The Use of Bluetooth in Linux and Location Aware Computing

by

Albert Huang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

Author .....
Department of Electrical Engineering and Computer Science
May 1, 2005

Certified by .....
Larry Rudolph
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Use of Bluetooth in Linux and Location Aware Computing

by

Albert Huang

Submitted to the Department of Electrical Engineering and Computer Science
on May 1, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The Bluetooth specification describes a robust and powerful technology for short-range wireless communication. Unfortunately, the specification is immense and complicated, presenting a formidable challenge for novice developers. Currently, there is a lack of satisfactory technical documentation describng Bluetooth application development and the parts of the Bluetooth specification that are relevant to software developers. This thesis explains Bluetooth programming in the context of Internet programming and shows how most concepts in Internet programming are easily translated to Bluetooth. It describes how these concepts can be implemented in the GNU/Linux operating system using the BlueZ Bluetooth protocol stack and libraries. A Python extension module is presented that was created to assist in rapid development and deployment of Bluetooth applications. Finally, an inexpensive and trivially deployed infrastructure for location aware computing is presented, with a series of experiments conducted to determine how best to exploit such an infrastructure.

Thesis Supervisor: Larry Rudolph
Title: Principal Research Scientist

Acknowledgments

Many thanks to Larry for his invaluable comments, advice, and endless patience. This thesis would not have been possible if it weren't for him. Thanks also to my family (Mom, Dad, Heather), to Stacy, and to all my friends for being so supportive. Finally, thanks go to the ORG team, the folks at Nokia, and the folks at Acer for all their help, with special thanks to Calvin for providing the initial implementation of PyBluez.

Contents

Chapter 1

Introduction

This thesis argues that Bluetooth is a suitable technology for indoor location aware computing, and that it is much simpler to use and program than has previously been believed. We support this claim by constructing a system for location aware computing, evaluating it, and explaining how to program the system and Bluetooth devices using concepts familiar to network programmers. This thesis does not attempt to explain Bluetooth in detail, as such documents already exist[23]. Rather, it approaches Bluetooth from a software developer's perspective, and describes only the portions of the specification that are relevant to typical network programming tasks.

1.1 Motivation

Fourteen years ago, Mark Weiser laid the foundation for ubiquitous computing in a seminal paper[32] describing his vision of the computer in years to come. In order for computers to truly have a positive impact on human society, he wrote, they must fade into society. They must assist and improve our lives without demanding our constant care and attention. Technologies such as modern plumbing, electricity, the telephone system, are all highly structured, complex, and requires years of study to fully understand and master; yet the average person takes advantage of them every day without even acknowledging their presence. They have faded into the background of society, improving our lives without demanding our attention. This is the goal to

which ubiquitous computing strives.

Imagine, for example, that Stacy is on her way home from work, and stops by the grocery store. Earlier in the day, Stacy had run out of something but now she can't remember what, only that she wanted to pick something up at the grocery. Fortunately, she glances down at her cell phone, which is telling her to buy some orange juice. A few hours ago, Stacy told it to remind her the next time she stopped by the grocery store to pick up some orange juice. Now, upon detecting that she's there, her cell phone alerts her.

On her way home, Stacy's car flashes a warning and tells her that she shouldn't take her normal route because an accident has stopped all traffic on the highway. An on-board computer receives real-time traffic updates and uses them in conjunction with an internal road map and the current position of the car to calculate a recommended path. Once Stacy responds and asks her car to show her the best way home, it begins guiding her on an alternate route that avoids the congestion.

Later on, Stacy decides to go to the movies with her friends and catch the latest showing. As they settle into their seats and watch the previews, none of them notice that there are no reminders telling them to turn off their cell phones and pagers. Nobody notices, because they don't need to. Instead, their cell phones have already detected that they've entered a movie theater and shouldn't ring out loud in the middle of the theater. As they shuffle out of the theater and back into their cars, their cell phones automatically switch back to their normal modes of operation.

We have chosen these examples to emphasize the focus of our research and to provide a plausible description of how computers may assist us in the near future. In two of the examples, we chose the cell phone to represent the personal computational device that almost everyone carries around, the device that assists and improves our lives while requiring hardly any maintenance at all. We made this choice because the cell phone is already commonplace. Indeed, a recent report[24] indicates that in Taiwan, there are more active mobile phone subscriptions than there are people. This is not an isolated trend, and cell phone usage has grown by staggering proportions all around the world in recent years. It seems reasonable that with the growing power

and falling cost of inexpensive computational components, cell phones could easily become a central platform in ubiquitous computing.

1.2 Location Aware Computing

In all three examples, location-awareness is a crucial component needed for the system to work properly. When we say that a computational device is *location aware*, we mean that it has some notion of its physical location. The form in which the location is stored can vary from precise latitude-longitude coordinates to broad conceptual terms such as *home, school, the office*. Once a computer becomes location aware, it can then perform different actions depending on its location. It can, as illustrated above, remind users to actively do something based on where they are. It could also take action without informing the user, such as switching to silent mode when entering a meeting, classroom, or movie theater. It could serve as a navigation aid, instructing the user where to go. There doesn't even need to be a user nearby for device to be location aware. A maintenance robot could perform routine tasks such as sweeping the kitchen floor or taking out the trash if it knew where to do the right things.

Many people have already had firsthand experience with location aware computing. The Global Positioning System (GPS)[12] currently serves as the backbone for many location aware systems. High end vehicles using GPS that can already perform the tasks mentioned in the second example are available to consumers today. Having cemented its usefulness in society, GPS powered applications range from helping navigationally challenged sailors and hikers find their way home to instantly giving police the exact coordinates of where a 911 emergency call was placed.

Having identified location awareness as a critical ability needed for an effective ubiquitous computing system, and having seen the proliferation of GPS and its effectiveness in providing these services, we might decide that this particular problem has been solved and move on. Examining it a little more closely, however, reveals that there are still many challenges that we face in location aware computing. GPS

currently does not work well indoors or in crowded city environments, where ceilings, walls, and buildings may obstruct the satellite signals needed to obtain a precise location estimate. As a centralized infrastructure tightly controlled by a single government, GPS is only reliable as the weakest point in the system. In the event of satellite failure, all GPS receivers would suffer. Finally, GPS can only provide accurate position estimates to within a few meters, which is not sufficient for all purposes.

Although reliability is a concern, it is usually the case that the more important a system becomes, the more carefully it will be maintained and serviced. The primary disadvantages of GPS then become its limited precision and its failure to work in areas without clear satellite reception, which unfortunately happens to be where a vast portion of the human population spends its time. In order for a computational device to be location aware in such places, it must utilize some other technology.

1.3 Bluetooth

Bluetooth [8, 9] was originally designed to provide power-efficient, low-cost short range radio communications, and has matured into a powerful technology widely available today. According to one market research study, 69 million Bluetooth chips were shipped in 2003, approximately double that of the previous year, and are forecasted to grow to 720 million units by 2008[16]. Laptops, cell phones, and PDAs are increasingly shipped with an integrated Bluetooth radio.

Bluetooth, however, has largely remained confined to large industry, in part due to its complexity and the lack of clear documentation. The specifications and documents that are available are generally complicated and convey much more information than is needed for a single purpose. Most of the literature attempts to explain Bluetooth in its entirety, and not simply the parts of Bluetooth that are relevant to a specific task. Specifically, software developers accustomed to network programming methods who wish to apply their skills to Bluetooth programming are faced with both an overwhelming quantity of information about Bluetooth in general as well as a dearth of concise documentation explaining how their techniques can be easily ap-

plied. Without the resources and expertise available to large industrial organizations, novice developers are often turned away from Bluetooth.

This thesis makes four contributions. First, it explains Bluetooth programming in the context of network programming and shows how techniques and methods typically used in the latter easily translate to the former. Second, it presents a Python extension module that provides a simple and intuitive API for Bluetooth programming in the GNU/Linux operating system. Third, it presents an introduction to Bluetooth programming using the BlueZ API in the GNU/Linux operating system. Fourth, it argues that Bluetooth is a technology well suited for creating an infrastructure for location aware computing. This claim is supported by a description and evaluation of such a system built and deployed using Bluetooth technologies.

