Chapter 1

Introduction

Bluetooth is *a way for devices to wirelessly communicate over short distances*. Wireless communication has been around since the late nineteenth century, and has taken form in radio, infrared, television, and more recently 802.11. What distinguishes Bluetooth is its special attention to short-distance communication, usually less than 30 ft. Both hardware and software are affected by this special attention.

A comprehensive set of documents, called the *Bluetooth Specifications*,* describes, in gory detail, everything from the basic radio signal to the high-level protocols for this wireless, short-range communication. Our goal is to help you master the essentials and then with a pat on the back and a firm handshake, let you wade through the gory details on your own, confident that you are well prepared. In other words, our goal is to help the reader get started.

1.1 Understanding Bluetooth as a Software Developer

There is nothing especially difficult about Bluetooth, although the unusually wide scope of Bluetooth makes it hard to get started. There is so much in the Bluetooth specification and at so many different levels, it is hard to distinguish the essentials from the elective. Technology specifications, especially ones

* http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications/

that are given folksy names, often refer to something very specific and with a narrow scope. For example, there is no single specification of the Internet. Rather each of the components has its own specification. Ethernet, for example, describes how to connect a bunch of machines together to form a simple network, but that's about it. TCP/IP describes two specific communication protocols that form the basis of the Internet, but they're just two protocols. Similarly, HTTP is the basis behind the World Wide Web, but also boils down to a simple protocol. The Internet is a collection of these and other pieces. A software developer first learning about Internet programming – how to connect one computer on the Internet to another and send data back and forth – will not initially bother with the details of Ethernet or e-mail, precisely because neither technology is central. Sure, e-mail is a wonderful example of Internet programming and Ethernet gives context on how the connections are implemented, but TCP/IP programming is the essence.

In many ways, the word Bluetooth is like the word Internet because it encompasses a wide range of subjects. Similar to USB, Ethernet, and 802.11, Bluetooth defines a lot of physical on-the-wire stuff, such as the radio frequencies and how to modulate and demodulate signals. Similar to Voice-over-IP protocols used in many Internet applications, Bluetooth also describes how to transmit audio between devices. But Bluetooth also specifies everything in between! It's no wonder that the Bluetooth specifications are thousands upon thousands of pages.

This text answers the question: How do we create programs that connect two Bluetooth devices and transfer data between them? We introduce the essential concepts, as well as application design considerations. Later chapters show how to actually do this with a number of popular programming languages and operating systems. Most importantly, we keep it simple but sufficient.

1.2 Essential Bluetooth Programming Concepts

The essentials of Bluetooth programming are neither numerous nor difficult. Throughout the rest of this chapter, they will be compared to those of Internet programming. Although Bluetooth was designed from the ground up, presumably independent of the Ethernet and TCP/IP protocols, it is quite reasonable to think of Bluetooth programming in the same way as Internet programming. Both fall under the general rubric of network programming, and share the same principles of one device communicating and exchanging data with another device. Bluetooth and Internet programming share so much in common that understanding one makes it much easier to understand the other.

TCP/IP programming is mature, ubiquitous, has plenty of examples, and its comparison with Bluetooth strengthens the reader's understanding of both topics. The biggest difference, as mentioned earlier, is that Bluetooth focuses on physically proximate devices, whereas Internet programming doesn't care about distance at all. This difference will greatly affect how two devices initially find each other and establish an initial connection. After that, everything is pretty much the same.

The actual process of establishing a connection depends on whether the device in question is establishing an *outgoing* or an *incoming* connection. Roughly, this is the difference between which device sends the first data packet to initiate communications and which device receives that packet. We'll often refer to these as the *client* and *server*, respectively.

Note: We use the words *client* and *server* only to distinguish between which device initiates a connection, and without implying any relationship to the Client–Server model of network programming. Despite the overlap, it is perfectly reasonable for a server (the way we use the word) to function as a client (in the Client–Server model sense), and vice versa.

Devices initiating an outgoing connection need to choose a target device and a transport protocol, before establishing a connection and transferring data. Devices establishing an incoming connection need to choose a transport protocol, and then listen before accepting a connection and transferring data. Figures 1.1 and 1.2 illustrate these basic concepts and how they translate to both Internet and Bluetooth programming. Notice that for outgoing connections, only the first two steps (choosing a target device, transport protocol, and port number) are different for Bluetooth and Internet programming. Once the outgoing connection is established, everything is pretty much the same. The processes for accepting an incoming connection are even more similar, with the main difference being the added support of Bluetooth for dynamically assigned port numbers.

