Subject:	**Integrators guide**	*Proj. nr.:*	**DCP2635-00**	*Page:*	**1 of 21**
Section:	**Smart AMM API**	*Date:*	**2008.07.11**	*Version:*	**1.1.0**
Reference:		*Refer to:*	**Morten Krogh Andersen**	*Replaces*	**1.0.1**
File name:	**SmartAMM API Integrators guide.doc**	*Appr.:*			

Integrators guide

Revision history

Version	Date	Initials	Description
0.0.1	2008.04.28	MKA	Initial draft.
0.0.5	2008.05.06	MKA	Released for internal review.
1.0.0	2008.06.23	MKA	Added description of IP modem configuration. Added description of what setting should be revised after installation.
1.0.1	2008.07.08	MKA	Updated SMI naming
1.1.0	2009.01.29	MKA	Removed all SMI naming. Updated code examples.

Introduction

Purpose	This document serves as a guide for deployment and systems integration specialists who will be configuring and developing for the SmartAMM system
Application	SmartAMM Communication Service Provider and API
Refer to	Morten Krogh Andersen
Remarks	

Contents

1 Introduction....3
1.1 Requirements....3
1.2 SmartAMM Communication Service Provider (CSP)....4
1.3 SmartAMM API....4
1.4 Usage scenarios....4
1.4.1 Embedded message broker....4
1.4.2 External message broker....4
1.4.3 Using the API....5
2 CSP Deployment....6
2.1 Configuration....6
2.1.1 Service/Daemon configuration....6
2.1.2 Initial configuration....6
2.2 Monitoring....7
2.2.1 JMX interface....7
3 SMI8
3.1 Identification of remote hosts and message delivery....9
3.2 Transport status....10

4 Appendices....11
4.1 CSP Configuration....11
4.1.1 Sections....11
4.1.2 Available component implementations....12
4.2 Client examples....16
4.2.1 Client setup....16
4.2.2 Client connect....18
4.2.3 Client subscription....18
4.2.4 Sending a message....19
4.2.5 Class implementing all event listener interfaces....20
4.3 Third party software....21
4.3.1 ActiveMQ....21
4.3.2 MINA....21
4.3.3 Smslib....21
4.3.4 Java Service Wrapper....21
4.3.5 Nullsoft Scriptable Install System (NSIS)....21

List of figures

1-1: System overview....3
1-2: Embedded broker....4
1-3: External message broker....5
1-4: Using the MI Client....5
3-5 : Example connect and usage of the remote host id....9

