on the host computer. The jSMS API will communicate with the device using a **javax.comm** compliant library.

The following example shows how to apply jSMS for this application:

```
package example1;

import java.io.File;
import com.objectxp.msg.*;

public class JSMSExample1
{
    public static void main(String[] args)
    {
        SmsService service = null;
        try{
            // Configuration
            File config = new File(”/path/to/your/jsms.conf”);

            // create service object.
            service = new GsmSmsService();
            service.init(config);
            // create a new Message.
            Message msg = new Message();
            msg.setRecipient(“+411234567”);
            msg.setMessage("jSMS is cool!");
            // Connect to the device
            service.connect();

            // send the Message
            service.sendMessage(msg);
            System.out.println("Message sent successfully ID is "+msg.getMessageId());
        } catch (Exception ex) {
            System.err.println("Message could not be sent: "+ex.getMessage());
            ex.printStackTrace();
        } finally{
            if(service!=null){
                try{
                    service.disconnect();
                }catch(Exception unknown ) {}
                service.destroy();
            }
        }
    }
}
```

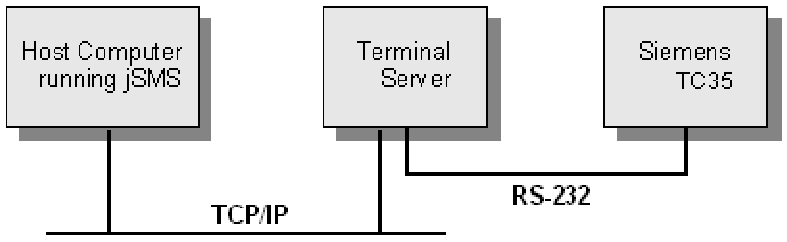
7.2. Receive SMS messages using a GSM device via IrDA



In this configuration, a IrCOMM capable GSM mobile phone will be used for sending and receiving Short Messages (SMS). As in the previous example, the jSMS API communicates with the GSM device using a javax.comm compliant library. **Notice**: the host OS must provide IrCOMM capabilities. The following example shows how to apply jSMS for this application:

```
package example2;

import java.io.File;
import com.objectxp.msg.*;

public class JSMSExample2 implements MessageEventListener
{
    //EventHandler
    public void handleMessageEvent(MessageEvent event){
        if(event.getType() == MessageEvent.MESSAGE_RECEIVED) {
            Message msg = event.getMessage();
            if(msg != null)
                System.out.println("SMS received:" + msg.getMessage());
        }
    }

    //main
    public static void main(String[] args)
    {
        SmsService service = null;
        try{
            //set the jSMS properites.
            File config = new File(”/path/to/your/jsms.conf”);

            //create service object.
            service = new GsmSmsService();
            service.init(config);

            //add listener to receive message.
            service.addMessageEventListener(new JSMSExample2());

            //Connect to the device and start Receiving messages
            service.connect();
            service.startReceiving();

            System.out.println("waiting for messages...");
            while(true){
                try{Thread.currentThread().sleep(1000);} catch (Exception e) {}
            }
        } catch (Exception e) {
            System.err.println("Could not receive messages:"+e.getMessage());
            e.printStackTrace();
        } finally{
            if(service!=null){
                try{
                    service.disconnect();
                }catch(Exception unknown ) {}
                service.destroy();
            }
        }
    }
}
```

7.3. Send SMS messages through a Terminal Server

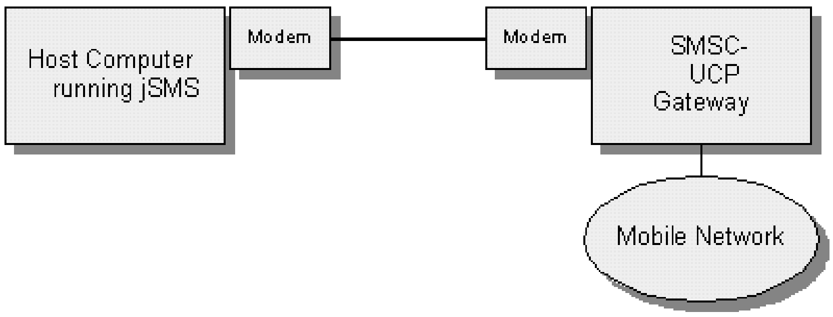


If the GSM device (for example a Siemens TC35) can not be connected directly to the host computer due to the lack of a radio signal, jSMS can communicate with the device by connecting to a terminal server reachable through TCP/IP. The terminal server forwards data from/to the host computer to the GSM device attached to (one of) the serial ports of the terminal server.
Since jSMS is designed with a unified interface covering all underlying transport facilities, you may use exactly the same java code as shown in example 5.1 above.