Chapter 2

Bluetooth programming

This chapter presents an overview of Bluetooth, with a special emphasis on the parts that concern a software developer. Bluetooth programming is explained in the context of TCP/IP and Internet programming, as the vast majority of network programmers are already familiar and comfortable with this framework. It is our belief that the concepts and terminology introduced by Bluetooth are easier to understand when presented side by side with the Internet programming concepts most similar to Bluetooth.

2.1 Overview

Hovering at over 1,000 pages[8, 9], the Bluetooth specification is quite large and daunting to the novice developer. But upon closer inspection, it turns out that although the specification is immense, the typical application programmer is only concerned with a small fraction of it. A significant part of the specification is dedicated to lower level tasks such as specifying the different radio frequencies to transmit on, the timing and signalling protocols, and the intricacies needed to establish communication. Interspersed with all of this are the portions that are relevant to the application developer. Although Bluetooth has its own terminology and ways of describing its various concepts, we find that explaining Bluetooth programming in the context of Internet programming is easy to understand since almost all network programmers

are already familiar with those techniques. We do not try to explain in great detail the actual Bluetooth specification, except where it aids in understanding how it fits in with Internet programming.

Although Bluetooth was designed from the ground up, independently of the Ethernet and TCP/IP protocols, it is quite reasonable to think of Bluetooth programming in the same way as Internet programming. Fundamentally, they have the same principles of one device communicating and exchanging data with another device. The different parts of network programming can be separated into several components

- Choosing a device with which to communicate
- Figuring out how to communicate with it
- Making an outgoing connection
- Accepting an incoming connection
- Sending data
- Receiving data

Some of these components do not apply to all models of network programming. In a connectionless model, for example, there is no notion of establishing a connection. Some parts can be trivial in certain scenarios and quite complex in another. If the numerical IP address of a server is hard-coded into a client program, for example, then choosing a device is no choice at all. In other cases, the program may need to consult numerous lookup tables and perform several queries before it knows its final communication endpoint.

2.2 Choosing a communication partner

Every Bluetooth chip ever manufactured is imprinted with a globally unique 48-bit address, which we will refer to as the *Bluetooth address* or *device address*. This is identical in nature to the MAC addresses of Ethernet[26], and both address spaces

are actually managed by the same organization - the IEEE Registration Authority. These addresses are assigned at manufacture time and 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. This address is used at 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 as their data link layer 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 to communicate with it.

In both cases, the client program will often not have advance knowledge of these target addresses. In Internet programming, the user will typically supply a host name, such as `csail.mit.edu`, which the client must translate to a physical IP address using the Domain Name System (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.

2.2.1 Device Name

Since humans do not deal well with 48-bit numbers like `0x000EED3D1829` (in much the same way we do not deal well with numerical IP addresses like 64.233.161.104), Bluetooth devices will almost always have a user-friendly 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 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 5 different phones, each of which is named

"My Phone".

Although names in Bluetooth differ from Internet names in that there is no central naming authority and names can sometimes be the same, the client program still has to translate from the user-friendly names presented by the user to the underlying numerical addresses. In TCP/IP, this involves contacting a local nameserver, issuing a query, and waiting for a result. In Bluetooth, where there are no nameservers, a client will instead broadcast inquiries to see what other devices are nearby and query each detected device for its user-friendly name. The client then chooses whichever device has a name that matches the one supplied by the user.

2.3 Choosing a transport protocol

Once our client application has determined the address of the host machine it wants to connect to, it must determine which transport protocol to use. This section describes the Bluetooth transport protocols closest in nature to the most commonly used Internet protocols. Consideration is also given to how the programmer might choose which protocol to use based on the application requirements.

Both Bluetooth and Internet programming involve using numerous different transport protocols, some of which are stacked on top of others. In TCP/IP, many applications use either TCP or UDP, both of which rely on IP as an underlying transport. TCP provides a connection-oriented method of reliably sending data in streams, and UDP provides a thin wrapper around IP that unreliably sends individual datagrams of fixed maximum length. There are also protocols like RTP for applications such as voice and video communications that have strict timing and latency requirements.

While Bluetooth does not have exactly equivalent protocols, it does provide protocols which can often be used in the same contexts as some of the Internet protocols.

2.3.1 RFCOMM + TCP

The RFCOMM protocol provides roughly the same service and reliability guarantees as TCP. Although the specification explicitly 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), it is quite simple to use it in many of the same scenarios as TCP.

In general, applications that use TCP are concerned with having a point-to-point connection over which they can reliably exchange streams of data. If a portion of that data cannot be delivered within a fixed time limit, then the connection is terminated and an error is delivered. Along with its various serial port emulation properties that, for the most part, do not concern network programmers, RFCOMM provides the same major attributes of TCP.

The biggest difference between TCP and RFCOMM from a network programmer's perspective is the choice of port number. Whereas TCP supports up to 65535 open ports on a single machine, RFCOMM only allows for 30. This has a significant impact on how to choose port numbers for server applications, and is discussed shortly.

2.3.2 L2CAP + UDP

UDP is often used in situations where reliable delivery of every packet is not crucial, and sometimes to avoid the additional overhead incurred by TCP. Specifically, UDP is chosen for its best-effort, simple datagram semantics. These are the same criteria that L2CAP satisfies as a communications protocol.

L2CAP, by default, provides a connection-oriented[1] protocol that reliably sends individual datagrams of fixed maximum length[2]. Being fairly customizable, L2CAP can be configured for varying levels of reliability. To provide this service, the transport protocol that L2CAP is built on[3] employs an transmit/acknowledgement scheme, where unacknowledged packets are retransmitted. There are three policies an application can use:

- never retransmit
- retransmit until total connection failure (the default)

[1]The L2CAP specification actually allows for both connectionless and connection-based channels, but a connectionless channels are rarely used in practice. Since sending "connectionless" data to a device requires joining its piconet, a time consuming process that is merely establishing a connection at a lower level, connectionless L2CAP channels afford no advantages over connection-oriented channels.

[2]The default maximum length is 672 bytes, but this can be negotiated up to 65535 bytes

[3]Asynchronous Connection-oriented logical transport

- drop a packet and move on to queued data if a packet hasn't been acknowledged after a specified time limit (0-1279 milliseconds). This is useful when data must be transmitted in a timely manner.

Although Bluetooth does allow the application to use best-effort communication instead of reliable communication, several caveats are in order. The reason for this is that adjusting the delivery semantics for a single L2CAP connection to another device affects *all* L2CAP connections to that device. If a program adjusts the delivery semantics for an L2CAP connection to another device, it should take care to ensure that there are no other L2CAP connections to that device. Additionally, since RFCOMM uses L2CAP as a transport, all RFCOMM connections to that device are also affected. While this is not a problem if only one Bluetooth connection to that device is expected, it is possible to adversely affect other Bluetooth applications that also have open connections.

The limitations on relaxing the delivery semantics for L2CAP aside, it serves as a suitable transport protocol when the application doesn't need the overhead and streams-based nature of RFCOMM, and can be used in many of the same situations that UDP is used in.

Given this suite of protocols and different ways of having one device communicate with another, an application developer is faced with the choice of choosing which one to use. In doing so, we will typically consider the delivery reliability required and the manner in which the data is to be sent. As shown above and illustrated in Table 2.3.2, we will usually choose RFCOMM in situations where we would choose TCP, and L2CAP when we would choose UDP.

Requirement	Internet	Bluetooth
Reliable, streams-based	TCP	RFCOMM
Reliable, datagram	TCP	RFCOMM or L2CAP with infinite retransmit
Best-effort, datagram	UDP	L2CAP (0-1279 ms retransmit)

Table 2.1: A comparison of the requirements that would lead us to choose certain protocols. Best-effort streams communication is not shown because it reduces to best-effort datagram communication.

