MyMIDP: An JDBC driver for accessing MySQL from mobile devices

Hagen Höpfner and Jörg Schad and Sebastian Wendland and Essam Mansour
International University in Germany
School of Information Technology
Campus 3, D-76646 Bruchsal, Germany

E-mail: *FIRSTNAME.LASTNAME*@i-u.de (*replace ö by o)*

Abstract

Cell phones are no longer merely used to make phone calls or to send short or multimedia messages. They more and more become information systems clients. Recent developments in the areas of mobile computing, wireless networks and information systems provide access to data at almost every place and anytime by using this kind of lightweight mobile device. But even though mobile clients support the Java Mobile Edition or the .NET Micro Framework, most information systems for mobile clients require a middle-ware that handles data communication. Java's JDBC provides a standard way to access databases in Java, but this interface is missing in Java ME. In this paper we present our implementation of an MIDP-based Java ME driver for MySQL similar to JDBC that allows direct communication of MIDP applications to MySQL servers without a middleware.

1. Introduction and Motivation

Nowadays ubiquitous, nomadic, and pervasive computing is not a longer vision but reality. Devices become smaller and easier to carry around. Accessing world wide information via wireless links is possible almost anytime and everywhere. Hence, cell phones or smart phones are no longer voice communication devices but small footprinted information system (IS) clients.

Even though most current mobile devices provide an environment for third party applications – most prominently Java ME's Connected Limited Device Configuration (CLDC) in combination with the Mobile Information Device Profile (MIDP) – there is a lack of standardized software for combining IS and mobile devices. Most existing approaches like SyncML or Hotsync concentrate on synchronization aspects or use an additional middle-ware. A direct access to data from a database server like MySQL is not possible. However, due to the recent developments in the areas of mobile hardware and wireless networks, this approach becomes more and more reasonable. Java desktop applications often use an JDBC driver for this purpose. In order to fill the gap we implemented an MIDP-based Java ME driver for MySQL similar to JDBC that allows direct communication of MIDP applications to MySQL servers without a middle-ware.

The remainder of this paper is structured as follows. Section 2 includes a brief introduction to the development of mobile applications with Java ME and illustrates the overall architecture of the driver. Section 3 describes how the driver can be used from within an MIDP application. Section 4 explains the limitations of our driver. Section 5 relates our work to other approaches. Section 6 briefly introduces MyMIDP-Client, a prototype MySQL client that uses the MyMIDP driver. Section 7 summarizes the paper and shows open issues that will be researched in the future.

2. Overall Architecture

In this section we present the driver architecture. As this requires some background information into Java ME, we would like to first introduce Java ME before giving a detailed architectural overview. We assume that the reader knows a little bit about programming with Java.

2.1 Java SE vs. Java ME delevopment

Java ME development differs significantly from development with Java SE. In Java SE, all clients have identical class libraries and virtual machines – given they use the same Java version – as they are standardized by Sun. With Java SE – and especially with the Java EE extension – Sun tries to fullfill the needs of every application developer.

Java ME on the other hand is based on the concept of minimal building blocks. First, so called *configurations* define a basic runtime environment. This environment only defines "a *minimum complement* or the '*lowest common denominator*' of Java technology", is "possibly incomplete for real target devices" and "*shall not define any optional features*" [14]. This results in execution environments with less than 80 classes for the popular CLDC in version 1.1. In comparison, Java SE has more packages than CLDC has classes.

Second, so called *profiles*, such as MIDP, define additional libraries on top of a configuration for specific device categories or purposes. Profiles adhere to the same general design principles as configurations but can differentiate between obligatory, optional and recommended features [15].

Third, a number of standard APIs from the JSR process define additional features like encryption, content handling, Bluetooth access and more. They are again bundled in JSR plattform definition standards (for example JSRs 75[1], 185[2] and 248[3]) to simplify development (as adherence to a plattform definition implies the support for all bundled standards). For more information on the differences in the Java versions we refere to Suns Java website `http://java.sun.com`.

As almost no two mobile devices are equal, each manufacture must implement his own version of the Java virtual machine on his devices. By doing so the manufacture can choose which standards and features should be implemented. This creates a problem because each Java ME implementation differs in some aspects. Additionally, each implementation comes with certain restrictions. For example, many virtual machines limit the maximum Jar file size and ignore everything above this limit. (A limit of about 64kB was and is very popular, especially on low-end devices.)

