Bluetooth Programming Problems

Andrew Davison, 17th February 2011, ad@fivedots.coe.psu.ac.th

Over the last few weeks I've been coding using the Java API for Bluetooth (variously known as JSR-82 or JABWT), encountering problems that make Bluetooth coding a time-consuming, irritating, and disappointing experience. Some of the issues are due to my use of Java, some to my choice of Windows XP, but most are inherent in Bluetooth's specification. I'll describe these issues under the broad headings of multiplicity, broadcasting, device and service discovery, and pairing.

Bluetooth initially seemed like a good match for my programming goals – a chat application for ad-hoc groups. The first version would only support text messages, but I wanted to add more complex media (e.g. graphics, audio, video) later.

Bluetooth offers plenty of advantages for this kind of software, including letting devices communicate directly, without the need for Internet infrastructure. Bluetooth utilizes short-range radio, usually over the 10m range, without requiring line-of-sight contact. The protocol consumes a small amount of power (ideal for mobile devices), and is supported by many platforms.

That's the good news, now it's time to turn to the problems.

Multiplicity (of Stacks, Protocols, Profiles, and Versions)

A Bluetooth stack consists of multiple protocols and profiles; some mandatory but most optional, which allows different vendors' Bluetooth stacks to vary quite considerably. From a developer's point-of-view, the most important Bluetooth transport protocols are RFCOMM and L2CAP, which roughly correspond to stream-based TCP and packet-based UDP in the networking world. L2CAP is a mandatory protocol, which should make it's presence a safe bet on every manufacturer's stack. This isn't the case though: Microsoft XP's Bluetooth stack only offers RFCOMM (although L2CAP is employed in RFCOMM's implementation). It's the 'optional' RFCOMM protocol that's present on every stack, and Microsoft's decision makes L2CAP a poor communications choice for software portability.

It's possible to replace Microsoft's default stack by a more fully-featured one (e.g. the Widcomm/Broadcom stack), but the process is tricky, and Widcomm isn't free. There are many more stacks to choose from apart from Microsoft and Widcomm, including CSR, Toshiba, and BlueSoleil, each offering a different mix of protocols and profiles. Thankfully, the situation on Linux and the Mac is much simpler – BlueZ is the official stack for Linux, and part of the kernel (http://www.bluez.org/), while Mac OS X also comes with an integrated stack

Bluetooth's many protocols are mostly built on top of L2CAP or RFCOMM. For instance: AVDTP (Audio/Video Distribution Transport Protocol), the Telephony control protocol, TCP/IP/UDP, and OBEX (Object Exchange Protocol). Unfortunately, of these, JSR-82 implementations may offer only OBEX, but even that's optional. This lack was a deliberate decision, the hope being that the gap would be filled by third-party developers, which hasn't happened. In practice, a Java programmer is limited to RFCOMM if he wants his software to stand a good chance of working across different devices.

Let's not forget the other 'p' word – profiles. Bluetooth profiles support higher level tasks implemented on top of transport protocols, such as headset connectivity, file transfer, music management, and printer use. None of these are part of JSR-82, which offers four core profiles, the important ones being SPP, allowing RFCOMM to be treated as a serial port link, and SDAP for service discovery (which I'll talk about later).

The richness of Bluetooth – its many protocols and profiles – is out of reach to a Java programmer. The obvious response is to look for a Bluetooth API in another language, such as Python or C, ignoring that JSR-82 is the most widely supported API across devices. Possibilities include the Wireless Communication Library (http://www.btframework.com/bt.htm) and 32feet.NET (http://inthehand.com/content/32feet.aspx), but neither is much different from JSR-82, although 32feet.NET can utilize several OBEX-based profiles as an add-on library.

The most serious omission from JSR-82, in programming terms, is support for the Host Controller Interface (HCI) protocol. HCI specifies how the device (the host) interacts with the Bluetooth adapter/dongle (the controller). This interaction includes pairing, when two devices form a communications link, which I'll talk about below.

HCI is represented in the JSR-82 world by the BCC (the Bluetooth Control Center), which doesn't have an API. This isn't entirely JSR-82's fault, since most Bluetooth stacks don't offer a programming interface for the underlying HCI either. One important exception is Linux's BlueZ stack.

Aside from multiple stacks, and lack of profiles and protocols, there's the issue of Bluetooth versions. Bluetooth has been around since the mid 1990's, with the first decent version (v1.1) appearing in the early 2000's. Since then, we've had v1.2 (2004), v2.0 + EDR (2005), v2.1 + EDR (2007), v3.0 + HS (2009), and v4.0 (2010), which have improved data transfer rates, simplified device discovery and pairing, and added new protocols. But there's a problem: JSR-82 is based on v1.1, and hasn't been updated as the specification has evolved. Useful features like faster device discovery and simple pairing are unavailable to Java programmers.