2.4 Port numbers and the Service Discovery Protocol

The second part of figuring out how to communicate with a remote machine, once a numerical address and transport protocol are known, is to choose the port number. Almost all Internet transport protocols in common usage are designed with the notion of port numbers, so that multiple applications on the same host may simultaneously utilize a transport protocol. Bluetooth is no exception, but uses slightly different terminology. In L2CAP, ports are called *Protocol Service Multiplexers*, and can take on odd-numbered values between 1 and 32767. In RFCOMM, *channels* 1-30 are available for use. These differences aside, both protocol service multiplexers and channels serve the exact same purpose that ports do in TCP/IP. L2CAP, unlike RFCOMM, has a range of reserved port numbers (1-1023) that are not to be used for custom applications and protocols. This information is summarized in table 2.4. Through the rest of this thesis, the word *port* is use in place of protocol service multiplexer and channel for clarity.

protocol	terminology	reserved/well-known ports	dynamically assigned ports
TCP	port	1-1024	1025-65535
UDP	port	1-1024	1025-65535
RFCOMM	channel	none	1-30
L2CAP	PSM	odd numbered 1-4095	odd numbered 4097 - 32765

Table 2.2: Port numbers and their terminology for various protocols

In Internet programming, server applications traditionally make use of well known port numbers that are chosen and agreed upon at design time. Client applications will use the same well known port number to connect to a server. The main disadvantage to this approach is that it is not possible to run two server applications which both use the same port number. Due to the relative youth of TCP/IP and the large number of available port numbers to choose from, this has not yet become a serious issue.

The Bluetooth transport protocols, however, were designed with many fewer available port numbers, which means we cannot choose an arbitrary port number at design time. Although this problem is not as significant for L2CAP, which has around 15,000 unreserved port numbers, RFCOMM has only 30 different port numbers. A consequence of this is that there is a greater than 50% chance of port number collision with just 7 server applications. In this case, the application designer clearly should not arbitrarily choose port numbers. The Bluetooth answer to this problem is the Service Discovery Protocol (SDP).

Instead of agreeing upon a port to use at application design time, the Bluetooth approach is to assign ports at runtime and follow a publish-subscribe model. The host machine operates a server application, called the SDP server, that uses one of the few L2CAP reserved port numbers. Other server applications are dynamically assigned port numbers at runtime and register a description of themselves and the services they provide (along with the port numbers they are assigned) with the SDP server. Client applications will then query the SDP server (using the well defined port number) on a particular machine to obtain the information they need.

This raises the question of how do clients know which description is the one they are looking for. The standard way of doing this in Bluetooth is to assign a 128-bit number, called the Universally Unique Identifier (UUID), at design time. Following a standard method[18] of choosing this number guarantees choosing a UUID unique from those chosen by anyone else following the same method. Thus, a client and

server application both designed with the same UUID can provide this number to the SDP server as a search term.

As with RFCOMM and L2CAP, only a small portion of SDP has been described here - those parts most relevant to a network programmer. Among the other ways SDP can be used are to describe which transport protocols a server is using, to give information such as a human-readable description of the service provided and who is providing it, and to search on fields other than the UUID such as the service name. Another point worth mentioning is that SDP is not even required to create a Bluetooth application. It is perfectly possible to revert to the TCP/IP way of assigning port numbers at design time and hoping to avoid port conflicts, and this might often be done to save some time. In controlled settings such as the computer science laboratory, this is quite reasonable. Ultimately, however, to create a portable application that will run in the greatest number of scenarios, the application should use dynamically assigned ports and SDP.

2.5 Establishing connections and transferring data

It turns out that choosing which machine to connect to and how to connect are the most difficult parts of Bluetooth programming. In writing a server application, once the transport protocol and port number to listen on are chosen, building the rest of the application is essentially the same type of programming most network programmers are already accustomed to. A server application waiting for an incoming Bluetooth connection is conceptually the same as a server application waiting for an incoming Internet connection, and a client application attempting to establish an outbound connection appears the same whether it is using RFCOMM, L2CAP, TCP, or UDP.

Furthermore, once the connection has been established, the application is operating with the same guarantees, constraints, and error conditions as are encountered

in Internet programming. Depending on the transport protocol chosen, packets may be dropped or delayed. Connections may be severed due to host or link failures. External factors such as congestion and interference may result in decreased quality of service. Due to these conceptual similarities, it is perfectly reasonable to treat Bluetooth programming of an established connection in exactly the same manner that as an established connections in Internet programming.

2.6 Bluetooth Profiles + RFCs

Along with the simple TCP, IP, and UDP transport protocols used in Internet programming, there are a host of other protocols to specify, in great detail, methods to route data packets, exchange electronic mail, transfer files, load web pages, and more. Once standardized by the Internet Engineering Task Force in the form of Request For Comments (RFCs)[19], these protocols are generally adopted by the wider Internet community. Similarly, Bluetooth also has a method for proposing, ratifying, and standardizing protocols and specifications that are eventually adopted by the Bluetooth community. The Bluetooth equivalent of an RFC is a Bluetooth Profile.

Due to the short-range nature of Bluetooth, the Bluetooth Profiles tend to be complementary to the Internet RFCs, with emphasis on tasks that can assume physical proximity. For example, there is a profile for exchanging physical location information[4], a profile for printing to nearby printers[5], and a profile for using nearby modems[6] to make phone calls. There is even a specification for encapsulating TCP/IP traffic in a Bluetooth connection, which really does reduce Bluetooth programming to Internet programming. An overview of all the profiles available is beyond the scope of this chapter, but they are freely available for download at the Bluetooth website[7]

[4]Local Positioning Profile
[5]Basic Printing Profile
[6]Dial Up Networking Profile
[7]`http://www.bluetooth.org/spec/`

Chapter 3

PyBluez

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

The Zen of Python, by Tim Peters

Chapter 2 introduced the high level concepts needed to apply standard network programming techniques to Bluetooth programming. This chapter describes a Linux Python extension module that allows these concepts to be easily and quickly implemented in just a few lines of code.

Python is a versatile and powerful dynamically typed object oriented language, providing syntactic clarity along with built-in memory management so that the programmer can focus on the algorithm at hand without worrying about memory leaks or matching braces. Although Python has a large and comprehensive standard library,

Bluetooth support is not yet part of the standard distribution. A well documented C API allows software developers to create third-party extension modules that extend the language capabilities and provide access to operating system resources not otherwise exposed in Python.

PyBluez is a Python extension module written in C that provides access to system Bluetooth resources in an object oriented, modular manner. It is written for the GNU/Linux operating system and the BlueZ Bluetooth protocol stack. Appendix A provides instructions on how to obtain and install PyBluez.

3.1 Choosing a device

Example 3.1 shows a Python program that looks for a nearby device with the user-friendly name "My Phone". An explanation of the program follows.

Example 3.1: findmyphone.py

```
import Bluetooth

target_name = "My Phone"
target_address = None

nearby_devices = bluetooth.discover_devices()

for bdaddr in nearby_devices:
    if target_name == bluetooth.lookup_name( bdaddr ):
        target_address = bdaddr
        break

if target_address is not None:
    print "found target bluetooth device with address ", target_address
else:
    print "could not find target bluetooth device nearby"
```

PyBluez represents a bluetooth address as a string of the form "xx:xx:xx:xx:xx", where each x is a hexadecimal character representing one octet of the 48-bit address, with most significant octets listed first. Bluetooth devices in PyBluez will always be identified using an address string of this form.

Choosing a device really means choosing a bluetooth address. If only the user-friendly name of the target device is known, then two steps must be taken to find

the correct address. First, the program must scan for nearby Bluetooth devices. The routine discover_devices() scans for approximately 10 seconds and returns a list of addresses of detected devices. Next, the program uses the routine lookup_name() to connect to each detected device, requests its user-friendly name, and compares the result to the target name.

Since both the Bluetooth detection and name lookup process are probabilistic, discover_devices() will sometimes fail to detect devices that are in range, and lookup_name() will sometimes return None to indicate that it couldn't determine the user-friendly name of the detected device. In these cases, it may be a good idea to try again once or twice before giving up.

3.2 Communicating with RFCOMM

Bluetooth programming in Python follows the socket programming model. This is a concept that should be familiar to almost all network programmers, and makes the transition from Internet programming to Bluetooth programming much simpler. Examples 3.2 and 3.3 show how to establish a connection using an RFCOMM socket, transfer some data, and disconnect.