The only change needed, is to adapt your jSMS configuration properties to the above configuration. In your jSMS configuration file, change the property `sms.gsm.connector` from `SERIAL` to **`TCP`** and specify the host name and port of the Terminal Server:

```
sms.gsm.connector=TCP
connector.tcp.host=hostname/IPaddressofyourterminalserver
connector.tcp.port=portnumber
```

7.4. Send Short Messages (SMS) using UCP

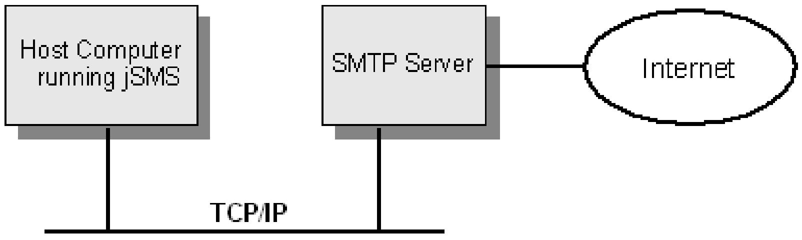


Most mobile network operators run a UCP (Universal Computer Protocol) gateway. This gateway usually can be reached over a modem / ISDN connection or using TCP/IP . In order to use UCP, you have to know the phone number and connection settings (parity, baud rate, etc.) of the UCP gateway (for Modem Connections). Using UCP over TCP/IP requires a contract (“large account”) with your mobile network provider. For more information about accessing a UCP gateway, contact your local network operator. For the below example to work, configure jSMS by placing the appropriate properties in your jSMS configuration file:

```
ucp.connector=SERIAL
ucp.smsc.number=phone number of the UCP Gateway
ucp.port.name=COM1
ucp.port.bps=9600
```

```
package example4;

import java.io.File;
import com.objectxp.msg.*;

public class JSMSExample4
{
    public static void main(String[] args)
    {
        SmsService service = null;
        try{
            // Configuration file
            File config = new File("/path/to/your/sms.conf");

            // create and initialize UCP service
            service = new UcpSmsService();
            service.init(config);

            // create message object
            Message msg = new Message();
            msg.setSender("<sender address>");
            msg.setRecipient("<recipient phone number>");
            msg.setMessage("jSMS over UCP is cool!");

            // connect to SMSC and send the message
            service.connect();
            service.sendMessage(msg);
            System.out.println("Message sent successfully ID is "+msg.getMessageId());
        } catch (Exception e) {
            System.err.println("Message could not be sent: "+e.getMessage());
            e.printStackTrace();
        } finally{
            if(service!=null){
                try{
                    service.disconnect();
                } catch(Exception unknown ) {}
                service.destroy();
            }
        }
    }
}
```

7.5. Send Internet mail messages using the SmtpService



The jSMS API contains a small footprint SMTP client capable of sending Internet mail messages (RFC822). The jSMS API has been designed in a way that the same message can either be sent through SMS or SMTP. The SmtpService uses a mail server reachable by TCP/IP to deliver Internet mail messages.

7.5.1. The SmtpService Class

This class may be used to send Internet mail messages (RFC822) using a SMTP mail server.

For the below example to work, configure jSMS by placing the appropriate properties in your jSMS configuration file:

```
smtp.host=your.smtp.host
smtp.sender=yourname@yourDomain.com
```

```
import com.objectxp.msg.*;
import java.io.*;
import java.util.*;

public class SendMail
{
    public static void main(String args[]) throws Exception
    {
        File file = new File(”/path/to/your/sms.conf”);
        SmtpService service = new SmtpService(file);

        SmtpMessage msg = new SmtpMessage();
        msg.addRecipient("user1@domain1.com");
        msg.addRecipient("user2@domain2.com",Recipient.RT_CC);
        msg.addRecipient("mySelf@myDomain.com",Recipient.RT_BCC);
        msg.setSubject("Business lunch");
        msg.setMessage("Business lunch today at the Restaurant 'Chez Max'");

        try {
            service.sendMessage(msg);
            System.out.println("Message sent successfully ID is "+
            msg.getMessageId());
        } catch(MessageException me ) {
            System.err.println("Message could not be sent:"+me.getMessage());
        }
    }
}
```

