JAVA-C++ BRIDGE FOR SYMBIAN BASED SMARTPHONES

Balázs Goldschmidt, Gergő Gyánó and Zoltán László
Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Magyar tudósok krt 2, Budapest, Hungary

Keywords: Symbian, smartphone, j2me.

Abstract: Authors propose a simple framework in order to help smartphone developers gain both from the advantages of J2ME and C++ in Symbian OS. The idea is that while JNI is not supported by J2ME, with the help of the built-in networking support a daemon written in C++ can serve as a function-provider in cases when problems with pure J2ME features can not be solved.

1 INTRODUCTION

The penetration of mobile phones is continuously increasing. In 2006 more than 30 countries passed 100% penetration.(Wallace, 2006) The proportion of smartphones among recent buyers in 2006 was 8.8% in Western Europe.(Telephia, 2006; m:metrics, 2007)

The major advantage of smartphones is that new applications can be installed on them after the device has been released and sold. In a wide range of devices the new applications can be Java midlets(J2ME, 2002) or native applications. The benefit is obvious: at purchase time the phones don't have to have all possible applications installed on them, it is the customer who can install new software later.

When developing a mobile application, the developer has to choose an implementation basis (J2ME or native OS). The most widespread native platform is Symbian(Symbian, 2005) that is supported by major smartphone manufacturers. This is why authors focused their attention on this platform alone.

J2ME is a well-defined subset of Java SE that is supported by smartphones. The phones usually provide a virtual machine (KVM), that runs the Java applications, while different specifications describe further support like the use of bluetooth, etc.

Advantages of developing J2ME applications are (1) wide acceptance: most newly released mobile phones support J2ME, (2) gradual learning curve and ease of use: programmers with knowledge of Java standard edition can easily learn the specifics and specialties of J2ME, and can easily develop and deploy J2ME midlets.

The disadvantage of J2ME applications is their limited access to native resources. Although a lot of JSR-s are specified for accessing native resources (bluetooth, calendar and files, SMS, etc), only a few of these specifications are implemented in different phones (for Nokia phone capabilities see figure 1).

Symbian platform provides an OS and runtime environment for applications usually written in C++. The advantages of developing applications for this platform are (1) wide acceptance: many smartphones that allow consumers to install new software on them run a version of Symbian; (2) low level access to system resources: when writing native applications, the programmer is allowed to access the phone's resources – like communication stacks, calendar entries and files, etc. – directly.

The disadvantages if this platform include (1) a steep learning curve: learning to develop Symbian applications even for programmers with firm C++ knowledge is hard. To get accustomed to the development process, to conforming special requirements, etc. need lots of training and effort. (2) Difficult development: the development process of Symbian applications needs significantly greater effort than that of J2ME.

Authors motivation was to make it possible to combine the advantages of both platforms. Let the

Goldschmidt B., Gyánó G. and László Z. (2008).
JAVA-C++ BRIDGE FOR SYMBIAN BASED SMARTPHONES.
In *Proceedings of the International Conference on Wireless Information Networks and Systems*, pages 241-244
DOI: 10.5220/0002027402410244
Copyright © SciTePress

JSR	API description
75	File & PIM
172	Web services
177	Security and Trust Services
179	Location
180	SIP
234	Advanced Multimedia Supplements
248	Mobile Service Architecture for CLDC

JSR	5320 N82 E51	6212	6600	N90 N70
75	●	●	●	●
172	●	●		●
177	●	●		
179	●			
180	●			
234	●	●		
248		●		

Figure 1: JSR support of selected Nokia smartphones. JSR-s supported by all phone types are not shown here. (Source: *http://www.forum.nokia.com*).

developers write the major part of an application in Java, and implement only specific parts C++, parts that need features not supported by J2ME or the JSRs implemented in smartphones.

In order to combine the advantages of the two platforms, however, the usual method used in J2SE, where the necessary native code based modules can be inserted into Java applications using Java native interface (JNI, 2003; Liang, 1999), can not be applied, because J2ME doesn't support JNI at all. It was also not an option for authors to create a new J2ME virtual machine that supports such native codes. The intention was to find a solution that is standards-based, portable and easy-to-use.

The basic idea is to create a simple C++ application, a daemon, that provides the necessary functions, and which can be accessed over TCP/IP by the Java applications that need the functionality.

