M2MGate Network Communication Framework

Whitepaper

INSIDE M2M GmbH

1st April 2011

Summary

There are many existing frameworks for communication between distributed software components. M2MGate Network is a framework developed in Java that is specially optimised for communication via a narrow-band and comparatively unstable network connection, such as GPRS.

The framework is divided into two separate components: a J2ME-compliant terminal component and a J2SE-compliant server component. Communication between the two components takes place by means of reciprocal method calls.

This document describes both parts of the framework and gives an example of its implementation to illustrate how it is used.

Contents

1 Architecture

M2MGate Network is a communication framework developed by INSIDE M2M GmbH. This framework makes it possible to carry out remote method calls in order to guarantee communication between distributed software components. Its purpose is to overcome the problems associated with GPRS as a transmission medium in order to provide safe, stable and narrow-band data transmission.

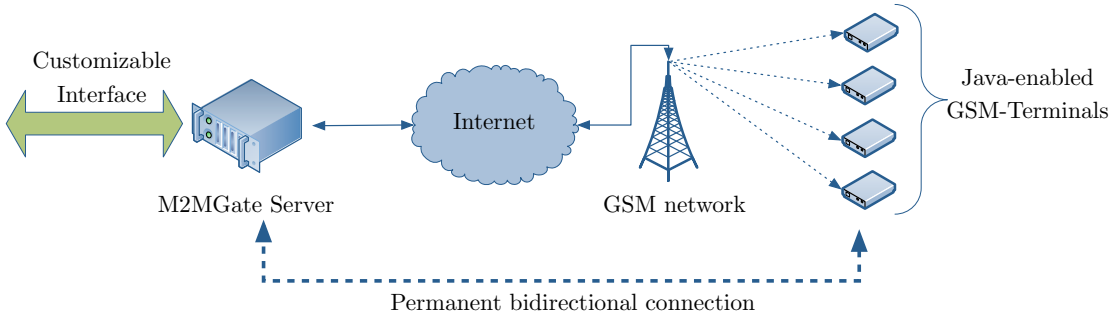

Figure 1: M2MGate Network-architecture

The framework comprises a server component (called the M2MGate Server) and a terminal component (called the M2MGate DeviceServer). These two parts provide a foundation and can be updated for special applications. Each GSM terminal maintains a permanent TCP connection to the M2MGate server. All communication with the terminals takes place via these connections.

1.1 M2MGate Server

The server component is a Java application that runs within a JVM (Java Virtual Machine) which supports JavaSE Version 5.0 or later version. So this component is both platform-independent and operating system-independent. Depending on the application, an interface to existing systems can be integrated into the M2MGate server. Thus, for example, a web service or CORBA interface can be made available. So the M2M Gate server serves as a kind of proxy which grants access to the terminals connected to it.

1.2 M2MGate DeviceServer

The terminal component is also a Java application. In principle, any Java-enabled GSM terminal can be used as a hardware-platform. Terminals equipped with Type TC65, TC65i or XT65 engines from the company, Cinterion Wireless Modules, are typically used. These provide a limited Java runtime environment, called JavaME. The M2MGate DeviceServer software is optimized for both the runtime environment and for the relatively limited computational power provided by the terminal.

Figure 2: Java-enabled GSM terminal from MC Technologies with Cinterion TC65 engine [2]

2 Remote Method Invocation

Communication between the terminal component and the server component takes place by means of reciprocal method calls. This means that the actual application is clearly separate from the communication of data. Hence, no network code, e.g., dealing with sockets, has to be written within the actual application. Instead, a method call takes place and data is transmitted by means of the framework.

2.1 Technical Implemantation

The technical implementation of the framework is based heavily on existing communication-orientated middleware solutions such as Java RMI [1]. Two proxy objects, called the stub and the skeleton, are created. The stub object has the task of receiving a local method call within a VM, serializing all parameters and sending them via a network channel to the skeleton object. The skeleton object, which runs within a different VM, receives the data, retrieves the parameters and calls the actual method.

If the method sends a return value, the skeleton object serializes it and transmits it via the network channel to the stub object. The stub object receives the data, retrieves the return value and sends the return value created at the remote station to the original local method call. Figure 3 illustrates the process described here.