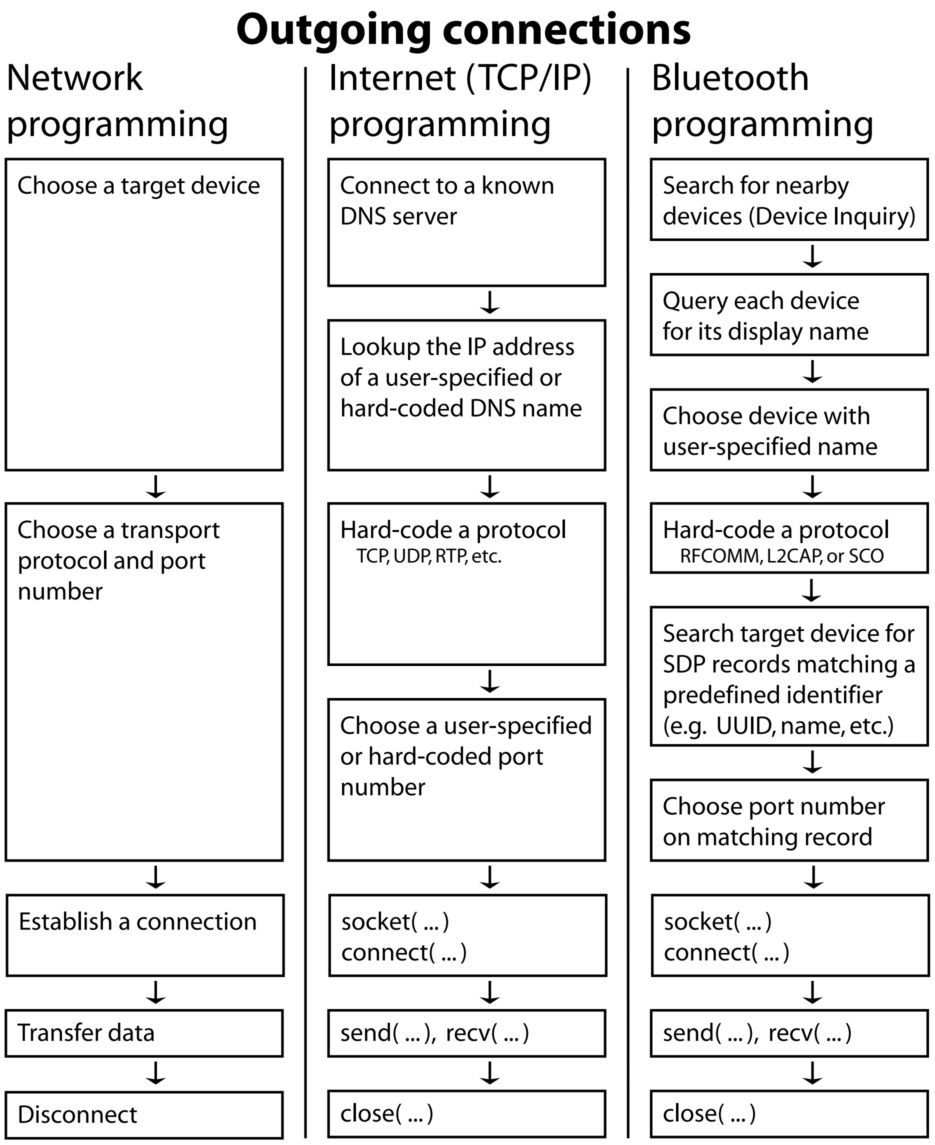
Outgoing connections
Network programming
Choose a target device
Choose a transport protocol and port number
Establish a connection
Transfer data
Disconnect
Internet (TCP/IP) programming
Connect to a known DNS server
Lookup the IP address of a user-specified or hard-coded DNS name
Hard-code a protocol TCP, UDP, RTP, etc.
Choose a user-specified or hard-coded port number
socket(...) connect(...)
send(...), recv(...)
close(...)
Bluetooth programming
Search for nearby devices (Device Inquiry)
Query each device for its display name
Choose device with user-specified name
Hard-code a protocol RFCOMM, L2CAP, or SCO
Search target device for SDP records matching a predefined identifier (e.g. UUID, name, etc.)
Choose port number on matching record
socket(...) connect(...)
send(...), recv(...)
close(...)