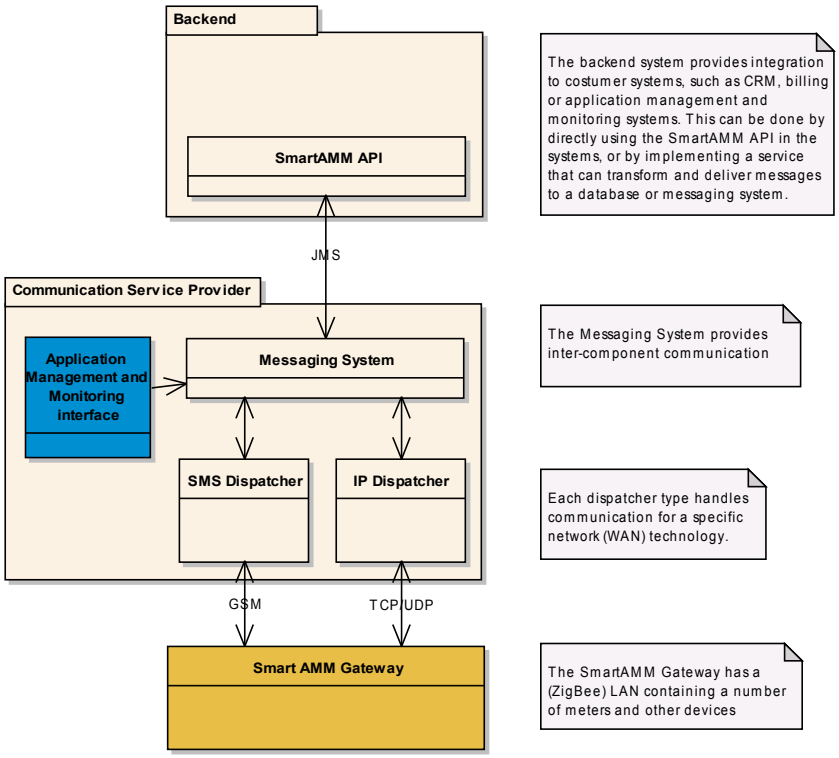
1 Introduction

The SmartAMM Communication System consists of two core components: the *Communication Service Provider (CSP)* and the *SmartAMM API (API)*.

The CSP is handling the communication with the remote SmartAMM gateways, and the API library is used by both the CSP and optionally systems wishing to utilize the CSP.

The communication between backend applications and the CSP is based on Apache ActiveMQ; a JMS compliant message broker.

A copy of the SmartAMM API, including documentation, is placed in the CSP installation folder upon installation.



1-1: System overview

1.1 Requirements

All components require Java SE 6.

Backend applications may be based on other languages, providing that a language specific implementation of the API is developed. See also: http://activemq.apache.org/openwire.html.

1.2 SmartAMM Communication Service Provider (CSP)

This is the communication hub of the SmartAMM system. It is capable of communicating with gateways using SMS, UDP or TCP communication. The CSP backend listens to a JMS compliant message queue for messages to send, and delivers incoming messages to one or more message queues. The CSP can handle multiple backend clients, including subscriptions to messages from specific SmartAMM gateways.

1.3 SmartAMM API

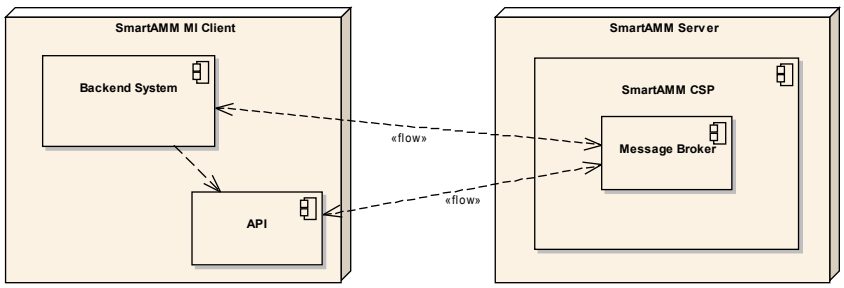
This is the API used for handling messages transmitted and received by the CSP. It also contains a message queue client, which encapsulates handling of message queues, enabling the application programmer to use method calls and callbacks for message delivery and reception. It is used by the CSP as well as integrating backend systems.

1.4 Usage scenarios

In the following are the principles for a few of the most commonly used usage scenarios described.

1.4.1 Embedded message broker

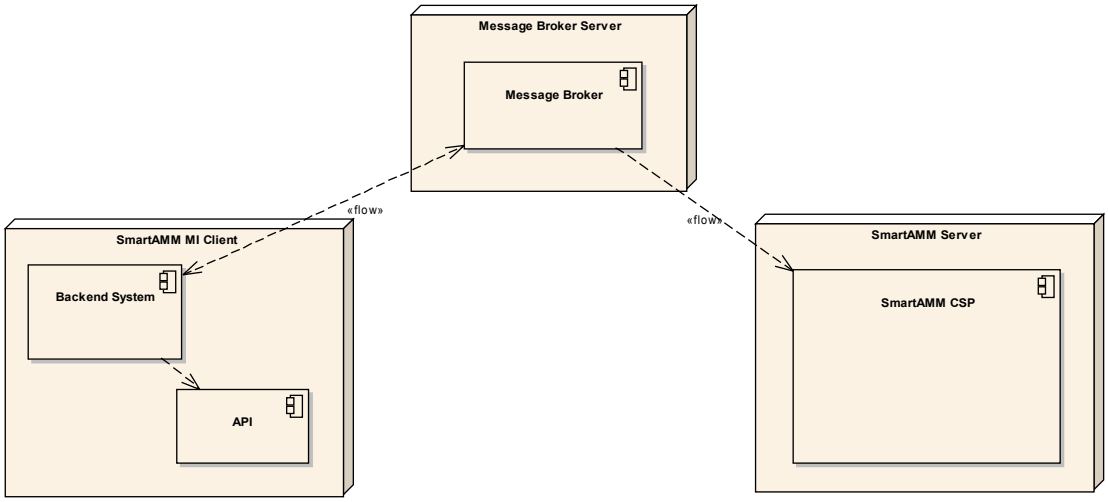
The CSP is capable of hosting an internal message broker, which can be used for receiving as well as delivering telegrams.



