**Programming Bluetooth-enabled
devices using J2ME**

Java

in
a
teacup

STEPHEN JOHNSON THALES RAYTHEON

Few technology sectors evolve as fast as the wireless industry. As the market and devices mature, the need (and potential) for mobile applications grows. More and more mobile devices are delivered with the Java platform installed, enabling a large base of Java programmers to try their hand at embedded programming. Unfortunately, not all Java mobile devices are created equal, presenting many challenges to the new J2ME (Java 2 Platform, Micro Edition) programmer. Using a sample game application, this article illustrates some of the challenges associated with J2ME and Bluetooth programming.

J2ME PRIMER

J2ME is Java for embedded devices and consumer electronics. Figure 1 shows the J2ME component stack: configuration is at the lowest level, followed by profile, with optional packages on top. The features (and APIs) provided by a given device depend on the mix of supported configurations, profiles, and optional packages.

The J2ME world is divided broadly into two classes: one servicing devices with less than 512 KB of memory (things you hold in your hand), and the other servicing those with larger memory footprints (things you plug into a wall). These classes of devices are the basis of the two J2ME configurations: CDC (connected device configuration) and CLDC (connected limited device configuration). A configuration determines the VM (virtual machine) and low-level Java API embedded in a device.

The CDC can be thought of as "J2SE lite." Since it is designed for J2SE (Java 2 Platform, Standard Edition) compatibility, source code written for a CDC platform would be indistinguishable from standard J2SE code. The VM used by CDC is compliant with Standard Edition VM specifications.

The CLDC is a restricted subset of the CDC, providing only a fraction of the features available in the Standard Edition. A significant difference is that the CLDC VM

(called the KVM) does not implement all the features of the JVM, such as the bytecode verifier.

Profiles sit above the configuration layer in the J2ME stack. A profile is a collection of features geared to a class of device. For example, an embedded device such as an automotive computer may not require a GUI API, but a PDA phone does. As the number of Java devices grows, more profiles are added through the JCP (Java community process). The choice of profile dictates the configuration used (CDC or CLDC). To a lesser extent, the version of the profile used determines the version of the configuration used. Profiles are generally targeted to a device class. The MIDP (mobile information device profile) is targeted at phones and small PDA applications, while the PBP (personal basis profile) is based largely on the Java TV API and is targeted at set-top box applications.

The combination of a J2ME configuration and profile constitutes a complete development environment for a device. Each profile dictates the type of managed application that the environment supports. Most Java developers are familiar with Web applets. The developer writes a special class (an extension of an applet) that implements callbacks invoked by the applet execution environment (usually within a Web browser). MIDP applications are called MIDlets, and PBP applications are called Xlets, but they generally follow this same pattern. The main differences between these application types are the specific callbacks and the execution environment.

J2ME devices may implement one or more of the optional packages as well, such as Bluetooth support, RMI (remote method invocation), and the mobile media API.

INTRODUCTION TO BLUETOOTH

Bluetooth is a radio-based network protocol that allows devices to participate in ad hoc networks. These networks, called *piconets*, consist of up to eight devices. One of the devices acts as the master, coordinating the timing of

Java in a teacup

communication across the piconet. Piconets can be daisy-chained to form larger *scatternets*, provided each participating piconet has a unique master device. Bluetooth devices operate in the unregulated 2.4-GHz spectrum and provide a range of approximately 30 meters. The relatively short range allows low-voltage and battery-powered devices to communicate. The Bluetooth radio uses a frequency-hopping algorithm with timing dictated by the master device in the piconet. Bluetooth can and will interfere with other communication gear in this spectrum, most notably cordless phones and 802.11 networks.

Perhaps you've heard about Bluetooth's security issues, examples of which include the unauthorized browsing of

another's address book. There is a fundamental assumption in Bluetooth, however, that ad hoc networks are a good thing and that the ability to discover services on anonymous devices is a feature, not a bug. The good news is that most of these loopholes are easily closed—by making the device “not discoverable” and using shared passwords for partnering with other devices. Before we get ahead of ourselves, however, we should examine how Bluetooth connections are made.