Even if JSR-82 had kept up with the Bluetooth specification, adapter and dongle makers have not. For instance, most low-cost Bluetooth dongles offer v2.0 + EDR, which predates the simplified pairing in v2.1, which would be of great use to programmers. Notably, Bluetooth versioning impacts Microsoft Windows: Windows XP SP 2 and SP3 releases natively support Bluetooth 1.1, 2.0, and 2.0+EDR, while Windows Vista SP2 and Windows 7 offer Bluetooth 2.1+EDR

Broadcasting

Bluetooth is frequently promoted as suitable for peer-to-peer networks, since a Bluetooth device can be both a server and a client. This wasn't emphasized in the early days of the specification, when Bluetooth was seen more as a cable replacement for battery devices.

In fact, Bluetooth inter-device communication programming (after the discovery phase) is very similar to socket programming over a network. Bluetooth has no API-level support for dual client/server functionality. It you want a device to be both, then you'll need to write code to do it, using socket-like programming techniques. Bearing this in mind, it's not surprising that virtually every Bluetooth coding example is based

on the simpler client/server model. One of the network's device's becomes the designated server which client devices contact. Client-to-client communication is implemented by utilizing the server as a central store or clearinghouse for messages. To say that Bluetooth is a peer-to-peer communication framework is similar to arguing that sockets are a P2P mechanism, a misleading notion at best.

There's no scalable way in Bluetooth for a device to broadcast to other devices in a network There is a protocol for the master of a piconet to broadcast a message to all its slaves using L2CAP packets, but a piconet can only contain at most eight devices. There's also no way for a device to broadcast its presence to other devices, for example when a new service wants to announce itself to all the local devices.

Broadcasting is a part of Bluetooth, but only during device discovery.

Device Discovery

A device that wants to look for (i.e. discover) other devices must pass through three stages: inquiry, selection, and paging. The device starts by broadcasting requests for local device information over a 10-100m radius. The communication hops through multiple frequency channels to make these messages harder to intercept, and to avoid radio congestion. A typical Bluetooth device changes channels every 625 microseconds (1600 times per second).

Bluetooth v1.2, and later, utilize adaptive frequency hopping, which allow devices to avoid channels with high interference (e.g., one that overlaps a nearby 802.11 network).

Listening devices, that are ready to be contacted, will also hop through the channels in the hope of receiving an inquiry. Unfortunately, as long as a listener is tuned into a different channel, then devices will never meet. This has been likened to the *whack-a-mole* game, and is why two devices, only inches apart, may take a very long time to detect each other.

According to the specification (http://www.bluetooth.com/), this inquiring stage may take **10.24 seconds** or more (with 10.24 being an optimistic estimate in practice). Discovery times can lengthen depending on the amount of radio interference. Possible sources are other Bluetooth devices, WiFi (which uses the same part of the radio spectrum, the 2.4 GHz frequency band), microwaves, and even human bodies (because of their water content). Interference becomes more likely if WiFi and Bluetooth devices are close together, as they are in mobile devices. It's worth pointing out that most urban environments where users might consider using Bluetooth to create ad hoc chatting groups (my application domain, if you recall) are full of such noisy sources (e.g. office cubicles and coffee shops).

In a noisy environment, there’s no guarantee that an inquiry will succeed even if both devices happen upon the same frequency at the same time, since packets transmitted at that time may be corrupted.

One of Bluetooth’s advantages is it's ability to adjust its radio power usage. If two devices are very close together (within a few feet) then the Bluetooth signal can be sent at low power, and be less likely to suffer from interference.

All discoverable devices within a 10-100 meter broadcast range may respond to a device inquiry, thereby generating a very extensive device names list back on the

client. The selection stage typically involves the user in a manual browse of the list, adding further seconds to the discovery process. However, JSR-82 allows this to be automated, e.g. by having code search through the list for a suitable choice.

The third discovery stage is paging, where a communication link between the devices is established. The *whack-a-mole* game continues: the client device repeatedly sends out connection request messages as it channel hops through a sequence based on the chosen server's address and estimated clock time. The server will also be hopping, waiting for a connection request. The Bluetooth specification claims that paging may take **7.68 seconds** or longer. When a connection is established, the client and server will start hopping in unison.

Based on the Bluetooth specification's time estimates, the discovery phase can take about **20 seconds** at best. This will seem like an eternity to a user, and the actual wait time may be much longer. In my own tests, in a radio-noisy computer lab, device discovery only became approximately reliable when performed inside a loop that repeated the discovery search if it failed. A single failed search often took 40 seconds to complete, resulting in perhaps a wait of 100 seconds or more before a device was successfully located! Incredibly, the device in question was often less than a meter away from the searching client.

Lengthy discovery time can be an application-killer when the devices are moving. The discovery time may exceed the time when the two devices are in range of one another, effectively making them unable to communicate.