1-2: Embedded broker

1.4.2 External message broker

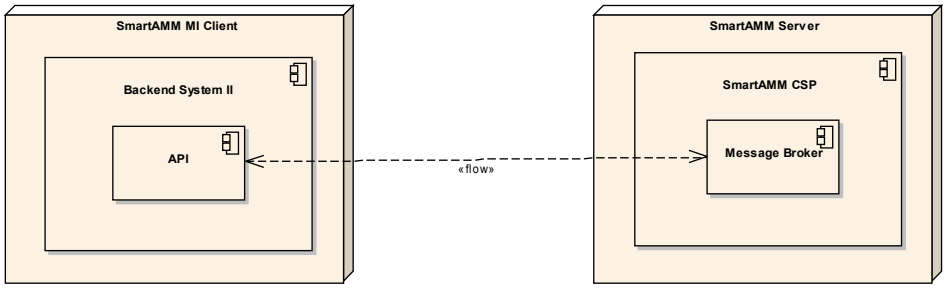
When the CSP is used in an enterprise environment, it might be required to receive and deliver message from/to an existing message broker service, possibly hosted in a fail-safe setup (such as clustering and/or fail-over).



1-3: External message broker

1.4.3 Using the API

The API contains an implementation of a client for the CSP, providing encapsulation of the handling of network objects. By using this client, it is possible to call the CSP using simple method calls without any knowledge of ActiveMQ/JMS.



1-4: Using the MI Client

2 CSP Deployment

The CSP is currently available in 32 bit versions for windows and linux. The windows package consists of an installer which installs the SmartAMM windows service automatically. The Linux package is an archive which contains a very simple installation description. The installation package also contains a file called *readme.txt* which contains descriptions of the included/installed folders and files.

2.1 Configuration

The configuration is done by editing the configuration file *config.xml* which is located in the *conf* folder.

The file describes the selected components of the system as well as their individual configurations.

A description of the configuration options can be found in section 4.1: CSP Configuration.

2.1.1 Service/Daemon configuration

The Windows Service and the Linux daemon is implemented using Java Service Wrapper, which has its own configuration file named *wrapper.conf* in the *conf* folder.

See http://wrapper.tanukisoftware.org/doc/english/properties.html for a detailed description of the configuration options.

2.1.2 Initial configuration

Prior to starting the CSP for the first time, the following settings should be revised:

wrapper.conf:

- path to java binary
- additional/JMX parameters
- log levels (console/logfile and system log)
- log file format
- log file rotation scheme
- windows nt service dependencies, if applicable (eg. database/backend service)

config.xml:

- Backend (the default is the ActiveMQ backend described in this document, but others may be used)
  - client broker uri + queue name
  - inbound broker uri + queue name
  - policy for delivering to inbound queue (*inboundqueuefallbackonly* true/false)
- Network

- IP Dispatcher
  - *tcp* + *udp* : address + port(s)
  - commenting (or removing) out a section (*udp* or *tcp*) disables the corresponding protocol
- SMS Dispatcher
  - The default gateway type is *ipmodem*, which is able to utilize a Multitech IP modem for transmitting and receiving SMS'es.
  - The settings for *hostaddress* and *hostport* should be revised.