Each Bluetooth device has a registry of service information called the SDD (service discovery database), which contains details about services supported by the device. If a Bluetooth device is set as “discoverable,” other devices can find it by broadcasting a query (devices listed as “not discoverable” do not respond to the query). The requesting device can then query the SDD for each of the devices and find out if a given service is supported. If the query is successful, the device replies by sending the service record information (pulled from the SDD) to the requesting device. The requester can then use the service record information to establish a connection with the other device.

The official Java API for Bluetooth (JSR 82) approaches this discovery process elegantly. The API uses the exist-

ing connection framework to establish connections between JSR 82-compliant devices. Each compliant device provides a LocalDevice singleton, which is a proxy for the device itself. The LocalDevice contains a DiscoveryAgent, which is used to query other devices for Bluetooth devices and services. Bluetooth service providers must create the service record and add it to the SDD. JSR 82 ties the service record, service application, and connection together for simplicity. When your application is ready to accept incoming connections to your service, the service record is added to the SDD. Unfortunately, this means that your application has to be running for the service

record to be present in the SDD. This takes away one of the more useful features of Bluetooth, which is discovery of services even when an application is not running.

TARGETED DEVELOPMENT

The first challenge facing the J2ME developer is choosing the target device. Many Java coders are spoiled by the “write once, run anywhere” portability of the standard Java platform. Not so here. J2ME devices are as varied and numerous as snowflakes. There are many combinations of configuration, profile, and optional packages to contend with—in addition to device- and vendor-specific APIs. Unlike standard Java, you cannot assume that the feature set you’re coding for exists on all devices.

For commercial development, the most viable J2ME platform is CLDC. Cellphone applications (and cellphone games in particular) are a rapidly expanding market. Cellphone game download revenue is expected to reach \$8.4 billion worldwide by 2010, according to a *Screen Digest* study. Most Java-enabled phones use some version of the CLDC platform and usually support some version of the MIDP. While most cellphone games are currently single-player, multiplayer games are not that far off. Even a great-selling game like poker would be a lot more interesting with human opponents. Bluetooth would allow multiplayer games without a cellphone signal, relying on the phones’ internal Bluetooth radios to communicate.

The devices section of the J2ME Web site (<http://www.java.sun.com/j2me>) presents a reasonable list of device vendors and products. Most include the configuration and profiles present on the device and a link to the vendor site, which may or may not offer more details. Choosing a feature-rich device makes development easier, but the commercial developer is more interested in a device’s market penetration. Not surprisingly, the most popular devices are often the least expensive—the \$49 wonders that come with a two-year contract. On the other hand, publishers are interested in developers who know how to make these devices “sing.” This involves a fundamental knowledge of Java: threading, garbage collection, and manipulating primitive data types. It also means turning a blind eye to much of the object-oriented best practices you may have grown accustomed to: Once optimized for performance, J2ME code is not often pretty.

Bleeding-edge features are often offset by the potential marketability of the application. In our example, we develop a Java Bluetooth application. Looking at the number of devices supporting JSR 82 is pretty sobering. Plenty of phones have Bluetooth support, but you have to look closely to find devices that specifically support

JSR 82. Our sample application uses the Siemens SL66 mobile phone. It supports MIDP 2.0, the latest version of the profile embedded in most Java-enabled phones. Siemens has an active developer community, which the company supports through its Web site. Not all vendors that make J2ME devices have such accessible developer support. Some wireless carriers (such as Sprint PCS) have development communities as well. It’s worth checking out support levels before selecting a device.

CHOOSING YOUR TOOLS

When you are developing embedded software, there is no substitute for running it on the target device. While the state of the art in emulation and debugging is impressive, running your code on a real-world device can be both humbling and exhilarating. Since hardware debugging environments are prohibitively expensive, emulation is the most popular option. J2ME emulation has three levels of increasing device fidelity: JWTK (Java Wireless Toolkit) with default emulation profiles; JWTK with vendor-provided emulation profiles; and vendor-supplied emulators. The differences in quality of each of these emulation levels are quite noticeable: there are many differences in the display alone; the differences in behavior are even more dramatic—and hard to debug. For new J2ME programmers, we recommend a vendor profile with the JWTK. That combination will get you about 90 percent there. We discuss putting the completed application on the device later in the article.