JSR-82 tries to improve matters by having two types of predefined devices: pre-known and cached. Pre-known devices are those that communicate frequently with the device, but it's not possible to install a list of these programmatically, since it requires access to the HCI. Cached device names are stored as a byproduct of previous inquiries.

Service Discovery

Twenty seconds have passed (if we're very lucky), and the devices are now connected, but discovery isn't over yet. The client must now select a service on the server by choosing a service channel identifier. In socket programming terms, the server address has been determined, but not a port number. Unfortunately, services channel IDs are assigned dynamically by Bluetooth, and so there's no reliable way to have a client contact a service directly. Instead the client must communicate with a Service Discovery Protocol (SDP) service on the server, in order to lookup the desired channel ID.

Faster Discovery?

Are there ways to speed up discovery? One pleasing solution is to ditch the entire device discovery mechanism, replacing it with infrared (IrDa): now the user brings his device into close proximity with the server to perform a fast, direct infrared link.

More recently, Bruce Hopkins has suggested a similar idea using JSR-257, the Contactless Communication API. JSR-257 relies upon device-to-device Near-Field Communication (NFC) to create a direct link, employing smart cards, RFIDs, or barcodes (http://java.sun.com/developer/technicalArticles/javame/nfc_bluetooth/). He

reported impressive speed-ups – a reduction in running time of his test application from 90 to 11 seconds.

A drawback with Hopkin's approach is that JSR-257 and NFC aren't present on many devices as yet. There's an easy-ish solution, which is to implement your own version, using a webcam and barcode libraries. I'll explain how to do this in a later chapter.

Direct device linkage is an excellent way of avoiding 20+ second device discovery, and the Bluetooth designers have responded. Version 2.1 features Near Field Pairing which automatically pairs two devices when they are held close together, the problem being that most devices don't support v2.1.

Near Field Pairing doesn’t impact service discovery, which still requires a SDP service on the server. Since a service's channel ID is assigned dynamically at run time, there’s no reliable way to store its value ahead of time inside a smart card, barcode or RFID chip.

Pairing

A device’s services typically expose it to change by its clients, so those clients must pass some security checks beforehand. These checks begin with pairing, which is triggered when a device receives a connection request from an unknown client. The client sends a PIN number (also called a passkey), which is checked on the server-side. If the number is correct then the service is made available. After pairing is completed, Bluetooth may carry out additional security measures, including authentication, encryption, and authorization.

Pairing adds some problems to the client/server programming model. The user must enter a PIN on the client-side, and the server must check it. Pairing is a security feature, so is handled by the HCI. As I mentioned earlier, JSR-82 doesn't have a programming interface to HCI, so there's no way to automate these two tasks from within Java.

This isn't too bad for the client because the device only displays a dialog box, which most users can deal with (if they can remember the PIN number). The problems are more serious on the server since a dialog box appears there whenever a client wants to connect. This means that a fully automated server application cannot be written in JSR-82 because a person is needed to respond to the dialog boxes.

There are solutions, but they employ non-standard features offered by some stacks and JSR-82 extensions. On the client-side, the BlueCove implementation of JSR-82 (http://bluecove.org/) offers a RemoteDeviceHelper.authenticate() method for sending a PIN number to a server, so the user isn't bothered by a dialog box. On the server-side, it's necessary to use the BlueZ stack on Linux which can create an 'agent' process to accept all pairing requests. It's also possible in the Widcomm stack to configure the server to accept predefined clients without requesting a PIN number.

These problems are a little less severe in Bluetooth v2.1. It introduced Simple Pairing, which lets the client device generate a PIN rather than requiring the user to type one. Depending on the device type, a dialog box may not appear at all.

Summary and Recommendations

Programming client/server applications in Bluetooth is a tricky process due to the wide variety of stacks, protocols, profiles, and versions. Bluetooth suffers from long discovery times, which become worse in the presence of radio interference (which is common). JSR-82's lack of access to the HCI makes it impossible to write a server application which can be fully automated.

The solution isn't to ditch JSR-82, which still offers the best chance of writing portable Bluetooth code. Instead, it's necessary to make a careful choice of stacks and protocol. The server side should utilize BlueZ on Linux (http://www.bluez.org/) so it's HCI agent capabilities can automate the server's part of the pairing process. The communication protocol should be RFCOMM, to maximize the choice of client-side stacks. My client coding is aimed at laptops and netbooks, which makes the BlueCove implementation of JSR-82 a good choice (http://bluecove.org/), and it includes extensions for performing automatic pairing from the client-side.

The treacle-like speeds of Bluetooth device discovery can be replaced by direct device discovery using the client's Webcam to read a barcode attached to the side of the server device. Admittedly, it's a bit low-rent, but works across multiple platforms, until JSR-257 and NFC become more commonplace.