2.2 Preliminary considerations

Before we started the development we set ourselves four design goals:

1. Keep the driver API as near to the JDBC specification as possible
2. Keep the Jar file size below 32kB – half the popular 64kB limit – to leave enough space for the application
3. Only implement required features
4. Keep the implementation code as simple and performant as possible

[1]JSR 75: PDA Optional Packages for the J2ME™ Platform: `http://jcp.org/en/jsr/detail?id=75`

[2]JSR 185: Java™ Technology for the Wireless Industry: `http://jcp.org/en/jsr/detail?id=185`

[3]JSR 248: Mobile Service Architecture: `http://jcp.org/en/jsr/detail?id=248`

These goals were mostly achieved. Our current development version provides database access sufficient for most applications in just 27kB. (In comparison, the MySQL JConnector JDBC driver has more than 500kB.) On the other hand we had to cut short on some aspects like parametrized queries and meta data usage.

2.2.1 Analysis

In order to communicate successful with a database server, a number of problems must be solved:

- Packet sequence control (available packet types and their expected order and content)
- Packet assembly and dissembly (order and type of packet fields and their expected or allowed values)
- Packet field encoding and decoding (types of fields and how are they encoded)
- Protocol flags (what are the protocol options and their meanings, which combinations are allowed or expected)
- Password encryption algorithm (how is the password encrypted during authentication)
- MySQL data types and their structure (how is database content transfered to the client)
- Database meta data (what kind of database meta data is available and must or should be analysed)

Below we give some information on some of the problems we encountered and their solution.

2.2.2 Conversion between Java and MySQL data types

The CLDC runtime environment only implements basic types but not extended SQL data types as provided by the Java SE *java.sql* package nor the large data types provided by the *java.math* package. This meant from the beginning that not every MySQL data type could be natively supported or without loss of usability.

To standardize our conversation approach, we set up a number of rules:

- All MySQL data types are converted in their nearest Java counterpart if possible
- Signed and unsigned numbers are handled the same way
- MySQL date and time data types are converted to the Java *Date* type only if it is possible with minimal effort, otherwise process as string

- Finally, process all inconvertible data types as string (i.e. MySQL *DECIMAL*, *ENUM* and similar)

As the MySQL client/server protocol uses two different data encoding schemes, namely 'converted to string' and 'binary as stored' [9], we had to implement two decoding routines. This results in a few conversion problems as we could not match both implementations perfectly. Fortunately one can work around this problem by avoiding problematic data types. In Section 4 we give additional information on this delicate topic.

2.2.3 MySQL client/server protocol documentation

Even though MySQL has a very good end user documentation, the kind of in-depth information we needed is not readily available, especially not background information on flags and field. This meant we had to look for documentation at a number of different places, including source code and captured packets.

Unfortunately, the information we gathered was sketchy at some places and even wrong at others. For example, one can not use the *CONNECT_WITH_DB* option of the authentication process as described in the MySQL Internals documentation but must specify the database afterwards. This fact is stated nowhere but was discovered during out tests and solved through packet analysis.

2.2.4 Application developer API

As stated before, our goal is an API very similar to the JDBC one. This meant identifying the necessary functionality and methods as well as making some design decisions. For example, we decided to drop all interfaces and to implement the functionality described directly as classes, using the same method and class names. In the end we implemented three JDBC interfaces, namely *Connection*, *Statement* and *ResultSet*. Additionally we implemented a *Query* class to provide missing functionality for parametrized statements.

2.3 Class overview

Figure 1 shows the basic class diagram of the driver. As one can see, the driver consists of just eight classes (plus four helper classes and exceptions not shown). Below follows some detailed information on the tasks of each class.

2.3.1 Buffer

The *Buffer* class is responsible for encoding and decoding packet fields as well as for the conversion between MySQL and Java data types. The large number of different field and encoding types in the client/server protocol results in a large number of methods, making this class the largest one. Its inner workings are inspired by the Buffer class of the MySQL JConnector and uses an automatically growing byte array.