Example 3.2: rfcomm-server.py

```
import bluetooth

server_sock=bluetooth.BluetoothSocket( bluetooth.RFCOMM )

port = 1
server_sock.bind(("",port))
server_sock.listen(1)

client_sock,address = server_sock.accept()
print "Accepted connection from ",address

data = client_sock.recv(1024)
print "received [%s]" % data

client_sock.close()
server_sock.close()
```

Example 3.3: rfcomm-client.py

```
import bluetooth

bd_addr = "01:23:45:67:89:AB"

port = 1

sock=bluetooth.BluetoothSocket( bluetooth.RFCOMM )
sock.connect((bd_addr, port))

sock.send("hello!!")

sock.close()
```

In the socket programming model, a socket represents an endpoint of a communication channel. Sockets are not connected when they are first created, and are useless until a call to either connect (client application) or accept (server application) completes successfully. Once a socket is connected, it can be used to send and receive data until the connection fails due to link error or user termination.

PyBluez currently supports two types of BluetoothSocket objects: RFCOMM and L2CAP. The RFCOMM socket, shown above, is created by passing RFCOMM as an argument to the BluetoothSocket constructor. As the name suggests, it allocates resources for an RFCOMM based communication channel. The second type of BluetoothSocket, the L2CAP socket, is described in the next section.

An RFCOMM BluetoothSocket used to accept incoming connections must be attached to operating system resources with the bind method. bind takes in a tuple specifying the address of the local Bluetooth adapter to use and a port number to listen on. Usually, there is only one local Bluetooth adapter or it doesn't matter which one to use, so the empty string indicates that any local Bluetooth adapter is acceptable. Once a socket is bound, a call to listen puts the socket into listening mode and it is then ready to accept incoming connections.

The RFCOMM BluetoothSocket used to establish an outgoing connection connects to its target with the connect method, which also takes a tuple specifying an address and port number. In example 3.3, the client tries to connect to the Bluetooth device

with address "01:23:45:67:89:AB" on port 1. This example, and example 3.2, assumes that all communication happens on RFCOMM port 1. Section 3.4 shows how to dynamically choose ports and use SDP to search for which port a server is operating on.

Error handling code has been omitted for clarity in the examples, but is fairly straightforward. If any of the Bluetooth operations fail for some reason (e.g. connection timeout, no local bluetooth resources are available, etc.) then a BluetoothError is raised with an error message indicating the reason for failure.

3.3 Communicating with L2CAP

Examples 3.4 and 3.5 demonstrate the basics of using L2CAP as a transport protocol. As should be fairly obvious, using L2CAP sockets is almost identical to using RFCOMM sockets. The only difference is passing L2CAP to the BluetoothSocket constructor, and choosing an odd port number between 0x1001 and 0x8FFF instead of 1-30. The default connection settings provide a connection for sending reliably sequenced datagrams up to 672 bytes in size.

Example 3.4: l2cap-server.py

```
import bluetooth

server_sock=bluetooth.BluetoothSocket( bluetooth.L2CAP )

port = 0x1001
server_sock.bind(("",port))
server_sock.listen(1)

client_sock,address = server_sock.accept()
print "Accepted connection from ",address

data = client_sock.recv(1024)
print "received [%s]" % data

client_sock.close()
server_sock.close()
```

Example 3.5: l2cap-client.py

```
import bluetooth

sock=bluetooth.BluetoothSocket(bluetooth.L2CAP)

bd_addr = "01:23:45:67:89:AB"
port = 0x1001

sock.connect((bd_addr, port))

sock.send("hello!!")

sock.close()
```

As a maximum-length datagram protocol, packets sent on L2CAP connections have an upper size limit. Both devices at the endpoints of a connection maintain an *incoming maximum transmission unit (MTU)*, which specifies the maximum size packet can receive. If both parties adjust their incoming MTU, then it is possible to raise the MTU of the entire connection beyond the 672 byte default up to 65535 bytes. It is also possible, but uncommon, for the two devices to have different MTU values. In PyBluez, the set_l2cap_mtu method is used to adjust this value.

```
l2cap_sock = bluetooth.BluetoothSocket( bluetooth.L2CAP )
    .
    . # connect the socket
    .
bluetooth.set_l2cap_mtu( l2cap_sock, 65535 )
```

This method is fairly straightforward, and takes an L2CAP BluetoothSocket and a desired MTU as input. The incoming MTU is adjusted for the specified socket, and no other sockets are affected. As with all the other PyBluez methods, a failure is indicated by raising a BluetoothException.

Although we expressed reservations about using unreliable L2CAP channels in section 2.3.2, there are cases in which an unreliable connection may be desired. Adjusting the reliability semantics of a connection in PyBluez is also a simple task, and can be done with the set_packet_timeout method

```
bluetooth.set_packet_timeout( bdaddr, timeout )
```

set_packet_timeout takes a Bluetooth address and a timeout, specified in millisec-

onds, as input and tries to adjust the packet timeout for any L2CAP and RFCOMM connections to that device. The process must have superuser privileges, and there must be an active connection to the specified address. The effects of adjusting this parameter will last as long as any active connections are open, including those which outlive the Python program.

3.4 Service Discovery Protocol

So far this chapter has shown how to detect nearby Bluetooth device and establish the two main types of data transport connections, all using fixed Bluetooth address and port numbers that were determined at design time. As mentioned in section 2.4, this is not a recommended practice in general.

Dynamically allocating port numbers and using the Service Discovery Protocol (SDP) to search for and advertise services is a simple process in PyBluez. The get_available_port method finds available L2CAP and RFCOMM ports, advertise_service advertises a service with the local SDP server, and find_service searches Bluetooth devices for a specific service.

```
bluetooth.get_available_port( protocol )
```

get_available_port returns the first available port number for the specified protocol. Currently, only the RFCOMM and L2CAP protocols are supported. get_available_port only returns a port number, and does not actually reserve any resources, so it is possible that the availability changes between the time we call get_available_port and bind. If this happens, bind will simply raise a BluetoothException.

```
bluetooth.advertise_service( sock, name, uuid )
bluetooth.stop_advertising( sock )
bluetooth.find_service( name = None, uuid = None, bdaddr = None )
```

These three methods provide a way to advertise services on the local Bluetooth

device and search for them on one or many remote devices. advertise_service takes a socket that is bound and listening, a service name, and a UUID as input parameters. PyBluez requires the socket to be bound and listening because there is no point in advertising a service that does not exist yet. The UUID must always be a string of the form "xxxx-xxxx-xxxx-xxxx", where each 'x' is a hexadecimal digit. The service will be advertised as long as the socket is open, or until a call is made to stop_advertising, specifying the advertised socket.

find_service can search either a single device or all nearby devices for a specific service. It looks for a service with name and UUID that match name and uuid, at least one of which must be specified.. If bdaddr is None, then all nearby devices will be searched. In the special case that " localhost" is used for bdaddr, then the locally advertised SDP services will be searched. Otherwise, the function search the services provided by the Bluetooth device with address bdaddr.

On return, find_service returns a list of dictionaries. Each dictionary contains information about a matching service and has the entries "host", "name", "protocol", and "port". *host* indicates the address of the device advertising the service, *name* is the name of the service advertised, *protocol* will be either "L2CAP", "RFCOMM", or "UNKNOWN", and *port* will be the port number that the service is operating on. Typically, only the protocol and port number are needed to connect. Examples 3.6 and 3.7 show the RFCOMM client and server from the previous section modified to use dynamic port assignment and SDP to advertise and discover services.

Here, the server from example 3.2 is modified to use get_available_port and advertise_service . The (poorly chosen) UUID "1234-5678-9ABC-DEF0" is used to identify the "FooBar service". The client from example 3.3 is modified to use find_service to search for the the server, and connects to the first server found. The client makes an implicit assumption that the transport protocol used by the server is RFCOMM.

Example 3.6: rfcomm-server-sdp.py

```
import bluetooth

server_sock=bluetooth.BluetoothSocket( bluetooth.RFCOMM )

port = bluetooth.get_available_port( bluetooth.RFCOMM )
server_sock.bind(("",port))
server_sock.listen(1)
print "listening on port %d" % port

uuid = "1234-5678-9ABC-DEF0"
bluetooth.advertise_service( server_sock, "FooBar Service", uuid )

client_sock,address = server_sock.accept()
print "Accepted connection from ",address

data = client_sock.recv(1024)
print "received [%s]" % data

client_sock.close()
server_sock.close()
```

3.5 Advanced usage

Although the techniques described in this chapter so far should be sufficient for most Bluetooth applications with simple and straightforward requirements, some applications may require more advanced functionality or finer control over the Bluetooth system resources. This section describes asynchronous device detection and the _bluetooth module.