2 GENERAL ARCHITECTURE

2.1 Major Modules

The general architecture of the framework can be seen on figure 2. It consists of the following major modules:

- **C++ Daemon.** Generated by the framework. This module hosts the *C++ function skeleton* that implements the necessary functionality. The module accepts TCP/IP client connections, and calls the required functions.
- **Java Function Stub.** Generated by the framework. This function can be called from the Java midlet, handles serialization, connects the *C++ daemon*, and returns the values passed back by the daemon.
- **C++ Function Skeleton.** Implemented by the developer. It gets the serialized parameters passed over TCP/IP, performs the necessary functionality, and returns the result.
- **Java Midlet.** The Java midlet that performs some task that depends on a functionality provided by the C++ daemon. It is implemented by the developer.

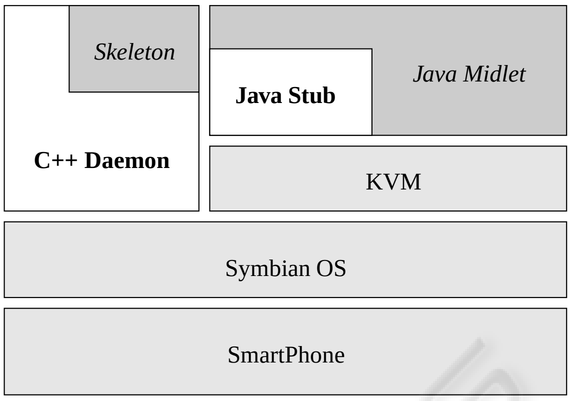

Figure 2: The general architecture of the framework.

Legend: *italic*: to be implemented by programmer, **bold**: generated by framework, roman: shipped with smartphone.

2.2 Communication

The general communication scenario is the following:

1. Midlet needs a functionality provided by the daemon. Calls the function of Stub (e.g. *add(3, 5)*).
2. Stub serializes parameters into a string, opens a TCP/IP connection to the daemon, sends the operation id and the parameters as character stream, then waits for the return values.
3. Daemon accepts the connection, reads operation id and parameters, calls the skeleton assigned to the operation id, and passes parameters.
4. Skeleton deserializes the parameter values from the character stream, then performs its task.
5. Daemon serializes the return value into a string, and sends it to Stub.

6. Stub receives the data, and deserializes the return value and returns it to the caller Midlet. TCP/IP connection is closed.

2.3 Protocol

The protocol of parameter-passing during the TCP/IP communication is a simple string-based data-transfer. Currently only integers, doubles and arrays of integers or doubles are supported.

sending operation id and parameters uses this form:

`<id><values>`

where `id` is the operation id coded as an integer, and `values` is the parameters. Both serialized as follows.

sending an integer or double uses this form:

`<L>:<V>`

where `<L>` stands for the length of variable, `V` stands for the value of variable.

Eg.: `3:1234:45.6` → values sent: 123 and 45.6.

sending array of integers or doubles uses this form:

`<LLA>:<LA><L#1>:<V#1>...<L#n><V#n>`

where `LLA` stands for the length of the integer describing the length of the array, `LA` stands for the length of the array.

Eg.: `1:23:1233:456` → array of two elements sent: (123, 456)

3 EXAMPLE APPLICATION

In this section a simple example is shown. The task is to create a function that is passed two integers, and returns the sum of them. The Java Stub on the client side looks like this:

```
public int add(int a, int b) {
    ClientCommunication c =
        new ClientCommunication();
    c.setCommand(F_ADD);
    c.add(a);
    c.add(b);
    if (!c.communicate())
        return Integer.MIN_VALUE;
    return c.getInt();
}
```

- The constructor of *ClientCommunication* initializes variables and the input and output character-streams.
- The function *setCommand()* sets the command parameter to *F_ADD*, a constant that is defined on both sides of the communication.
- The function *add()* adds the parameters to the character-stream that is to be sent to the C++ daemon.
- The function *communicate()* sends the character-stream to the daemon and receives the return values, which are then put into the output character-stream. If the communication fails, it returns false.
- The function *getInt()* retrieves an integer from the output character-stream set by *communicate()*.