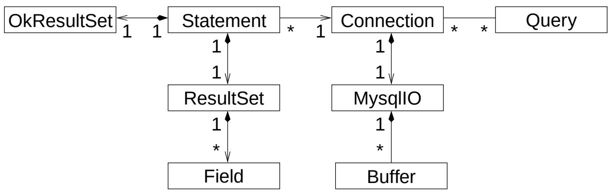


Figure 1. Class Overview

The aim with this class was to keep memory consumption small and to reduce garbage collection as it is quite processing intensive. Reusing a byte array as much as possible is a good way to do this. Only in case the array is to small to hold all data, a new, bigger one is created and the data is copied over. In addition, by making the array larger then actually needed, the growing only happens infrequently, thus keeping the effort small. Internal pointers keep track of the actually used array space and the 'reserve'; internal boundary checks perform the necessary pointer adjustments.

2.3.2 MysqlIO

The MysqlIO class handles the communication with the database server. It contains methods to assemble and dissemble all packet types, but does not have any clue about the packet sequence. Most other classes can access an instance of this class and use its methods to perform their tasks.

The *MysqlIO* class uses two *Buffer* instances, one for sending and one for receiving, to which it has exclusive access, ensuring strict task separation between the classes.

2.3.3 Connection

The *Connection* class is very similar to the *Connection* interface of JDBC. It owns an instance of *MysqlIO* and uses it to provide connection specific methods like opening and closing a connection and changing the database. It also works as factory for *Statement* and *Query* objects.

2.3.4 Statement

The *Statement* class is very similar to the JDBC *Statement* interface. It provides methods to execute database queries and fetch the the result. For this it implements the packet sequence logic necessary, but relies on the instance of the *MysqlIO* class hold by the *Connection* class factory for doing all packet processing.

2.3.5 Query

The *Query* class provides basic functionality for parametrized queries. It uses a string buffer to replace occurences of question marks in the query string with the desired parameters. Because MIDP does not provide any regular expression capabilities, the algorithm works strictly linear, does not consider already processed or inserted parts of the query string and does not offer any escape capabilities.

2.3.6 Field, ResultSet and OKResultSet

These three classes manage the retrieved database data. The *OKResultSet* is an simple query information storage and is solely used by the *Statement* class. It is not directly available to the application developer but must be accessed through methods provided by the *Statement* class.

The *ResultSet* class performs the same job as the JDBC *ResultSet* interface, providing exactly the same row-pointer based access methods. It uses an array of *Field* class instances to store and process all column specific data. In fact, the *ResultSet* does only act as a facade to the *Field* class, managing the row dimension of the database result set.

The *Field* class provides column wise storage for database result sets and meta data. A large number of simple methods provide access to specific column- and meta data. It is only used internally by the *ResultSet*.

2.3.7 Helper classes

Three additional helper classes provide a number of static methods for common tasks not really part of the driver (like string operations). An additional *Constants* class contains all the necessary constants. Finally, there is one *SQLException* class used throughout the driver.

3 Using MyMIDP

3.1 Device requirements

Every MIDP 2.0 compatible device should be able to use the driver when the following, additional requirements are fulfilled.

- The mobile device must support socket connections (optional in MIDP 2.1)
- The mobile device must support JSR 177[4] (needed for MySQL authentication via SHA-1)
- The device should have at least one megabyte of free heap memory (depending on the implementing application)

3.2 Simple usage scenario

Since the driver API is a very similar to JDBC, a developer familiar with JDBC will not have any problems using our driver. And even developers new to database APIs will find our driver easy to use as it always follows four steps.

1. Create a database connection
2. Create and execute a database statement
3. Process the result set
4. Close the connection

Steps two and three can be repeated in case more than one query must be executed.

For illustration purposes the following listing shows a short usage example.

```
import de.iu.db.mysql.mini.Connection;
import de.iu.db.mysql.mini.ResultSet;
import de.iu.db.mysql.mini.Statement;
import de.iu.db.mysql.mini.exceptions.SQLException;

public class Demo {

  public static void main(String[] args) {

    try {
      // connecting to database 'catsanddogs' on server
      // test.somenetwork.net:3006, user 'test', pwd 'run'
      Connection con = new
          Connection("test.somenetwork.net", 3006,
          "test", "run", "catsanddogs");

      // retrieve some data
      Statement st = con
        .createStatement
        ("SELECT␣name,␣age,␣owner␣FROM␣dogs");
      ResultSet rs = st.executeQuery();

      // loop through the result set
      for (; rs.current() < rs.getResultCount(); rs.next()) {
        // access the data using row and column pointer
        System.out
            .println("The␣Dog␣" + rs.getAsString(0) + "␣("
                + rs.getAsInt(1) + ")␣is␣owned␣by␣"
                + rs.getAsString(2));
      }

      // adding some data
      long count = st
        .executeUpdate
          ("INSERT␣INTO␣dogs␣(name,␣age,␣owner)" +
          "VALUES␣('Angel',␣12,␣'Charlie')");
      System.out.println
          ("Added␣" + count + "␣datasets␣with␣message:␣"
          + st.getMessage());

      // end the session
      con.close();
    } catch (SQLException e) {
      // do some error handling
      e.printStackTrace();
    }
  }
}
```