3.5.1 Asynchronous device discovery

The device discovery and remote name request methods described earlier are both synchronous methods in that they don't return until the requests are complete, which can often taken a long time. During this time, the controlling thread blocks and can't do anything else, such as responding to user input or displaying other information. To avoid this, PyBluez provides the DeviceDiscoverer class for asynchronous device discovery and name lookup.

Example 3.7: rfcomm-client-sdp.py

```
import sys
import bluetooth

uuid = "1234-5678-9ABC-DEF0"
service_matches = bluetooth.find_service( uuid = uuid )

if len(service_matches) == 0:
    print "couldn't find the FooBar service"
    sys.exit(0)

first_match = service_matches[0]
port = first_match["port"]
name = first_match["name"]
host = first_match["host"]

print "connecting to \"%s\" on %s" % (name, host)

sock=bluetooth.BluetoothSocket( bluetooth.RFCOMM )
sock.connect((host, port))
sock.send("hello!!")
sock.close()
```

Example 3.8: asynchronous-inquiry.py

```
import bluetooth
import select

class MyDiscoverer(bluetooth.DeviceDiscoverer):

    def pre_inquiry(self):
        self.done = False

    def device_discovered(self, address, device_class, name):
        print "%s - %s" % (address, name)

    def inquiry_complete(self):
        self.done = True

d = MyDiscoverer()
d.find_devices(lookup_names = True)

readfiles = [ d, ]

while True:
    rfds = select.select( readfiles, [], [] )[0]

    if d in rfds:
        d.process_event()

    if d.done: break
```

To asynchronously detect nearby bluetooth devices, create a subclass of DeviceDiscoverer and override the pre_inquiry, device_discovered, and inquiry_complete methods. To start the

discovery process, invoke find_devices, which returns immediately. pre_inquiry is called immediately before the actual inquiry process begins, and inquiry_complete is called as soon as the process completes.

MyDiscoverer exposes a fileno method, which allows it to be used with the select module. This provides a way for a single thread of control to wait for events on many open files at once, and greatly simplifies event-driven programs.

Call process_event to have the DeviceDiscoverer process pending events, which can be either a discovered device or the inquiry completion. When a nearby device is detected, device_discovered is invoked, with the address and device class of the detected device. If lookup_names was set in the call to find_devices, then name will also be set to the user-friendly name of the device. For more information about device classes, see [7]. The DeviceDiscoverer class can be used directly with the select module, and can easily be integrated into event loops of existing applications.

3.5.2 The `_bluetooth` module

The bluetooth module provides classes and utility functions useful for the most common Bluetooth programming tasks. More advanced functionality can be found in the _bluetooth extension module, which is little more than a thin wrapper around the BlueZ C API described in the next chapter. Lower level Bluetooth operations, such as establishing a connection with the actual Bluetooth microcontroller on the local machine and reading signal strength information, can be performed with the _bluetooth module in most cases without having to resort to the C API. An overview of the classes and methods available in _bluetooth is beyond the scope of this chapter, but the module documentation and examples are provided with the PyBluez distribution.

3.6 Alternatives

The main purpose of PyBluez is to allow Bluetooth programmers to quickly and easily develop and deploy Bluetooth applications. By providing high level objects and methods, PyBluez allows the programmer to focus on designing the algorithm and structure of the program instead having to worry about syntactic details, memory management, and parsing complex data structures and byte strings. PyBluez is not the only, or first, high level Bluetooth implementation with this goal.

3.7 Java

There are a number of Java bindings for Bluetooth programming currently available. The Java community has the advantage of having standardized on an API for Bluetooth development, called JSR-82. Almost all Java Bluetooth implementations adhere to this specification. This makes porting Bluetooth applications from one device to another much simpler. Current implementations of JSR-82 for the GNU/Linux operating system include Rocosoft Impronto[1], Avetana[2], and JavaBluetooth[3].

A disadvantage of using Java is that JSR-82 is very limited, providing virtually no control over the device discovery process or established data connections. For example, JSR-82 provides no method for adjusting delivery semantics, flushing a cache of previously detected devices during a device discovery, or obtaining signal strength information[4]. While JSR-82 is acceptable for creating simple Bluetooth applications, it is not well suited for research and academic purposes. Furthermore, Java and many JSR-82 implementations are not available on a number of platforms.

[1] `http://www.rococosoft.com`

[2] `http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.eng.xm`

[3] `http://www.javabluetooth.org`

[4] Cache flushing and signal strength were not covered in this chapter, but are described in the PyBluez documentation and examples

3.8 PyAffix

PyAffix[5] is also a Python extension module for GNU/Linux that provides access to system Bluetooth resources. Unlike PyBluez, PyAffix is written for the Affix Bluetooth protocol stack, which is an alternative Bluetooth implementation for GNU/Linux. BlueZ is the official Bluetooth protocol stack for GNU/Linux, and almost all Linux distributions are shipped with BlueZ. In order to use PyAffix, the user must first remove BlueZ and install Affix - a complex and laborious process requiring a high level of expertise. PyBluez originated as a port of PyAffix for BlueZ, and grew to provide functionality not present in PyAffix.

[5] `http://affix.sourceforge.net/pyaffix.shtml`

Chapter 4

Programming in BlueZ

There are reasons to prefer developing Bluetooth applications in C instead of in a high level language such as Python. The Python environment might not be available or might not fit on the target device; strict application requirements on program size, speed, and memory usage may preclude the use of an interpreted language like Python; the programmer may desire finer control over the local Bluetooth adapter than PyBluez provides; or the project may be to create a shared library for other applications to link against instead of a standalone application. As of this writing, BlueZ is a powerful Bluetooth communications stack with extensive APIs that allows a user to fully exploit all local Bluetooth resources, but it has no official documentation. Furthermore, there is very little unofficial documentation as well[1]. Novice developers requesting documentation on the official mailing lists[2] are typically rebuffed and told to figure out the API by reading through the BlueZ source code. This is a time consuming process that can only reveal small pieces of information at a time, and is quite often enough of an obstacle to deter many potential developers.

This chapter presents a short introduction to developing Bluetooth applications in C with BlueZ. The tasks covered in chapter 2 are now explained in greater detail

[1] in conversation with BlueZ developers on the BlueZ mailing lists

[2] `http://www.bluez.org/lists.html`

here for C programmers.

4.1 Choosing a communication partner

A simple program that detects nearby Bluetooth devices is shown in example 4.1. The program reserves system Bluetooth resources, scans for nearby Bluetooth devices, and then looks up the user friendly name for each detected device. A more detailed explanation of the data structures and functions used follows.

Example 4.1: A simple way to detect nearby Bluetooth devices

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>

int main(int argc, char **argv)
{
    inquiry_info *ii = NULL;
    int max_rsp, num_rsp;
    int dev_id, sock, len, flags;
    int i;
    char addr[19] = { 0 };
    char name[248] = { 0 };

    dev_id = hci_get_route(NULL);
    sock = hci_open_dev( dev_id );
    if (dev_id < 0 || sock < 0) {
        perror("opening socket");
        exit(1);
    }

    len  = 8;
    max_rsp = 255;
    flags = IREQ_CACHE_FLUSH;
    ii = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));

    num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL, &ii, flags);
    if( num_rsp < 0 ) perror("hci_inquiry");

    for (i = 0; i < num_rsp; i++) {
        ba2str(&(ii+i)->bdaddr, addr);
        memset(name, 0, sizeof(name));
        if (hci_read_remote_name(sock, &(ii+i)->bdaddr, sizeof(name),
            name, 0) < 0)
        strcpy(name, "[unknown]");
        printf("%s  %s\n", addr, name);
```

```
    }

    free( ii );
    close( sock );
    return 0;
}
```

Compilation

To compile our program, invoke gcc and link against libbluetooth

```
# gcc -o simplescan simplescan.c -lbluetooth
```

Explanation

```
typedef struct {
        uint8_t b[6];
} __attribute__((packed)) bdaddr_t;
```

The basic data structure used to specify a Bluetooth device address is the bdaddr_t. All Bluetooth addresses in BlueZ will be stored and manipulated as bdaddr_t structures. BlueZ provides two convenience functions to convert between strings and bdaddr_t structures.