The JWTK can be run as a stand-alone execution environment, or it can be integrated with Java IDEs such as Sun’s NetBeans. The JWTK integration with NetBeans is seamless and fairly robust. The JWTK also supports the optional Bluetooth API, which saves you from purchasing a third-party development environment.

WRITING THE APPLICATION

An analysis of the cellphone game Paper Clips allows us to demonstrate a program that implements both client and server roles in Bluetooth communication. In the game, players take turns removing paper clips from four rows of clips. Each player must remove at least one clip from a selected row. The player forced to remove the last clip loses. A turn-based game serves as a good example because it is easy to follow. Whereas most PC or console games are fast action games, cellphones encourage a slower-paced gaming style. At the very least, phone games should be playable with one hand and very forgiving of control input errors. Not all phones have game-friendly interfaces. The input lag between pressing a key and

Java in a teacup

processing the event can make “twitch” games very frustrating to play. That isn’t to say you can’t make an action game. As long as there are limited animations on the screen, proper use of Java threads can give satisfactory results. Most current phones can handle an input thread and one or two animation/screen paint threads. You can animate more than one sprite per thread. One of my first games, for example, was a duck hunt, which animated up to three duck sprites and a crosshair. Many J2ME books cover this sort of application in detail.

The Paper Clips application contains two threads: a server thread to respond to connection requests, and a client thread that responds to local user inputs. The beauty of a turn-based game such as Paper Clips is that it is symmetrical. Once the connection between the two devices is established, the behavior is identical, and the game logic has to concern itself only with generating a `GameTurn` object and passing it to the other player. Because the logic is not aware of client or server designation, this structure scales to any number of players, provided you know whose turn is next.

Now you’re ready to set up the Bluetooth communication. The client thread responds to a New Game request by getting the local device’s `DiscoveryAgent` class and sending out a `startInquiry()` broadcast to look for other Bluetooth devices. Note that all `DiscoveryAgent` queries are asynchronous, and the framework communicates through callback functions. For example, the thread that calls `startInquiry()` waits until the callback method `inquiryCompleted()` is called by the execution environment. (If nothing else, writing asynchronous code for twitchy radio devices is a good way to exercise your Java threading skills). Once other devices are discovered, a second query is invoked on the `RemoteDevice`: `searchServices()`. Again, this is an asynchronous call that ends when the waiting thread is notified by `serviceSearchCompleted()`.

If the service search is successful, you now have one or more service records from the other devices. You can send an invitation (a special case of the `GameTurn` object

mentioned earlier) to the other device. This is where the Java Bluetooth implementation shines. You can get a connection to the other device by using the URL pulled from the `ServiceRecord` object:

```
Connector.open( record.getConnectionURL() )
```

At this point, you have a standard Java `StreamConnection` object. You can continue coding the application without being concerned about the underlying protocol. Abstraction of the communication protocol is a good thing, and a feature of the Java connection framework.

This is neat, you say, but how does the application put the service record in the SDD to begin with? You just saw that the client thread uses the `Connector.open()` method to establish a connection to a remote device using the service record URL. The server thread uses a similar trick to establish the service record in the local SDD. When the application begins, the server thread opens a local connection using an URL in the form:

```
Connector.open("btspp://localhost:<UUID>;name=<btName>;authorize=false");
```

The UUID is the universal identifier for your service. The client thread uses it to call the `searchServices()` method. The `btName` parameter is a friendly name associated with the service. The connector is smart enough to know that you’re not attempting to establish a remote connection and does not return a `StreamConnection`, as it did before. In this case, you are given a `StreamConnectionNotifier` object that is linked to that specific service. At this point, the service record has been created. You can add attributes by asking the `LocalDevice` for the `ServiceRecord` and passing the `StreamConnectionNotifier`:

```
localDevice.getRecord(notifier);
```

Finally, you can add the service record to the SDD and listen for any connections to this service by calling `acceptAndOpen()`:

```
notifier.acceptAndOpen();
```

This call returns a `StreamConnection` to the requesting device. As before, you can now treat this connection as a normal Java `StreamConnection`. Once the connection is closed, the service record is removed from the SDD.