2.2 Monitoring

The default configuration for the Java Wrapper Service sets it to deliver logging with level *error* and *fatal* to the system log (Windows Event Log).

It is possible to configure another log level for the wrappers own log file, which is located in the *logs* folder.

2.2.1 JMX interface

In addition to logging, it is possible to perform detailed real-time monitoring and management of the CSP through JMX. The interface is configured in the Java Service Wrapper configuration, and the default setting is currently a non-restricted access at port 9999.

3 SmartAMM API

The SmartAMM API consists of a jar file containing the classes to be used for handling messages and the API documentation for these classes.

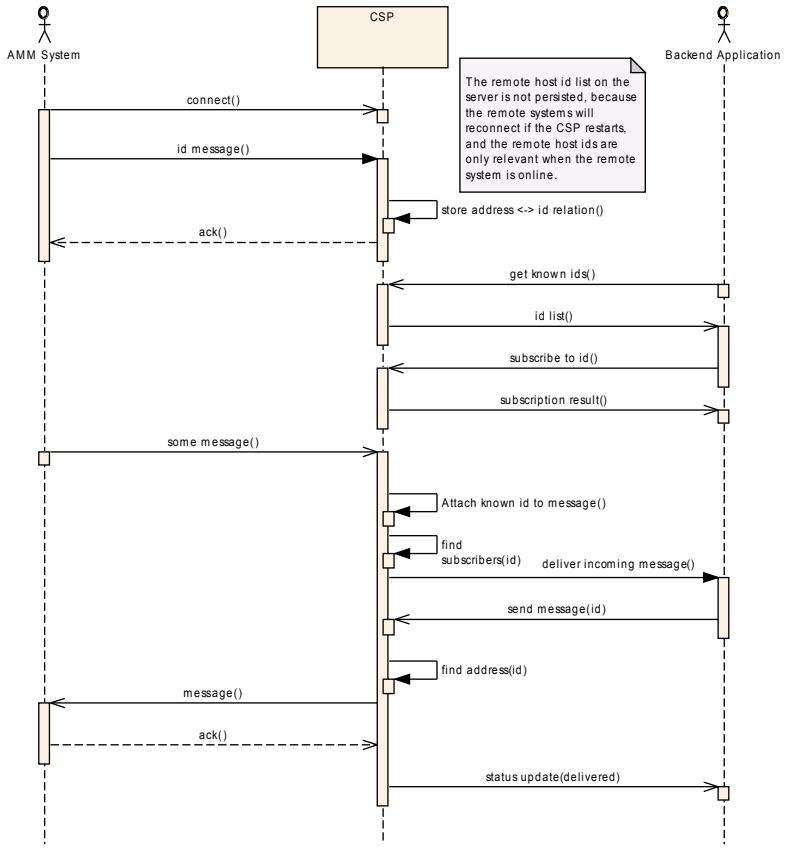
This section contains an explanation of some of the behavior of the system, as well as examples of how to use the API.

The API documentation is created using Javadoc and is packaged with the jar file.

The entire API is packaged in a zip file.

3.1 Identification of remote hosts and message delivery

In order to identify the remote hosts on networks with a dynamic addressing scheme, such as IP over GRPS and other IP based networks (using DHCP), the AMM Systems sends an message containing an id, each time they connect to the server. The id used by the user/backend systems to identify each remote host, due to the non deterministic nature of their network addresses. Hosts on networks using static addressing schemes can use their network address (ie. telephone number as identification). The ids of the remote hosts are stored in memory on the server, as they are useless when the corresponding host isn't connected to the server. The sequence diagram below shows the connect procedure.



3-5 : Example connect and usage of the remote host id

3.2 Transport status

When a message is submitted at the CSP for transmission, the CSP delivers status update events to the client, each time the transport status changes.