```
int str2ba( const char *str, bdaddr_t *ba );
int ba2str( const bdaddr_t *ba, char *str );
```

str2ba takes an string of the form "XX:XX:XX:XX:XX:XX", where each XX is a hexadecimal number specifying an octet of the 48-bit address, and packs it into a 6-byte bdaddr_t. ba2str does exactly the opposite.

Local Bluetooth adapters are assigned identifying numbers starting with 0, and a program must specify which adapter to use when allocating system resources. This can be done with hci_get_route, which returns the the resource number of the specified adapter. Usually, there is only one adapter or it doesn't matter which one is used, so passing NULL to hci_get_route will retrieve the resource number of the first available Bluetooth adapter.

```
int hci_get_route( bdaddr_t *bdaddr );
int hci_open_dev( int dev_id );
```

Most Bluetooth operations require the use of an open socket. hci_open_dev is a convenience function that opens a Bluetooth socket with the specified resource number[3]. To be clear, the socket opened by hci_open_dev represents a connection to the microcontroller on the specified local Bluetooth adapter, and not a connection to a remote Bluetooth device. Performing low level Bluetooth operations involves sending commands directly to the microcontroller with this socket, and section 4.5 discusses this in greater detail.

If there are multiple Bluetooth adapters present, then to use the adapter with address "01:23:45:67:89:AB", pass the bdaddr_t representation of the address to hci_get_route.

```
char *dest_str = "01:23:45:67:89:AB";
bdaddr_t dest_ba;
str2ba( dest_str, &dest_ba );
dev_id = hci_get_route( &dest_ba );
```

After choosing the local Bluetooth adapter to use and allocating system resources, the program is ready to scan for nearby Bluetooth devices. In the example, hci_inquiry performs a Bluetooth device discovery and returns a list of detected devices and some basic information about them in the variable ii. On error, it returns -1 and sets errno accordingly.

```
int hci_inquiry(int dev_id, int len, int max_rsp, const uint8_t *lap,
                inquiry_info **ii, long flags);
```

hci_inquiry is one of the few functions that requires the use of a resource number instead of an open socket, so we use the dev_id returned by hci_get_route. The inquiry lasts for at most 1.28 * len seconds, and at most max_rsp devices will be returned in

[3]for the curious, it makes a call to socket(AF_BLUETOOTH, SOCK_RAW, BTPROTO_HCI), followed by a call to bind with the specified resource number.

the output parameter ii, which must be large enough to accommodate max_rsp results. We suggest using a max_rsp of 255 for a standard 10.24 second inquiry.

If flags is set to IREQ_CACHE_FLUSH, then the cache of previously detected devices is flushed before performing the current inquiry. Otherwise, if flags is set to 0, then the results of previous inquiries may be returned, even if the devices aren't in range anymore.

The inquiry_info structure is defined as

```
typedef struct {
    bdaddr_t     bdaddr;
    uint8_t      pscan_rep_mode;
    uint8_t      pscan_period_mode;
    uint8_t      pscan_mode;
    uint8_t      dev_class[3];
    uint16_t     clock_offset;
} __attribute__ ((packed)) inquiry_info;
```

For the most part, only the first entry - the bdaddr field, which gives the address of the detected device - is of any use. Occasionally, there may be a use for the dev_class field, which gives information about the type of device detected (i.e. if it's a printer, phone, desktop computer, etc.) and is described in the Bluetooth Assigned Numbers[7]. The rest of the fields are used for low level communication, and are not useful for most purposes. The interested reader can see the Bluetooth Core Specification[9] for more details.

Once a list of nearby Bluetooth devices and their addresses has been found, the program determines the user-friendly names associated with those addresses and presents them to the user. The hci_read_remote_name function is used for this purpose.

```
int hci_read_remote_name(int sock, const bdaddr_t *ba, int len,
                         char *name, int timeout)
```

hci_read_remote_name tries for at most timeout milliseconds to use the socket sock to query the user-friendly name of the device with Bluetooth address ba. On success,

hci_read_remote_name returns 0 and copies at most the first len bytes of the device's user-friendly name into name. On failure, it returns -1 and sets errno accordingly.

4.2 RFCOMM sockets

As with Python, establishing and using RFCOMM connections boils down to the same socket programming techniques we already know how to use for TCP/IP programming. The only difference is that the socket addressing structures are different, and we use different functions for byte ordering of multibyte integers. Examples 4.2 and 4.3 show how to establish a connection using an RFCOMM socket, transfer some data, and disconnect. For simplicity, the client is hard-coded to connect to "01:23:45:67:89:AB".

Example 4.2: rfcomm-server.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int s, client, bytes_read;
    int opt = sizeof(rem_addr);

    // allocate socket
    s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

    // bind socket to port 1 of the first available
    // local bluetooth adapter
    loc_addr.rc_family = AF_BLUETOOTH;
    loc_addr.rc_bdaddr = *BDADDR_ANY;
    loc_addr.rc_channel = (uint8_t) 1;
    bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));

    // put socket into listening mode
    listen(s, 1);

    // accept one connection
    client = accept(s, (struct sockaddr *)&rem_addr, &opt);

    ba2str( &rem_addr.rc_bdaddr, buf );
    fprintf(stderr, "accepted connection from %s\n", buf);
    memset(buf, 0, sizeof(buf));

    // read data from the client
    bytes_read = read(client, buf, sizeof(buf));
    if( bytes_read > 0 ) {
        printf("received [%s]\n", buf);
    }

    // close connection
    close(client);
    close(s);
    return 0;
}
```

Example 4.3: rfcomm-client.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/rfcomm.h>

int main(int argc, char **argv)
{
    struct sockaddr_rc addr = { 0 };
    int s, status;
    char dest[18] = "01:23:45:67:89:AB";

    // allocate a socket
    s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_RFCOMM);

    // set the connection parameters (who to connect to)
    addr.rc_family = AF_BLUETOOTH;
    addr.rc_channel = (uint8_t) 1;
    str2ba( dest, &addr.rc_bdaddr );

    // connect to server
    status = connect(s, (struct sockaddr *)&addr, sizeof(addr));

    // send a message
    if( status == 0 ) {
        status = write(s, "hello!", 6);
    }

    if( status < 0 ) perror("uh oh");

    close(s);
    return 0;
}
```

Most of this should look familiar to the experienced network programmer. As with Internet programming, first allocate a socket with the socket system call. Instead of AF_INET, use AF_BLUETOOTH, and instead of IPPROTO_TCP, use BTPROTO_RFCOMM. Since RFCOMM provides the same delivery semantics as TCP, SOCK_STREAM can still be used for the socket type.

Addressing structures

To establish an RFCOMM connection with another Bluetooth device, incoming or outgoing, create and fill out a **struct** sockaddr_rc addressing structure. Like the **struct** sockaddr_in that is used in TCP/IP, the addressing structure specifies the details of an outgoing connection or listening socket.

```
struct sockaddr_rc {
        sa_family_t     rc_family;
        bdaddr_t        rc_bdaddr;
        uint8_t         rc_channel;
};
```

The rc_family field specifies the addressing family of the socket, and will always be AF_BLUETOOTH. For an outgoing connection, rc_bdaddr and rc_channel specify the Bluetooth address and port number to connect to, respectively. For a listening socket, rc_bdaddr specifies the local Bluetooth adapter to use, and is typically set to BDADDR_ANY to indicate that any local Bluetooth adapter is acceptable. For listening sockets, rc_channel specifies the port number to listen on.

A note on byte ordering

Since Bluetooth deals with the transfer of data from one machine to another, the use of a consistent byte ordering for multi-byte data types is crucial. Unlike network byte ordering, which uses a big-endian format, Bluetooth byte ordering is little-endian, where the least significant bytes are transmitted first. BlueZ provides four convenience functions to convert between host and Bluetooth byte orderings.