Figure 1.1 There are five major conecptual steps for programming an outgoing connection. Only the initial process of choosing a target device, transport protocol, and port number differs significantly from Internet to Bluetooth programming.

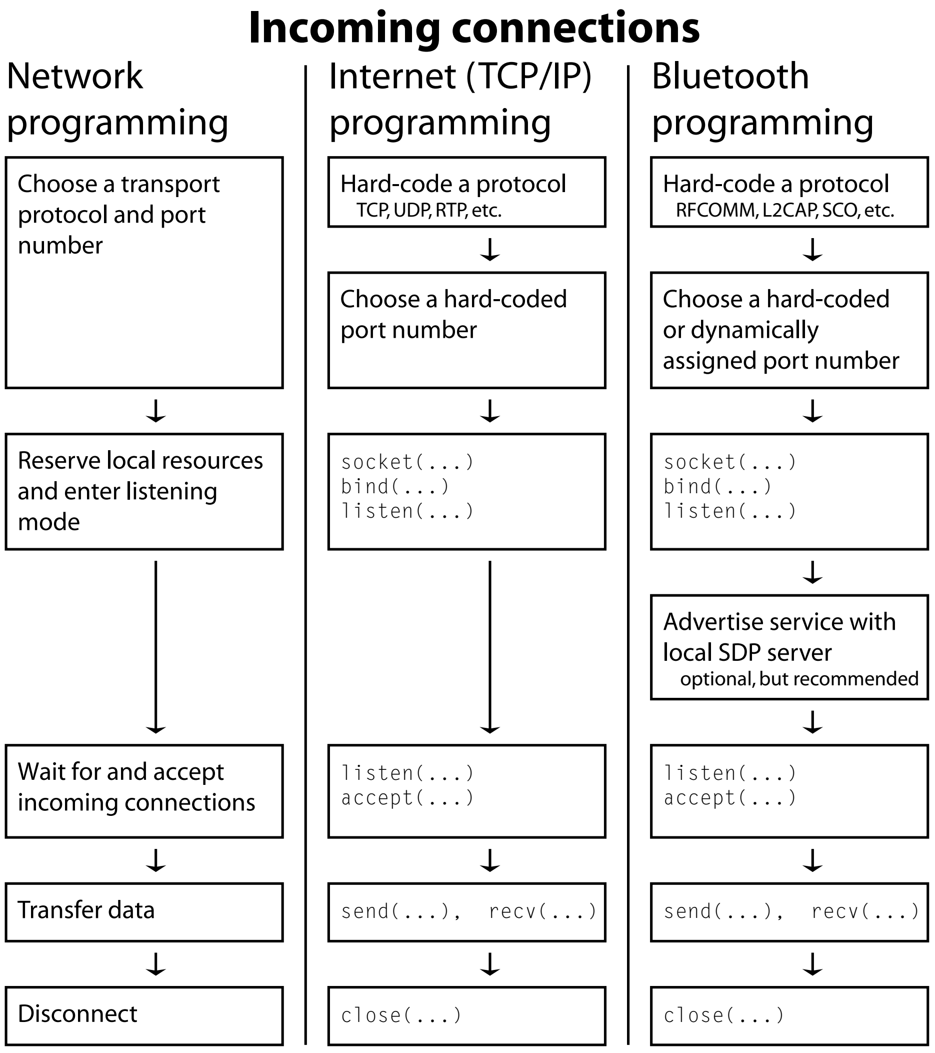
Incoming connections
Network programming
Internet (TCP/IP) programming
Bluetooth programming
Choose a transport protocol and port number
Hard-code a protocol TCP, UDP, RTP, etc.
Hard-code a protocol RFCOMM, L2CAP, SCO, etc.
Choose a hard-coded port number
Choose a hard-coded or dynamically assigned port number
Reserve local resources and enter listening mode
socket(...) bind(...) listen(...)
socket(...) bind(...) listen(...)
Advertise service with local SDP server optional, but recommended
Wait for and accept incoming connections
listen(...) accept(...)
listen(...) accept(...)
Transfer data
send(...), recv(...)
send(...), recv(...)
Disconnect
close(...)
close(...)

Figure 1.2 There are also five major conecptual steps for programming an incoming connection. As with outgoing connections, the details of Internet and Bluetooth programming are related, although slightly different.