Possible status values are:

- **Send**
  Set by the backend, to imply that the telegram should be sent.
- **Transmission**
  Indicates that the telegram has been received by the backend, and is been evaluated for transmission.
- **Transmitted**
  The telegram was successfully sent via the network.
- **Delivered**
  Acknowledge received from the destination. The telegram has been successfully delivered.
- **GotReply**
  Reply from the destination. The telegram has been successfully delivered and a reply has been received. (Not applicable for version 3 of the protocol)
- **InvalidDestination**
  The specified destination address was invalid for the CSP.
- **DestinationNotFound**
  The specified destination was unknown at the CSP.
- **TransmissionError**
  An unknown transmission related error has occurred.
- **TooManyOutstandingTelegrams**
  The maximum number of outstanding telegrams has been reached for the specified destination. The telegram is dropped.
- **DeliveryTimeout**
  A timeout occurred when attempting to deliver the telegram. Only applicable when acknowledge or reply is expected.
- **Received**
  Set on all incoming telegrams before they are delivered to the backend.
- **UnknownError**
  An unknown error has occurred.

4 Appendices

4.1 CSP Configuration

The configuration file contains a basic structure, which corresponds to the application layers of the CSP Service. It consists of the basic elements "logging", "core", "backend" and "network" which must always be present.

These elements all contains a sub element in which the name of the class implementing the layer is found. The entire sub tree of the element containing the class name will be submitted as configuration to the specified class when it has been successfully instantiated.

4.1.1 Sections

4.1.1.1 logging

This section contains an element called "loghandlerclass" which must contain the full name of the class to be used for logging, as well as its configuration. The class must be extend the abstract class `com.develco.amr.gw.logging.AbstractLogHandler`.

4.1.1.2 core

This section contains an element called "implementationclass" which must contain the full name of the class to be used as core, as well as its configuration. The class must extend the abstract class `com.develco.amm.service.core.AbstractCore`.

4.1.1.3 backend

This section contains an element called "implementationclass" which must contain the full name of the class to be used for backend, as well as its configuration. The class must extend the abstract class `com.develco.amm.service.backend.AbstractBackendProxy`.

4.1.1.4 network

This section contains a number of elements called "dispatcher", each containing the full name of and configuration for a class to be used as network dispatcher.

The dispatcher classes must extend the abstract class
`com.develco.amm.service.network.AbstractNetworkProxy`.

4.1.2 Available component implementations

4.1.2.1 logging

Class: `com.develco.amm.logging.ConsoleLogHandler`
Prints all log information (including debug) to stdout

Class: `com.develco.amm.logging.SimpleFileLogHandler`
Writes log information to a file. No log rotation available

Config elements:
`filename`
The name of the log file to write to.
`append` (values: `true`/`false`)
If this is false, the specified file is truncated on startup.
`level` (values: `debug`/`info`/`warning`/`error`/`fatal`)
The level at which logging begins.
Ie. when "`debug`" is specified, all levels are logged, when "warning" is specified debug and info logging is omitted

Class: `com.develco.amm.service.logging.JavaServiceWrapperLogHandler`
Delivers logging to the Java Service Wrappers log system.
Configuration is done in the wrapper configuration.

4.1.2.2 core

- Class: `com.develco.amm.service.core.impl.AMRServerCore`
  Currently this is the only available core. It handles transmission, timeout, retransmission, status updates and so on.
  Config elements:
  - `maxoutstandingtelegramsprhost`
    The maximum number of telegrams, requiring acknowledge, that can be sent to a host before a status update must be received.
  - `serviceintervalmillis`
    The number of milliseconds between each timeout check and retransmission invocation.

4.1.2.3 Backend

Class: `com.develco.amm.service.backend.mq.MessageQueueBackend`

This backend sends and receives messages from and to other systems using JMS compliant message queuing. The backend contains a default queue, where messages from hosts without subscribers are sent.

Config elements:

`clientembeddedbroker` (values: `true`/`false`)

Indicates whether to create and use an embedded broker or connect to an external.