```
unsigned short int htobs( unsigned short int num );
unsigned short int btohs( unsigned short int num );
unsigned int htobl( unsigned int num );
unsigned int btohl( unsigned int num );
```

Like their network order counterparts, these functions convert 16 and 32 bit unsigned integers to Bluetooth byte order and back. They are used when filling in the socket addressing structures, communicating with the Bluetooth microcontroller, and when performing low level operations on transport protocol sockets.

Dynamically assigned port numbers

For Linux kernel versions 2.6.7 and greater, dynamically binding to an RFCOMM or L2CAP port is simple. The rc_channel field of the socket addressing structure used to bind the socket is simply set to 0, and the kernel binds the socket to the first available port. Unfortunately, for earlier versions of the Linux kernel, the only way to bind to the first available port number is to try binding to *every* possible port and stopping when bind doesn't fail. The following function illustrates how to do this for RFCOMM sockets.

```
int dynamic_bind_rc(int sock, struct sockaddr_rc *sockaddr, uint8_t *port)
{
    int err;
    for( *port = 1; *port <= 31; *port++ ) {
        sockaddr->rc_channel = *port;
        err = bind(sock, (struct sockaddr *)sockaddr, sizeof(sockaddr));
        if( ! err || errno == EINVAL ) break;
    }
    if( port == 31 ) {
        err = -1;
        errno = EINVAL;
    }
    return err;
}
```

The process for L2CAP sockets is similar, but tries odd-numbered ports 4097-32767 (0x1001 - 0x7FFF) instead of ports 1-30.

RFCOMM summary

Advanced TCP options that are often set with setsockopt, such as receive windows and the Nagle algorithm, don't make sense in Bluetooth, and can't be used with RFCOMM sockets. Aside from this, the byte ordering, and socket addressing structure differences, programming RFCOMM sockets is virtually identical to programming TCP sockets. To accept incoming connections with a socket, use bind to reserve operating system resource, listen to put it in listening mode, and accept to block and accept an incoming connection. Creating an outgoing connection is also simple and merely involves a call to connect. Once a connection has been established, the standard calls

to read, write, send, and recv can be used for data transfer.

4.3 L2CAP sockets

As with RFCOMM, L2CAP communications are structured around socket programming. Examples 4.4 and 4.5 demonstrate how to establish an L2CAP channel and transmit a short string of data. For simplicity, the client is hard-coded to connect to "01:23:45:67:89:AB".

Example 4.4: l2cap-server.c

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>

int main(int argc, char **argv)
{
    struct sockaddr_l2 loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int s, client, bytes_read;
    int opt = sizeof(rem_addr);

    // allocate socket
    s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);

    // bind socket to port 0x1001 of the first available
    // bluetooth adapter
    loc_addr.l2_family = AF_BLUETOOTH;
    loc_addr.l2_bdaddr = *BDADDR_ANY;
    loc_addr.l2_psm = htobs(0x1001);

    bind(s, (struct sockaddr *)&loc_addr, sizeof(loc_addr));

    // put socket into listening mode
    listen(s, 1);

    // accept one connection
    client = accept(s, (struct sockaddr *)&rem_addr, &opt);

    ba2str( &rem_addr.l2_bdaddr, buf );
    fprintf(stderr, "accepted connection from %s\n", buf);

    memset(buf, 0, sizeof(buf));

    // read data from the client
    bytes_read = read(client, buf, sizeof(buf));
    if( bytes_read > 0 ) {
        printf("received [%s]\n", buf);
```

```
    }

    // close connection
    close(client);
    close(s);
}
```

Example 4.5: l2cap-client.c

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/l2cap.h>

int main(int argc, char **argv)
{
    struct sockaddr_l2 addr = { 0 };
    int s, status;
    char *message = "hello!";
    char dest[18] = "01:23:45:67:89:AB";

    if(argc < 2)
    {
        fprintf(stderr, "usage: %s <bt_addr>\n", argv[0]);
        exit(2);
    }

    strncpy(dest, argv[1], 18);

    // allocate a socket
    s = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);

    // set the connection parameters (who to connect to)
    addr.l2_family = AF_BLUETOOTH;
    addr.l2_psm = htobs(0x1001);
    str2ba( dest, &addr.l2_bdaddr );

    // connect to server
    status = connect(s, (struct sockaddr *)&addr, sizeof(addr));

    // send a message
    if( status == 0 ) {
        status = write(s, "hello!", 6);
    }

    if( status < 0 ) perror("uh oh");

    close(s);
}
```

For simple usage scenarios, the only differences are the socket type specified, the protocol family, and the addressing structure. By default, L2CAP connections provide reliable datagram-oriented connections with packets delivered in order, so the socket

type is SOCK_SEQPACKET, and the protocol is BTPROTO_L2CAP. The addressing structure **struct** sockaddr_l2 differs slightly from the RFCOMM addressing structure.

```
struct sockaddr_l2 {
    sa_family_t     l2_family;
    unsigned short  l2_psm;
    bdaddr_t        l2_bdaddr;
};
```

The l2_psm field specifies the L2CAP port number to use. Since it is a multibyte unsigned integer, byte ordering is significant. The htbos function, described earlier, is used here to convert numbers to Bluetooth byte order.

Maximum Transmission Unit

Occasionally, an application may need to adjust the maximum transmission unit (MTU) for an L2CAP connection and set it to something other than the default of 672 bytes. In BlueZ, this is done with the getsockopt and setsockopt functions.

```
struct l2cap_options {
    uint16_t    omtu;
    uint16_t    imtu;
    uint16_t    flush_to;
    uint8_t     mode;
};

int set_l2cap_mtu( int sock, uint16_t mtu ) {
        struct l2cap_options opts;
    int optlen = sizeof(opts), err;
    err = getsockopt( s, SOL_L2CAP, L2CAP_OPTIONS, &opts, &optlen );
    if( ! err ) {
        opts.omtu = opts.imtu = mtu;
        err = setsockopt( s, SOL_L2CAP, L2CAP_OPTIONS, &opts, optlen );
    }
    return err;
};
```

The omtu and imtu fields of the **struct** l2cap_options are used to specify the *outgoing MTU* and *incoming MTU*, respectively. The other two fields are currently unused and reserved for future use. To adjust the connection-wide MTU, both clients must adjust their outgoing and incoming MTUs. Bluetooth allows the MTU to range from a minimum of 48 bytes to a maximum of 65,535 bytes.

Unreliable sockets

It is slightly misleading to say that L2CAP sockets are reliable by default. Multiple L2CAP and RFCOMM connections between two devices are actually logical connections multiplexed on a single, lower level connection[4] established between them. The only way to adjust delivery semantics is to adjust them for the lower level connection, which in turn affects *all* L2CAP and RFCOMM connections between the two devices.

As we delve deeper into the more complex aspects of Bluetooth programming, the interface becomes a little harder to manage. Unfortunately, BlueZ does not provide an easy way to change the packet timeout for a connection. A handle to the underlying connection is first needed to make this change, but the only way to obtain a handle to the underlying connection is to query the microcontroller on the local Bluetooth adapter. Once the connection handle has been determined, a command can be issued to the microcontroller instructing it to make the appropriate adjustments. Example 4.6 shows how to do this.

Example 4.6: set-flush-to.c

```
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/hci_lib.h>

int set_flush_timeout(bdaddr_t *ba, int timeout)
{
    int err = 0, dd;
    struct hci_conn_info_req *cr = 0;
    struct hci_request rq = { 0 };

    struct {
        uint16_t handle;
        uint16_t flush_timeout;
    } cmd_param;

    struct {
        uint8_t  status;
        uint16_t handle;
```

[4] In Bluetooth terminology refers to this as the ACL connection

```
    } cmd_response;

    // find the connection handle to the specified bluetooth device
    cr = (struct hci_conn_info_req*) malloc(
            sizeof(struct hci_conn_info_req) +
            sizeof(struct hci_conn_info));
    bacpy( &cr->bdaddr, ba );
    cr->type = ACL_LINK;
    dd = hci_open_dev( hci_get_route( &cr->bdaddr ) );
    if( dd < 0 ) {
        err = dd;
        goto cleanup;
    }
    err = ioctl(dd, HCIGETCONNINFO, (unsigned long) cr );
    if( err ) goto cleanup;

    // build a command packet to send to the bluetooth microcontroller
    cmd_param.handle = cr->conn_info->handle;
    cmd_param.flush_timeout = htobs(timeout);
    rq.ogf = OGF_HOST_CTL;
    rq.ocf = 0x28;
    rq.cparam = &cmd_param;
    rq.clen = sizeof(cmd_param);
    rq.rparam = &cmd_response;
    rq.rlen = sizeof(cmd_response);
    rq.event = EVT_CMD_COMPLETE;