One significant advantage compared to other solutions is that overhead is kept very low during data transmission. All communication occurs in a binary format and is reduced to the bare minimum necessary.

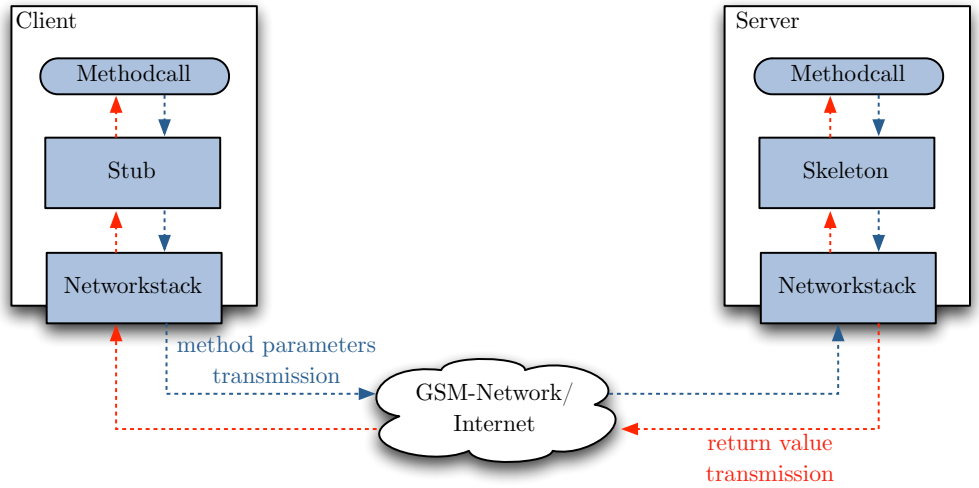

Figure 3: The Remote Method Invocation-process

3 Example of Implementation

The following example follows the UML diagram in Figure 4.

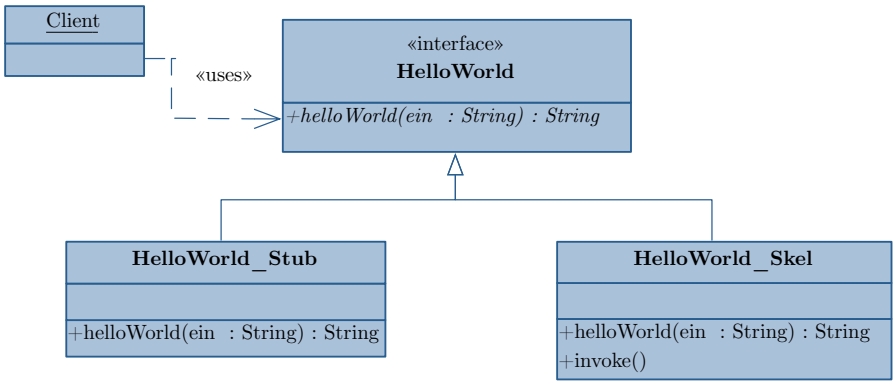

Figure 4: The diagram is simplified and does not represent all methods and classes for clarityś sake. The classes represented are called 'remote components'

Firstly, an interface is defined containing the methods which will be remotely called later (see Listing 1).

```
1  public interface HelloWorld{
2    public String helloWorld(String myname) throws IOException, M2MException;
3  }
```

Listing 1: Hello World Interface

Next, the stub and skeleton classes which carry out the parameter serialisation and return value serialisation are implemented.

```
1  public class HelloWorld_Stub extends M2MComponent_Stub implements HelloWorld
       {
2
3    public String helloWorld(String myname) throws IOException, M2MException {
4      if (closed) {
5        throw new M2MClosedException();
6      }
7      final Trafo trafo = this.trafo;
8      try {
9        RequestHandler msg = trafo.getRequestHandler();
10       DataOutput out = msg.invoke(1, ref, 10);
11       out.writeUTF(myname);
12       DataInput in = msg.fetch();
13       String result = in.readUTF();
14       return result;
15     } catch (IOException ioex) {
16       trafo.exceptionOccured(ioex);
17       throw ioex;
18     }
19   }
20 }
```