In a symmetric, networked application such as Paper Clips, much of the challenge is properly managing the

multiple threads on both phones. As with other Java GUI applications, you don't want to hang the GUI (client) thread. Nor do you want to allow the users to do things they shouldn't. You don't want to let the users feel as if they don't have control, but sometimes you don't want to give them control, either. Fortunately, these problems are easily fixable in emulation. Of course, all you have now is an application that runs on a fake phone. The real fun begins when you load your code onto the device.

BUILDING AND TESTING

Now you're ready to build the application, load it on the device (a process called *provisioning*), and run it. Preparing a MIDlet for deployment involves the following tasks:

- Compile source into class files (bytecode).
- Preverify the class files.
- Update the application descriptor.
- Package the MIDlet suite into a *.jar file.

You might be wondering what *preverify* means. As mentioned earlier, the CLDC VM does not implement a bytecode verifier. The computational overhead of verifying incoming bytecode was deemed too expensive for CLDC-class devices and was left out. This means that bytecode verification needs to be done outside the VM, before the class is loaded at runtime. This presents a security issue, since verification by the VM class loader (at runtime) guarantees that code being processed passes verification immediately before it's used. Separating the runtime VM from the verification process allows a larger security hole for hostile applications. The MIDP 2.0 specification attempts to mitigate this risk using code signing, certificates, and a domain-based security model. This process is outside the scope of this article, but you can read about it in the MIDP 2.0 specification.

The application descriptor contains metadata and attributes of your MIDlet. Unlike descriptors in other Java execution environments, the JAD (Java Application Descriptor) is external to the application (MIDlet) jar file. This is done to assist in OTA (over the air) provisioning. The MIDP 2.0 specification formalizes this method for distributing MIDP applications (bundled in suites) over a wireless (HTTP) network. The simplest case of OTA works as follows:

1. The user discovers a MIDlet suite by browsing JADs.
2. The device checks the JAD for security settings, memory requirements, configuration, and platform versions.
3. If the MIDlet suite is compatible, the device downloads the suite *.jar file using an URL in the JAD.
4. The device installs the MIDlet suite after approval by the user.

From a development perspective, OTA isn't the most efficient way to get your code on the phone. It requires a phone with an active account, Internet service, and developer access to a public Web server. The Web server has to understand the JAD and JAR mime types, and most do not. If you are developing for several phones, it can get very expensive.

Many phones allow you to upload data via a USB cable, though not all will automatically load a MIDlet suite JAR. A newer phone such as the Siemens allows file transfer via Bluetooth and is smart enough to recognize the JAR as a Java application and install it in the Applications folder of the phone UI.

Once it is loaded on the phone, the game might take some work to get it running properly. For example, the discovery agent's `startInquiry()` might not appear to be asynchronous—that is, the client thread is blocked, which freezes the application GUI—but then when you load the code onto another JSR 82 phone, the application behaves much as it did on the emulator. Bugs such as these are very difficult to find, especially without a hardware debugging capability. This type of problem—differing vendor implementations of the Java specification—should be familiar to anyone who has done Java Enterprise Edition development. Given the number of competing Java vendors and the difficulty of writing specifications that focus on outward behavior (without specifying implementation), these problems occur more often than we would like.

CONCLUSION

This article has attempted to provide enough information to acquaint you with J2ME. By taking you through the development cycle and onto the phone, we hope to have shared some of the excitement (and pain) of developing for a mobile device. If you spend a lot of your time developing software for large systems, J2ME provides a different and enjoyable challenge. ☺

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

STEPHEN JOHNSON is a principal software engineer of the Command View II product line at Thales-Raytheon Systems in Fullerton, California. His interests include software architecture and net-centric systems. His current pursuit is improving the interoperability of command-and-control systems through service-oriented architecture. Johnson holds a Sc.M. in system engineering from George Mason University. © 2006 ACM 1542-7730/06/0400 \$5.00