The seasoned network programmer will notice that some of the steps illustrated don't always apply, and don't necessarily have to be completed in the order shown. For example, if the address of a server is hard-coded into a client program, then there's no need to use the Domain Name System (DNS) or a device inquiry. The key here is that these diagrams show the "vanilla" way of doing things, which can be adapted and tweaked to serve the needs of each individual application. As we go through each concept, in the next several sections, we mention each of these deviations in turn. Subsequent chapters elaborate on how these concepts are implemented across various platforms and programming languages.

1.2.1 Choosing a Target Device

Every Bluetooth chip ever manufactured is imprinted with a globally unique 48-bit address, referred to as the *Bluetooth address* or *device address*. This is identical in nature to the Machine Address Code (MAC) address for Ethernet.* In fact, both address spaces are managed by the same organization – the IEEE Registration Authority. These Bluetooth addresses, assigned at manufacture time, are intended to be unique and remain static for the lifetime of the chip. It conveniently serves as the basic addressing unit in all of Bluetooth programming.

For one Bluetooth device to communicate with another, it must have some way of determining the other device's Bluetooth address. The address is used in all layers of the Bluetooth communication process, from the low-level radio protocols to the higher level application protocols. In contrast, TCP/IP network devices that use Ethernet discard the 48-bit MAC address at higher layers of the communication process and switch to using IP addresses. The principle remains the same however, in that the unique identifying address of the target device must be known in order to communicate with it.

The client program may not have advance knowledge of these target addresses. In Internet programming, the user typically knows or supplies a host name, such as `www.kernel.org`, which must then be translated to a physical IP address by DNS. In Bluetooth, the user will typically supply some user-friendly name, such as "My Phone," and the client translates this to a numerical address by searching nearby Bluetooth devices and checking the name of each device.

* `http://www.ietf.org/rfc/rfc0826.txt`

Note: Throughout the book, we will use the words *adapter* and *device*, with slightly different meanings. In general, a Bluetooth *device* refers to any machine capable of Bluetooth communication. When we're talking about writing programs, *adapter* refers specifically to the Bluetooth device on the local computer (the one that's running the program). This is to reduce confusion and help specify exactly which device is in question.

Device Name

Humans do not deal well with 48-bit numbers, such as `0x000EED3D1829` (in much the same way, we do not deal well with numerical IP addresses like 68.15.34.115), and so Bluetooth devices will almost always have a *user-friendly name* (also called the *display name*). This name is usually shown to the user in lieu of the Bluetooth address to identify a device, but ultimately it is the Bluetooth address that is used in actual communication. For many machines, such as mobile cell phones and desktop computers, this name is configurable and the user can choose an arbitrary word or phrase. There is no requirement for the user to choose a unique name, which can sometimes cause confusion when many nearby devices have the same name. When sending a file to someone's phone, for example, the user may be faced with the task of choosing from five different phones, each of which is named "My Phone."

Names in Bluetooth are similar to DNS names in that both are human-friendly identifiers that eventually get translated to machine-friendly identifiers. They differ in that DNS names are unique (there can only be one `www.google.com`) and registered with a central database, whereas Bluetooth names are more or less arbitrary, frequently duplicated, and are registered only on the device itself. In TCP/IP, one begins with a DNS name. Translating a DNS name to an IP address involves contacting a nameserver, issuing a query, and waiting for a result. In Bluetooth, the lookup process is reversed. First, a device searches for nearby Bluetooth devices and compiles a list of their addresses. Then, it contacts each nearby device individually and queries it for its user-friendly name.

Searching for Nearby Devices

Device discovery, also known as device inquiry, is the process of searching for and detecting nearby Bluetooth devices. It is simple in theory: To figure out what's nearby, broadcast a "discovery" message and wait for replies. Each reply consists of the address of the responding device and an integer identifying the general class of the device (e.g., cell phone, desktop PC, headset, etc.). More detailed information, such as the name of a device, can then be obtained by contacting each device individually.*

In practice, this is often a confusing and irritating subject for Bluetooth developers and users. The source of this aggravation stems from the fact that it can take a long time to detect nearby Bluetooth devices. To be specific, given a Bluetooth cell phone and a Bluetooth laptop sitting next to each other on a desk, it will usually take an average of 5 seconds before the phone detects the presence of the laptop, and it sometimes can take upward of 10–15 seconds. This might not seem like that much time, but put in context it is suprising. During the device discovery process, the phone is changing frequencies more than a thousand times a second, and there are only 79 possible frequencies on which it can transmit. It is a wonder why they don't find each other in the blink of an eye.