Listing 2: HelloWorld_Stub

The class 'HelloWorld_Stub' in Listing 2 extends the class M2MComponent_Stub. This provides the basic function of being able to communicate with the remote station via the network using a transformer. Line 10 indicates to the remote station that a method is being called, and the number 10 is transmitted as the method reference. Then the string type parameter is serialized and written into the network stream. The 'fetch()' method call indicates that all parameters have been written into the network stream and that the method can be carried out at the remote station. Subsequently, the return value of the method which has been carried out is read using 'readUTF' and returned.

'HelloWorld_Skel' in Listing 3 implements the 'Skel' interface. Among other things, this defines the method, ïnvoke(int,int,ResponseHandler, Trafo)¨; which is called by the framework. The 'mid'-parameter is used to indicate which method the remote station wants to call, which corresponds to the method reference in line 10. In lines 12 and 13 the parameters are read from the network stream, in line 14 the method is called, and in line 15 the return value is transmitted to the remote station.

```
1  public class HelloWorld_Skel implements Skel, HelloWorld{
2
3    public void invoke(int cmd, int mid, ResponseHandler answer, Trafo src)
         throws IOException,
4         M2MException {
5      switch (mid) {
6      case 0: {// close
7        answer.answer(0);
8        return;
9      }
10     case 10: {
11       DataInput in = answer.paras();
12       String myname = in.readUTF();
13       String result = helloWorld(myname);
14       DataOutput out = answer.answer(1);
15       out.writeUTF(result);
16       answer.done();
17       return;
18     }
19     default: {
20       throw new M2MException("HelloWorld_Skel.invoke() unknown MID=" + mid);
21     }
22     }
23   }
24
25   public String helloWorld(String myname){
26     System.out.println(``helloWorld executed on server, myname=''+myname);
27     return ``I am the server'';
28   }
29   \\...
30 }
```

Listing 3: HelloWorld_Skel

3.1 Connector

The connector is part of the terminal software component and has the following functions:

- Establishing a TCP connection to the M2MGate server.
- Initialising all registered stub objects so that remote method calls can be carried out on the server.
- Calling the invoke method with all skeleton objects so that the M2MGate server can carry out remote method calls at the terminal.
- Re-establishing a broken connection to the M2MGateServer.

The source code in Listing 4 demonstrates the application of the connector and carries out a remote method call.

```
1  class Example{
2    public static final String VERSION = "1";
3    public static void runExample() throws Exception{
4
5      HelloWorld_Stub helloWorld = new HelloWorld_Stub();
6      ReportStationaryDevice_Stub report=new ReportStationaryDevice_Stub();
7      Stub[] stubs={ report,helloWorld };
8      Invokable[] skels = {};
9      String imei = "IMEI" //query imei..
10     String[] roots = new String[]{"socket://m2mgate.de:5000"}
11     int reboots = 0;  //query reboots
12     short midletVersion = 1;
13
14     StationaryTC65Connector connector;
15     connector = new StationaryTC65Connector(imei, roots, invokables,stubs,0
           x622,VERSION,report,midletVersion,reboots,null);
16     connector.connect();
17
18     String result;
19     result = helloWorld.helloWorld("Test")
20   }
21 }
```

Listing 4: Benutzung des Connectors

In the lines 5-11 all variables which are necessary for the connector are initialized. The connector is created in line 15 using the following parameters:

imei unambiguous identification number of the terminal

roots data regarding connection to the 'M2MGate Server' (IP address and port number)

invokables All skeletons to be registered. In this case an empty array will be transmitted, since the M2MGate server will not carry out methods at the terminal.

stubs stubs to be initialised. Two components are transferred, including the Hello World example given.

0x622 is an identification number which informs the M2MGate server which stubs and skeletons are available at the terminal

VERSION The terminal software version as a character string

report a stub component which is used by the connector to transmit connection information to the M2MGate server

midletVersion The terminal software version as a character string

reboots number of start procedures of the terminal. This value is transmitted to the 'M2MGate Server'.