[4]JSR 177: Security and Trust Services API for J2ME™: `http://jcp.org/en/jsr/detail?id=177`

3.3 Main differences to JDBC

There are a few important usage differences to JDBC we would like to point out. First, the driver does not use the JDBC style connection URL but method parameters for simplicity and performance reasons. Second, the *Statement* object can be reused thus improving garbage collection. Third, it is not possible to execute multiple statements at the same time as they use the same *MysqlIO* class instance and thus share buffers.

4 Limitations

Because of the previously stated restrictions of the CLDC/MIDP runtime, we had to compromise and drop a number of features available in the normal MySQL JDBC driver. Following is an overview of the limitations and their cause.

4.1 No support for pre 4.1 MySQL server

The most significant restrictions of our driver is the missing support for pre 4.1 MySQL server. This decision was caused by two facts.

First, version 4.1 of the MySQL server introduced an enhanced version of the client/server protocol. This new protocol differs significantly from the previous versions, providing a new authentication mechanism using SHA-1 and extended meta data support. These changes had a price, namely the loss of its backward compatibility. Hence two protocol implementations would be necessary to support pre 4.1 server as well as current ones. This would have required hundreds of additional lines of code at the prize of an increased `jar` file size.

Second, the extended support time frame for all pre 4.1 MySQL server will end at the end of 2008. [10] This made the likelihood of encountering an unsupported server version unlikely at best.

4.2 Limited character set support

The CLDC specification states that the "Character information is based on the Unicode Standard, version 3.0. However, since the full character tables required for Unicode support can be excessively large for devices with tight memory budgets, by default the character property and case conversion facilities in CLDC assume the presence of ISO Latin-1 range of characters only. More specifically, implementations must provide support for character properties and case conversions for characters in the '*Basic Latin*' and *Latin-1 Supplement*' blocks of Unicode 3.0. Other Unicode character blocks may be supported as necessary." [14]

For this reason, character conversation capabilities are very limited in CLDC. In order to keep our driver small and fast, we decided to limit all string conversions to the ISO Latin-1 range and to use the standard conversations provided by the String class instead.

This means a number of restrictions for both the MySQL server and the mobile device. First, the server must use Latin-1 for all communication with the client and second the client device must use the Latin-1 character set as default. Also, the target database should use the Latin-1 character set but this is not necessary if the returned strings are post processed by the application.

4.3 No support for prepared statements

In the current implementation the driver does not support prepared statements. This has two reasons. First, the implementation code is quite complex, requiring the implementation of at least five additional packet types. Second, the protocol documentation in the MySQL Internals wiki is still 'tentative' and incomplete [9].

4.4 Limited data type support

The client/server protocol uses two different encoding schemas: All standard result sets are returned as a character string – non-character types are converted into a string – while results from prepared statements and default column values are returned as binary [9]. To keep the implementation size small, we did not match both conversation routines perfectly. This is especially true for the date and time data types.

In the current implementation state this problem is not very serious as we use the binary conversion only at one place – decoding the default column value. But in future implementations with support for prepared statements this might be a problem.

Additionally, we recommend to use only types which can be converted directly by Java (i.e. signed integers, floating-point numbers, character strings and timestamps) as these types can be used without complication.

4.5 No transaction handling

Due to `jar` size considerations, we droped any built in support for transaction handling as we deemed it not necessary for mobile devices. The fact that MySQL just uses standard SQL commands for transaction handling (SQL *START TRANSACTION*, *COMMIT* and *ROLLBACK*) and no custom command codes (as it does for the *USE* command), made this decision easy as an application developer can add his own transactional logic if needed.

4.6 No meta data analysis

In the current development state the driver does not analyse any server meta data like *SERVER_STATUS_AUTOCOMMIT* and *SERVER_STATUS_CURSOR_EXISTS* as it is of no practical use. This must be changed if more advanced features are required.