The technical reasons for this are mostly due to the result of a strangely designed search algorithm, explained in greater detail in Section 1.3.9. Newer versions of Bluetooth (starting in Bluetooth 1.2) attempt to reduce the average search time, but don't expect any miracles in the near future. Suffice to say, device discovery takes too long.

Note: A common misconception is that when a Bluetooth device enters an area, it somehow "announces" its presence so that other devices will know that it's around. This never happens (even though it's not a bad idea), and the *only* way for one device to detect the presence of another is to conduct a device discovery.

Discoverability and Connectability

For privacy and power concerns, all Bluetooth devices have two options that determine whether or not the device responds to device inquiries and connection attempts. The *Inquiry Scan* option controls the former, and the

* Bluetooth 2.1 introduces the *Extended Inquiry Response*, where the most commonly requested information, such as the name of a responding device and a summary of the services it offers, can be transmitted directly in the inquiry response, saving some time.

Table 1.1 Inquiry Scan and Page Scan.

Inquiry Scan	Page Scan	Interpretation
On	On	The local device is detectable by other Bluetooth devices, and will accept incoming connection requests. This is often the default setting.
Off	On	Although not detectable by other Bluetooth devices, the local device still responds to connection requests by devices that already have its Bluetooth address. This is often the default setting.
On	Off	The local device is detectable by other Bluetooth devices, but it will not accept any incoming connections. This is mostly useless.
Off	Off	The local device is not detectable by other Bluetooth devices, and will not accept any incoming connections. This could be useful if the local device will only establish outgoing connections.

Page Scan option controls the latter. Both are described in Table 1.1, and are configurable to some degree on most devices.

Although the names *inquiry scan* and *page scan* are confusing, it is important not to confuse these two terms with the actual processes of detecting and connecting to other devices. The reasoning behind these names is that these options control whether the local device *scans* for device inquiries and connection attempts. A device is in discoverable mode when inquiry scan is on.

1.2.2 Choosing a Transport Protocol

Different applications have different needs, hence the reason for different transport protocols. This section describes the ones you should know about for Bluetooth programming, and why your application might use them.

The two factors that distinguish the protocols here are *guarantees* and *semantics*. The guarantees of a protocol state how hard it tries to deliver a packet sent by the application. A *reliable* protocol, like TCP, takes the “deliver-or-die” mentality; it ensures that all sent packets get delivered, or dies trying (terminates the connection). A *best-effort* protocol, like UDP, makes a reasonable attempt at delivering transmitted packets, but ignores the case when the packets do not arrive, say, due to signal noise, a crashed router, an act of god, and so on.

The semantics of a protocol can be either packet-based or streams-based. A packet-based protocol like UDP sends data in individual datagrams of fixed maximum length. A streams-based protcol, like TCP, on the other hand, just sends data without worrying about where one packet ends and the next one starts. This choice is more a matter of preference than any specific requirements, because with a little arm-twisting, a packet-based protocol can be used like a streams-based protocol, and vice versa.

Bluetooth contains many transport protocols, nearly all of which are special purpose. We consider four protocols to be essential, although only two, RFCOMM and L2CAP, are likely to be used to get started. And of these two, only the first, RFCOMM, will be relevant for many readers. The protocols are presented in the order of their relevance; the anxious reader can skim over the latter two or three.

RFCOMM

The Radio Frequency Communications (RFCOMM) protocol is a reliable streams-based protocol. It provides roughly the same service and reliability guarantees as TCP. The Bluetooth specification states that it was designed to emulate RS-232 serial ports (to make it easier for manufacturers to add Bluetooth capabilities to their existing serial port devices), but we prefer to turn that definition around and say that *RFCOMM is a general-purpose transport protocol that happens to work well for emulating serial ports*.

The choice of port numbers is the biggest difference between TCP and RFCOMM from a network programmer's perspective. Whereas TCP supports up to 65,535 open ports on a single machine, RFCOMM allows only 30. This has a significant impact on how to choose port numbers for server applications.

A distinguishing attribute of RFCOMM is that, depending on the application and the target platform, sometimes it is the only choice. Some environments, such as the Microsoft Windows XP Bluetooth API and Nokia Series 60 Python, support only the RFCOMM transport protocol. This really isn't all that bad because it's a robust general-purpose protocol, but it is something worth keeping in mind if you were to consider the other protocols mentioned. For more information, see Section 1.3.10.