After the connector has been initialized, the connection to the M2MGateServer is made in line 16. Then a remote method call takes place in line 19, and the method is carried out on

the M2MGate server.

Figure 5 represents a remote method call as a sequence diagram, following the example given.

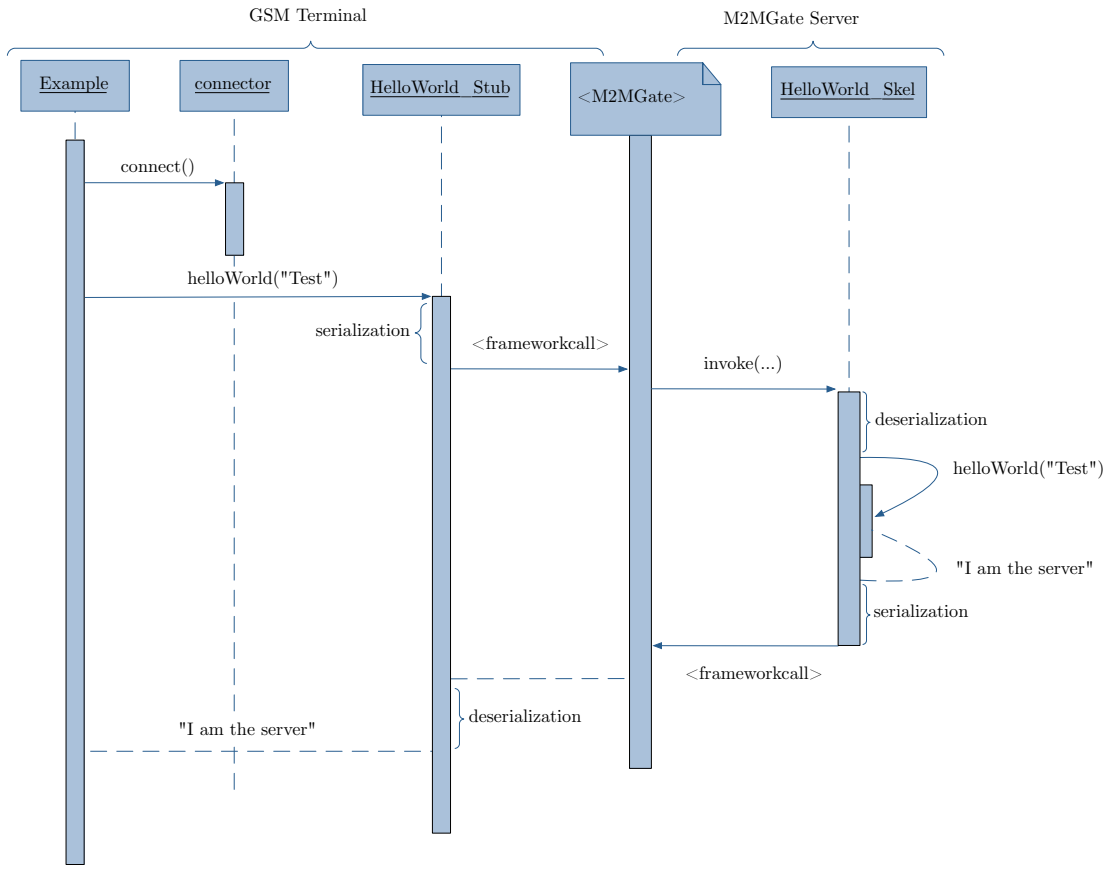

Figure 5: Simplified sequence diagram illustrating the HelloWorld example. The diagram is simplified and does not represent all methods and classes for clarity's sake.

3.2 Error-Handling

During a remote method call errors can easily occur. For example, the terminal could be in a dead zone or simply switched off. For a remote method call, these network errors are indicated by IOExceptions. When developing this application, error handling, which was neglected in this example, must be provided, because a 100%-reliable connection cannot be assumed.

References

[1] Java RMI. http://java.sun.com/javase/6/docs/technotes/guides/rmi/, Juli 09.

[2] Mc product catalog. http://www.mc-technologies.net/downloads/katalog.pdf, Juli 09.