4.7 Limited error handling

Again, `jar` size considerations had us cut the error handling capabilities. The entire driver only defines one custom exception. All exceptions thrown inside the driver are encapsulated into this exception and handed up to the application. On the other hand, to ensure proper error handling, all caught exceptions are attached to ours and handed up thus allowing more extending error handling if needed by the application.

Another feature we deemed unnecessary is the conversion of standard MySQL error messages into the standard SQL error statements as it is done in JDBC. This kind of error handling is not needed on a limited device as it does not provide any additional information and as such is only needed in applications with exchangeable data sources. However, the MySQL server passes the standard SQL error code in its error messages and is as such available to the application.

4.8 No support for parallel processing

In the current driver version we try to reuse objects as much as possible to keep the memory footprint small and the garbage collection to a minimum. On the other hand, this means that parallel operations like multi threading will not work properly as data is lost or overridden. During normal operation this does not play any role as any database operation will be processed fully before an API method returns, thus leaving the connection always in a determined state. Only if the executing thread is interrupted, the connection might end up in an unknown state. Anything can happen from this point on.

4.9 Limited SQL support

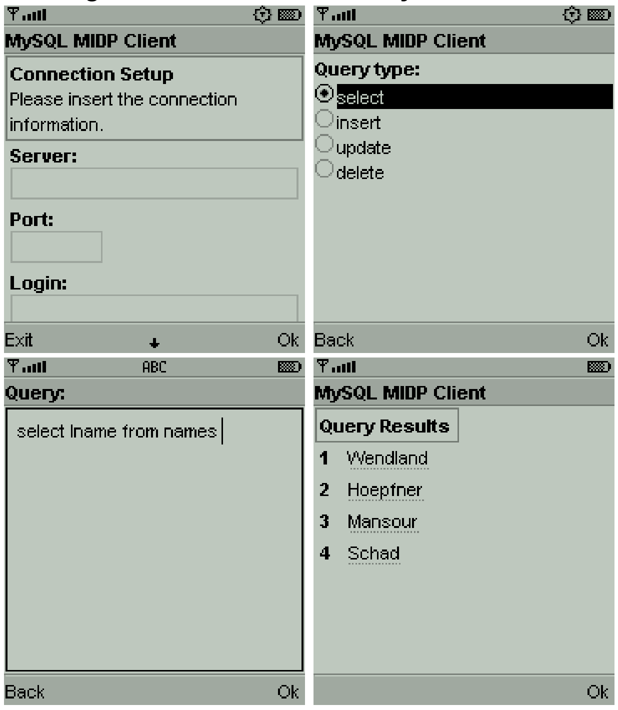
The client/server protocol uses special packets for some SQL commands, for example for the SQL *USE* but also for the MySQL specific administration commands. Currently, with the exception of *USE*, the driver does not implement these special packets, they are hence not supported.

5 Related Work

Almost all vendors of database management systems offer a lightweight version of their database management for mobile devices [3]. However, to our knowledge none of them allows a direct communication between the application and the database management system. Oracle Lite [11, 13, 12] and IBM's DB2 Everyplace [4, 5] use a middleware approach for synchronizing data between client and server. Microsoft's SQL Server CE [8] needs Active Sync and Sybase Adaptive Server Anywhere [16] either uses SQL-Remote and its message oriented replication or MobiLink as a session based approach. All these systems are designed for handling replicated data [7] but not for simple client/server data access. Furthermore, no one of the major players supports MIDP devices so far.

In previous works [1, 6] we used a simple web service that forwards queries to the server and returns the result to the requesting client using an HTTP-connection. However, this approach is comparable to the middle-ware solutions and requires additional software (the web service) that might be an additional point of failure. We aim at developing a pure client/server solution without additional middleware or messaging server.

Figure 2. Screenshots of MyMIDP-client


MySQL MIDP Client
Connection Setup
Please insert the connection information.
Server:
Port:
Login:

MySQL MIDP Client
Query type:
[✓ select]
insert
update
delete

Query:
select lname from names

MySQL MIDP Client
Query Results
1 Wendland
2 Hoepfner
3 Mansour
4 Schad

6 MyMIDP-Client

Figure 2 shows four screenshots of the MyMIDP-client, a MyMIDP based prototype MySQL client application that has been implemented for demonstration purposes [2]. After inserting the connection information (server address, port, user, password, database) the client offers alternative query types to the user. However, this step helps reducing the text the user has to typ in via the cell phones' keyboard. It results in the third screen that allows to complete the query. The next step is to submit the query. Due to security reasons each java application authomatically requests confirmation before establishing any internet connection. The screen of this request, that is not shown in Figure 2, is followed by a screen that displays the query results. This query result is the result of the direct communication between our client and a MySQL server via the MyMIDP driver. Due to the limited display size we decided to show only the first attribute of each tuple and to number the tuples. The remaining attribute values can be displayed by selecting the respective row. However, this function managed by the application only and *no* feature of the MyMIDP driver.