`clientbrokeruri`

The uri of the broker containing the inbound queue.

This is the uri that will be used to create the embedded broker, if required.

`clientqueue`

The name of the inbound queue.

`inboundqueuebrokeruri`

The uri of the broker containing the inbound queue

If this is empty or exactly the same as the broker for the inbound queue, that broker is used.

`inbound`

The name of the default queue.

`inboundqueuefallbackonly` (values: `true`/`false`)

Indicates that messages only should be delivered to the inbound queue when there is no subscribers to messages from the sender.

Class: `com.develco.amm.service.backend.PollingBackend`

A generic backend which provides interval based polling, connect and reconnect.

It is meant to be used for communicating with an asynchronous backend such as a database or file system.

It uses a `PollingConnector` for backend communication and message translation.

Config elements:

`pollingintervalmillis`

The amount of milliseconds between each poll operation

`reconnectintervalmillis`

The amount of milliseconds between each attempt to (re)connect to the backend, when the connection is unavailable.

`queueretryintervalmillis`

The amount of milliseconds between each attempt to re-deliver the next telegram from the queue, after a failure.

`pollingconnectorimpl`

The configuration of the PollingConnector. This element contains the name of the implementing class, as well as the configuration for it.

Class: `com.develco.amm.service.backend.impl.ConsoleOutputBackend`

Prints all incoming telegrams to `stdout`.

Class: `com.develco.amm.service.backend.impl.LogFileBackend`

Writes all incoming telegrams to a log file.

Config elements:

`filename`

The name of the log file to write to.

`append` (values: `true`/`false`)

If this is false, the specified log file is truncated on service startup

4.1.2.4 network

Class: `com.develco.amm.service.network.impl.IPDispatcher`

Transmits and receives messages over IP based networks.

Uses UDP and/or TCP for transmission and reception.

Config elements:

`maxattempts`

The maximum number of transmission attempts for each telegram.

`timeoutsecs`

The number of seconds to wait between each transmission attempt.

`udp`

Subconfig for udp transport.

`listenaddress`

The ip address to listen to for incoming data (0.0.0.0 for all)

`listenport`

The udp port to listen to for incoming data

`defaultremoteport`

The port to send to, if none is specified for outbound messages.

`tcp`

Sub config for tcp transport.

`listenaddress`

The ip address to listen to for incoming data (0.0.0.0 for all)

`listenport`

The tcp port to listen to for incoming data

Class: `com.develco.amm.service.network.impl.SMSDispatcher`

Transmits and receives messages using number of sms gateways, currently there is no way of prioritizing the gateways for outgoing messages. When sending a message, a round-robin scheme is used for selecting the gateway for transmitting the message. Only the gateways marked as outbound will be used in this situation.

The gateway types must be known to the SMS Dispatcher, and currently "`serialmodem`" and "`ipmodem`" is available.

Config elements:

`maxattempts`

The maximum number of transmission attempts for each telegram.

`timeoutsecs`

The number of seconds to wait between each transmission attempt.

`gateway`

Sub config for one gateway

`type`

The type of gateway.

`inbound`

The gateway can receive SMS'es.

`outbound`

The gateway can send SMS'es.

The rest of the subconfig-elements depends on this.

Type: `serialmodem`

This gateway type is able to handle transmission and reception of sms'es using a serial modem.

`serialport`

The name of the serial port to which the modem is connected. Eg. Windows: COMn, Linux: /dev/ttyN.

`baudrate`

The baudrate to communicate at.

`pin`

The pin code for the sim card.

`manufacturer`

The manufaturer of the modem. (This may influence on the selection of AT command set.)

`model`

The model of the modem. (This may influence on the selection of AT command set.)

Type: `ipmodem`

This gateway type is able to handle transmission and reception of sms'es using a ip modem.
Currently the only supported modem is the MultiTech MultiModem.

`hostaddress`

The network address of the modem. May be a host name or an IP address.

`hostport`

The port to connect to at the given `hostaddress`.