8. Debugging your Application

jSMS uses a logging API based on **Log4j** (http://jakarta.apache.org/log4j/) to log its execution. Logging can be enabled by adding Log4j to the `CLASSPATH` and specifying the name of the log file using the Java System Property `jsms.logfile`.

If you experience problems while accessing your GSM device or SMSC, rerun your application with jSMS logging enabled.

Follow these steps to produce a jSMS log file:

1. Download Log4j at http://jakarta.apache.org/log4j/
2. Add log4j.jar to your class path
3. Specify the name of the jSMS log file using the System property "jsms.logfile"

Example:

```
c:\> java -cp lib\jSMS.jar;lib\log4j.jar;. -Djsms.logfile=jsms.log MyApplication
```

If you already use Log4j for your application, there is no need for setting the `jsms.logfile` system property: Just modify your existing Log4j configuration to include the category `com.objectxp.msg`.

9. Frequently Asked Questions (FAQ)

Q: I've just downloaded jSMS. What other hardware/software do I need to send/receive SMS?

A: You need a Java JDK/JRE (at least Version 1.2x) and either a GSM device or a Modem/ISDN-Adapter.

Q: Your documentation mentions a "Siemens TC35". Does this mean that I need a TC35 to use your software?

A: No, jSMS should work with any GSM device with a built-in modem that is capable of sending/receiving Short Messages (SMS) and can be accessed using a serial port.

Q: I try to use jSMS with my Falcom/Wavecom GSM device. After calling `init` on the `GsmSmsService`, I get an exception saying there is no response from the device. What can I do against it?

A: Try to increase the initialization timeout in your jSMS properties to a higher value (e.g. sms.gsm.inittime=30)

Q: Does jSMS support J2ME (Mobile Edition)

A: No, jSMS requires a lot of classes which J2ME does not (yet) provide.

Q: Will jSMS work with my Nokia 5510 mobile phone?

A: Yes, but since older Nokia phones (like the 5510) don't have a built-in modem, you have to use Nokia's DataSuite which provides a “virtual” com port.

Q: I get an error when running jSMS saying that my serial port 'COM1' doesn't exist. What is wrong?

A: Make sure that you have correctly installed `javax.comm`. Follow the instructions for installing javax.comm given in the document “`install_commapi.html`”

Q: Does jSMS support the EMI UCP protocol?

A: Yes, jSMS complies to version 4.0 of the EMI UCP Specification. Supported operations are: 60 - Open Session (Subtype 1), 01,30,52 - Sending Messages 01,52 - Receiving messages, 53 - Receiving status reports. Notice: You need a special license key to enable this functionality. Contact us if you'd like to evaluate UCP support.

Q: Does your Java API use JNI to interface to a C library, or is the whole protocol stack written in Java?

A: The common SMS functionality of jSMS doesn't contain any native code. But to use the Multimedia functionality on a Windows OS, jSMS relies on a native C library.

Q: I have already purchased a developer license, how can I purchase a runtime license?

A: Runtime licenses can be ordered at our jSMS support page:
https://www.objectxp.com/merchant/support.do
To access this page, you have to enter the login and password you received when purchasing the development license.

Q: When sending messages using a GSM phone, does jSMS make use of PDU mode? Is there support for Text mode?

A: jSMS uses PDU mode for sending messages. We have dropped text mode a long time ago, since not all GSM devices support it and you can't set some message properties in text mode.

Q: May I send Multimedia Messages with my Nokia 6310?

A: Yes, sending a Multimedia Message is possible with any GPRS device, even if the Mobile device is not able to handle Multimedia Messages.

Q: May I receive Multimedia Messages with my Nokia 6310?

A: No, since the Mobile Device catches the `MMSNotification` even if it can not handle it. Therefore jSMS doesn't receive notifications about incoming Multimedia Messages. Use a dedicated GPRS device instead (e.g. a Siemens MC35).