7 Summary and Outlook

In this paper we presented our implementation of a MIDP-based Java ME driver for MySQL. Similar to JDBC for Java SE it allows direct communication of MIDP applications to MySQL servers without a middle-ware. Hence, it is possible to directly connect a mobile device like a cell phone to a MySQL server without using a middle-ware. We discussed implementation details as well as how to use the driver. Furthermore we described existing limitations which mostly result from the restrictions of Java ME and MIDP.

The driver is only the first step in the direction of a full MySQL support for mobile devices. Wireless transmissions are expensive and relatively slow. So we plan to include caching strategies that reduces the retransmissions of data. Furthermore, we plan to complete the driver with the communities help. Therefore, we will make the source code available under the GPL.

NOTE: The MyMIDP sources and the MyMIDP-client prototype implementation are GPL lizensed and available at `http://it.i-u.de/dbis/MyMIDP`.

References

[1] A. Caracaş, I. Ion, M. Ion, and H. Höpfner. Towards Java-based Data Caching for Mobile Information System Clients. In B. König-Ries, F. Lehner, R. Malaka, and C. Türker, editors, *MMS 2007: Mobilität und mobile Informationssysteme; Proceedings of the 2nd conference of GI-Fachgruppe MMS; March 06, 2007, Aachen, Germany*, volume P-104 of *Lecture Notes in Informatics (LNI) - Proceedings*, pages 97–101, Bonn, Germany, 2007. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH.

[2] H. Höpfner, J. Schad, S. Wendland, and E. Mansour. MyMIDP and MyMIDP-Client: Direct Access to MySQL Databases from Cell Phones, Mar. 2009. BTW-DEMO.

[3] H. Höpfner, C. Türker, and B. König-Ries. *Mobile Datenbanken und Informationssysteme — Konzepte und Techniken*. dpunkt.verlag, Heidelberg, Germany, July 2005. in German.

[4] IBM Corporation. *IBM DB2 Everyplace Application and Development Guide Version 8.2*, Aug. 2004.

[5] IBM Corporation. *IBM DB2 Everyplace Sync Server Administration Guide Version 8.2*, Aug. 2004.

[6] I. Ion, A. Caracaş, and H. Höpfner. MTrainSchedule: Combining Web Services and Data Caching on Mobile Devices. *Datenbank-Spektrum*, 21:51–53, May 2007.

[7] B. König-Ries, C. Türker, and H. Höpfner. Informationsnutzung und -verarbeitung mit mobilen Geräten – Verfügbarkeit und Konsistenz. *Datenbank-Spektrum*, 7(23):45–53, 2007. in German.

[8] Microsoft Corporation. `http://msdn.microsoft.com/library/`, 2008.

[9] MySQL AB. *MySQL Internals ClientServer Protocol*, 2008. Retrieved on January 18, 2008 from `http://forge.mysql.com/wiki/MySQL_Internals_ClientServer_Protocol`.

[10] MySQL AB. *MySQL Lifecycle Policy*, 2008. Retrieved on February 21, 2008 from `http://www.mysql.com/about/legal/mysql_lifecycle_policy.pdf`.

[11] Oracle Corporation. *Oracle Database Lite, Administration and Deployment Guide 10g (10.0.0)*, June 2004.

[12] Oracle Corporation. *Oracle Database Lite, Developer's Guide 10g (10.0.0)*, June 2004.

[13] Oracle Corporation. *Oracle Database Lite, SQL Reference 10g (10.0.0)*, June 2004.

[14] Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California, 95054 U.S.A. *Connected Limited Device Configuration (CLDC) Specification*, 1.1 edition, Mar. 2003.

[15] Sun Microsystems, Inc. and Motorola, Inc. *Mobile Information Device Profile Specification*, 2.1 edition, May 2006.

[16] Sybase Inc. `http://www.sybase.com/ianywhere/products`, 2008.