`manufacturer`

The manufaturer of the modem. (This may influence on the selection of AT command set.)

`model`

The model of the modem. (This may influence on the selection of AT command set.)

4.2 Client examples

The examples below are from the class `com.develco.amm.example.ExampleClient`, which can be found in the jar file.

The example code below may not be accurate due to changes in the code base, such as refactoring, but the version found in the jar file could at least compile.

4.2.1 Client setup

```
public class ExampleClient {

	// This class implements interfaces required to get all events from the
	// MessageQueueClient
	private MessageQueueClientListener messageQueueClientListener;

	// This is the actual client, which connects to the server.
	private SmartAMMMessageQueueClient messageQueueClient;

	// This is the uri of the broker containing the inbound queue of the server
	private String brokerUri = "tcp://server:62000";
	// This is the name of the inbound queue. The default name is "inbound"
	private String queueName = "inbound";

	/* The constructor creates the client and connects to the server */
	public ExampleClient() {

		messageQueueClientListener = new MessageQueueClientListener();
		messageQueueClient = new SmartAMMMessageQueueClient(messageQueueClientListener,
				messageQueueClientListener, messageQueueClientListener,
				messageQueueClientListener, brokerUri, queueName);
	}
```

4.2.2 Client connect

```
/*
 * This method shows an example of a connect procedure The method also
 * requests a list of remote host id from the server.
 */
public void connectToServer() {

        // The connect method is synchronous, but will also call the
        // IConnectionLister.connected method when finished
        boolean connected = messageQueueClient.connect();

        if (connected) {
          // Requesting a list of known remote host ids at the server
          // The reply is delivered to the
          // ICommandReplyListener.remoteHostsListReply() method.
          messageQueueClient.getRemoteHostIdList();
        }
}
```

4.2.3 Client subscription

```
/*
 * This method contains examples of subscribing and unsubscribing remotehost
 * ids at the server.
 */
public void subscriptionExamples() {

        // Remote host ids are acquired from the server, using the
        // MessageQueueClient.getRemoteHostIdList() method

        // Subscribe to a single remote host id
        messageQueueClient.subscribe("Remote host id");

        // Subscribe to all incoming messages
        messageQueueClient.subscribeAll();

        // Unsubscribe a single remote host id
        messageQueueClient.unsubscribe("Remote host id");

        // Unsubscribe all remote host ids
        messageQueueClient.unsubscribeAll();

}
```

4.2.4 Sending a message

```
/*
 * This method shows how to populate and send a telegram
 */
public void sendATelegram() {

	// Creating the telegram.
	// The class "V3TelegramEncapsulation" extends the class
	// "AbstractTelegramEncapsulation" which is the class known by the
	// server and transport mechanisms.
	// There is also another implementation of telegram encapsulation, used
	// for versions 1 and 2 of the transport protocol.
	// See the javadoc for details
	SmartAMMTelegram telegram = new SmartAMMTelegram();

	// The Message telegram type is actually set as the default type.
	telegram.type(TelegramType.Message);
	// The acknowledge telegram will be returned by the destination SAP.
	telegram.ackRequired(true);
	// The SAP to which the telegram sent.
	telegram.destinationSAP(0xAB);
	// Setting the source sap dosn't make sense here, since it's done by the
	// gateway upon reception of the telegram
	// telegram.sourceSAP(0xCD);
	// Setting the telegram number is actually done by the network
	// dispatcher, so setting it here doesn't make any sense.
	// telegram.setTelegramNumber(0xEF);
	// Set the payload of the telegram.
	// This is the actual information to send to the remote system.
	telegram.payload(new byte[] { 0x12, 0x34, 0x56, 0x78 });

	// The telegram transport contains and handles information relevant for
	// handling the telegram at the server.
	// Only the telegram itself is sent to the gateway.
	SmartAMMTransport transport = new SmartAMMTransport(telegram);

	// Setting the status is not needed - it will be set to Send by the
	// MessageQueueClient
	// transport.setStatus(..);

	// In order to be able to identify the destination network address to
	// send the telegram to, the transport
	// needs to contain a remote host id or an network address object.
	// Setting a remote host id
	transport.setRemoteHostId("some remote host id");
	// or a network address (which has higher priority than the id)
	// This will not work, since there's no ip address set.
	transport.setNetworkAddress(new TcpEndpointAddress());
	// The dispatcher selects the transport media/network based on the type
	// of network address set.

	// Transmit the message using the MessageQueueClient
	// If null is returned, the transmission failed
	// Status updates are delivered to the IMessageListener.statusUpdate
```

```
        // method along with the message id.
        // The boolean value indicates whether or not to automatically subscribe to
        // all messages from the remote host id and network address of the SmartAMMTransport
        String messageId = messageQueueClient.transmit(transport, true);
    }
```