The function is implemented in the skeleton (C++ server side) as follows:

```
// reads in two integers
// and returns their sum
void CSRemoteFunctions::add() {
    TInt ia = getInt();
    TInt ib = getInt();
    TInt ret = ia+ib;
    addInt(ret);
}
```

Function `add()` is accessed through an array of function-pointers. After implementing the function it has to be put into this array:

```
#define F_ADD 0
#define F_SUB 1
#define F_SUM 2
...
void CSRemoteFunctions::ConstructL() {
    funcArray[F_ADD] =
        &CSRemoteFunctions::add;
    funcArray[F_SUB]   =
        &CSRemoteFunctions::sub;
    funcArray[F_SUM] =
        &CSRemoteFunctions::arraysum;
    ...
}
```

When the daemon receives the client connection and reads in the function id, it calls the respective function, and the return value is converted back to string format and sent back to the Java stub.

4 FRAMEWORK API

The functions needed to implement the Java stub are collected in class **ClientCommunication**. The class provides the following public methods. *XXX* stands for *Int* or *Double*, *xxx* stands for *int* or *double*.

- **ClientCommunication()** Constructor. Initializes the input-output streams.

- **void add(*xxx* i)** Adds an integer or a double to the values sent to the daemon.
- **void add(xxx[] array)** Adds an array of integers or doubles to the values sent to the daemon.
- ***xxx* get*XXX*()** Retrieves an integer or double from the return values.
- ***xxx*[] get*XXX*Array()** Retrieves an array of integers or doubles from the return values.
- **void setCommand(int cmd)** Sets the identifier of the command to be called on the daemon's side.
- **boolean communicate()** Opens the network connection, sends the parameters, reads return values, and closes the connection.

The following functions can be used during C++ skeleton implementation:

- **T*XXX* CSRemoteFunctions::get*XXX*()**
  Reads an integer or double value from the input stream that was received from the Java Stub.
- **CArrayFixFlat<T*XXX*>* CSRemoteFunctions::get*XXX*Array()**
  Reads an array of integers or doubles from the input stream that was received from the Java Stub.
- **void CSRemoteFunctions::add*XXX* (T*XXX* x)**
  Writes an integer/double to the output stream that will be sent to the Java Stub.
- **void CSRemoteFunctions::add*XXX*Array (CArrayFixFlat<T*XXX*>* array)**
  Writes an integer/double array to the output stream that will be sent to the Java Stub.

5 MEASUREMENTS

Tests were conducted to measure the overhead of the function call in the framework between the Java stub and C++ skeleton. For the test the *add* function introduced above was used. It was called 1000 times from a J2ME midlet. The function was also implemented in Java for control purposes. The smartphone used in the tests was a Nokia N90.

The results show that the average execution time was *78.53ms* when the connection was not opened for each call, *105.33ms* when for each call a new connection was established, and *0.014ms* when the Java implementation of the function was called.

The measurements show that the overhead of the network communication is considerable. Still, authors are convinced that for numerous problems the idea proposed offers the only solution.

6 SUMMARY AND FURTHER WORK

Authors proposed a simple framework in order to help smartphone developers gain both from the advantages of J2ME and C++ in Symbian OS. The idea is that while JNI is not supported by J2ME, with the help of the built-in networking support a daemon written in C++ can serve as a function-provider in cases when problems with pure J2ME features can not be solved.

The results described in the paper are preliminary. Authors consider the following the major focal points in further development.

Adding more parameter types. The current set of parameter types (int, double, and their arrays) is insufficient. Authors plan to support more primitive types (boolean, char, byte), as well as compound types (like structs in IDL).

Automated skeleton and stub generation. Currently all code is written by the developer. In the near future authors will develop an simplified IDL-like language for specifying the functions, and a simple compiler that generates both C++ skeleton code and Java stub code.

Protocol revision. The protocol currently used has to be revised regarding efficiency and fault-tolerance.

Performance measurements. It has to be measured how the framework performs in different circumstances.

REFERENCES

J2ME (2002). *Java 2 Micro Edition*. http://www.jcp.org/en/jsr/detail?id=68.

JNI (2003). *Java Native Interface Specification*. http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html.

Liang, S. (1999). *The Java Native Interface Programmer's Guide and Specification*. Prentice Hall PTR.

m:metrics (2007). Mobile market measures. http://images.servicesmobiles.fr/press/passport-sping2007.pdf.

Symbian (2005). *Symbian OS*. http://www.symbian.com/.

Telephia (2006). Americans lag behind europeans in smartphone adoption.

Wallace, B. (2006). 30 countries passed 100% mobile phone penetration in q1. *Telecommunications Online*.