4.2.5 Class implementing all event listener interfaces

```
/*
 * This class implements the interfaces required to get all events from the
 * MessageQueueClient
 */
private class MessageQueueClientListener implements IConnectionListener,
            IMessageListener, ICommandReplyListener, IExceptionListener {

    /*
     * This method is called by the MessageQueueClient when a connection
     * attempt is complete, succesful or not.
     */
    public void connected(boolean yes) {
        System.out.println(String.format("Connection attempt %s",
                yes ? "succeeded!" : "failed!"));
    }

    /*
     * This method is called by the MessageQueueClient when the server
     * disconnects or when someone calls disconnect(true) on the
     * MessagQueueClient
     */
    public void disconnected() {
        System.out.println("Disconnected from the server");
    }

    /*
     * This method is called when a new message arrives
     */
    public void message(SmartAMMTransport message) {
        System.out.println(String.format("Incoming message: %s", message));
    }

    /*
     * This method is called each time the status for a telegram changes.
     * The supplied messageId is the id assigned to the telegram on delivery
     */
    public void statusUpdate(SmartAMMTransport message, String messageId) {
        System.out.println(String.format("The transport status for the message " +
                                         "with id \"%s\" is now %s",
                                         messageId, message.getStatus()));
    }

    /*
     * Called whenever an exception occurs in the MessageQueueClient.
     */
    public void onException(Throwable t) {
        System.out.println("An exception occured in the MessageQueueClient, " +
                           "some action should probably be taken here.");
```

```
        }

        /*
         * Called when a requested list of remote host ids known at the server
         * is received.
         */
        public void remoteHostsListReply(List<String> remoteHostIds,
                        String requestId) {
                System.out.println(String.format("A reply to a request for known remote host" +
                                                 " ids has been received (containing %s ids):",
                                                 remoteHostIds.size()));
                for (String id : remoteHostIds) {
                        System.out.println(id);
                }
        }

        /*
         * Called when a reply to a subscription request is received.
         */
        public void subscriptionReply(boolean success, String requestId) {
                System.out.println(String.format(
                                "An attempt to subscribe to remotehost id was %s ",
                                success ? "successful!" : "unsuccessful!"));
        }

}
```

4.3 Third party software

4.3.1 ActiveMQ

Home: http://activemq.apache.org
Brief: Open Source implementation of the Java JMS specification.
Usage: Backend communication; integration to administrative and other systems.

4.3.2 MINA

Home: http://mina.apache.org
Brief: Open Source networking framework.
Usage: Sending and receiving messages over TCP and UDP.

4.3.3 Smslib

Home: http://smslib.org

Brief: Open Source project for transmitting, receiving and interpretation of SMS messages, using serial or IP based communication.

4.3.4 Java Service Wrapper

Home: http://wrapper.tanukisoftware.org

Brief: Open Source project which provides a unified and simple way of using a Java application as a Windows (NT) Service or a Linux Daemon.

4.3.5 Nullsoft Scriptable Install System (NSIS)

Home: http://nsis.sourceforge.org

Brief: A very flexible installation packager.

Usage: Packaging the CSP installation, as well as the installation for the Service tool (not covered by this guide)