    // send the command and wait for the response
    err = hci_send_req( dd, &rq, 0 );
    if( err ) goto cleanup;

    if( cmd_response.status ) {
        err = -1;
        errno = bt_error(cmd_response.status);
    }

cleanup:
    free(cr);
    if( dd >= 0) close(dd);
    return err;
}
```

On success, the packet timeout for the low level connection to the specified device is set to timeout $*$ 1.28 milliseconds. A timeout of 0 is used to indicate infinity, and is how to revert back to a reliable connection. The bulk of this function is comprised of code to construct the command packets and response packets used in communicating with the Bluetooth controller. The Bluetooth Specification defines the structure of these packets and the magic number 0x28. In most cases, BlueZ provides convenience functions to construct the packets, send them, and wait for the response. Setting the packet timeout, however, seems to be so rarely used that no convenience function for

it currently exists.

4.4 Service Discovery Protocol

The process of searching for services involves two steps - detecting all nearby devices with a device inquiry, and connecting to each of those devices in turn to search for the desired service. Despite Bluetooth's piconet abilities, the early versions don't support multicasting queries, so this must be done the slow way. Since detecting nearby devices was covered in section 4.1, only the second step is described here.

Searching a specific device for a service also involves two steps. The first part, shown in example 4.7, requires connecting to the device and sending the search request. The second part, shown in in example 4.8, involves parsing and interpreting the search results.

Example 4.7: Step one of searching a device for a service with UUID 0xABCD

```
#include <bluetooth/bluetooth.h>
#include <bluetooth/sdp.h>
#include <bluetooth/sdp_lib.h>

int main(int argc, char **argv)
{
    uint32_t svc_uuid_int[] = { 0x0, 0x0, 0x0, 0xABCD };
    uuid_t svc_uuid;
    int err;
    bdaddr_t target;
    sdp_list_t *response_list = NULL, *search_list, *attrid_list;
    sdp_session_t *session = 0;

    str2ba( "01:23:45:67:89:AB", &target );

    // connect to the SDP server running on the remote machine
    session = sdp_connect( BDADDR_ANY, &target, SDP_RETRY_IF_BUSY );

    // specify the UUID of the application we're searching for
    sdp_uuid128_create( &svc_uuid, &svc_uuid_int );
    search_list = sdp_list_append( NULL, &svc_uuid );

    // specify that we want a list of all the matching applications' attributes
    uint32_t range = 0x0000ffff;
    attrid_list = sdp_list_append( NULL, &range );

    // get a list of service records that have UUID 0xabcd
    err = sdp_service_search_attr_req( session, search_list, \
            SDP_ATTR_REQ_RANGE, attrid_list, &response_list);
    .
    .
```

The uuid_t data type is used to represent the 128-bit UUID that identifies the desired service. To obtain a valid uuid_t, create an array of 4 32-bit integers and use the sdp_uuid128_create function, which is similar to the str2ba function for converting strings to bdaddr_t types. sdp_connect synchronously connects to the SDP server running on the target device. sdp_service_search_attr_req searches the connected device for the desired service and requests a list of attributes specified by attrid_list . It's easiest to use the magic number 0x0000ffff to request a list of all the attributes describing the service, although it is possible, for example, to request only the name of a matching service and not its protocol information.

Continuing our example, we now get to the tricky part - parsing and interpreting the results of a search. Unfortunately, there isn't an easy way to do this. Taking the result of our search above, example 4.8 shows how to extract the RFCOMM channel of a matching result.

Example 4.8: parsing and interpreting a search result

```
sdp_list_t *r = response_list;

// go through each of the service records
for (; r; r = r->next ) {
    sdp_record_t *rec = (sdp_record_t*) r->data;
    sdp_list_t *proto_list;

    // get a list of the protocol sequences
    if( sdp_get_access_protos( rec, &proto_list ) == 0 ) {
    sdp_list_t *p = proto_list;

    // go through each protocol sequence
    for( ; p ; p = p->next ) {
        sdp_list_t *pds = (sdp_list_t*)p->data;

        // go through each protocol list of the protocol sequence
        for( ; pds ; pds = pds->next ) {

            // check the protocol attributes
            sdp_data_t *d = (sdp_data_t*)pds->data;
            int proto = 0;
            for( ; d; d = d->next ) {
                switch( d->dtd ) {
                    case SDP_UUID16:
                    case SDP_UUID32:
                    case SDP_UUID128:
                        proto = sdp_uuid_to_proto( &d->val.uuid );
                        break;
                    case SDP_UINT8:
                        if( proto == RFCOMM_UUID ) {
                            printf("rfcomm channel: %d\n",d->val.int8);
                        }
```

```
                            break;
                        }
                    }
                }
                sdp_list_free( (sdp_list_t *)p->data, 0 );
            }
            sdp_list_free( proto_list, 0 );

            }

            printf("found service record 0x%x\n", rec->handle);
            sdp_record_free( rec );
        }

        sdp_close(session);
}
```

Getting the protocol information requires digging deep into the search results. Since it's possible for multiple application services to match a single search request, a list of *service records* is used to describe each matching service. For each service that's running, it's (theoretically, but not usually done in practice) possible to have different ways of connecting to the service. So each service record has a list of *protocol sequences* that each describe a different way to connect. Furthermore, since protocols can be built on top of other protocols (e.g. RFCOMM uses L2CAP as a transport), each protocol sequence has a list of *protocols* that the application uses, only one of which actually matters. Finally, each protocol entry will have a list of *attributes*, like the protocol type and the port number it's running on. Thus, obtaining the port number for an application that uses RFCOMM requires finding the port number protocol attribute in the RFCOMM protocol entry.

In this example, several new data structures have been introduced that we haven't seen before.

```
typedef struct _sdp_list_t {
    struct _sdp_list_t *next;
    void *data;
} sdp_list_t;

typedef void(*sdp_free_func_t)(void *)

sdp_list_t *sdp_list_append(sdp_list_t *list, void *d);
sdp_list_t *sdp_list_free(sdp_list_t *list, sdp_list_func_t f);
```

Since C does not have a built in linked-list data structure, and SDP search criteria

and search results are essentially nothing but lists of data, the BlueZ developers wrote their own linked list data structure and called it sdp_list_t. For now, it suffices to know that appending to a NULL list creates a new linked list, and that a list must be deallocated with sdp_list_free when it is no longer needed.

```
typedef struct {
        uint32_t handle;
        sdp_list_t *pattern;
        sdp_list_t *attrlist;
} sdp_record_t;
```

The sdp_record_t data structure represents a single service record being advertised by another device. Its inner details aren't important, as there are a number of helper functions available to get information in and out of it. In this example, sdp_get_access_protos is used to extract a list of the protocols for the service record.

```
typedef struct sdp_data_struct sdp_data_t;
struct sdp_data_struct {
        uint8_t dtd;
        uint16_t attrId;
        union {
                int8_t    int8;
                int16_t   int16;
                int32_t   int32;
                int64_t   int64;
                uint128_t int128;
                uint8_t   uint8;
                uint16_t  uint16;
                uint32_t  uint32;
                uint64_t  uint64;
                uint128_t uint128;
                uuid_t    uuid;
                char     *str;
                sdp_data_t *dataseq;
        } val;
        sdp_data_t *next;
        int unitSize;
};
```

Finally, there is the sdp_data_t structure, which is ultimately used to store each element of information in a service record. At a high level, it is a node of a linked list that carries a piece of data (the val field). As a variable type data structure, it can be used in different ways, depending on the context. For now, it's sufficient to know that each protocol stack in the list of protocol sequences is represented as a singly linked list of sdp_data_t structures, and extracting the protocol and port information

requires iterating through this list until the proper elements are found. The type of a sdp_data_t is specified by the dtd field, which is what we use to search the list.

sdpd - The SDP daemon

Every Bluetooth device typically runs an SDP server that answers queries from other Bluetooth devices. In BlueZ, the implementation of the SDP server is called sdpd, and is usually started by the system boot scripts. sdpd handles all incoming SDP search requests. Applications that need to advertise a Bluetooth service must use inter-process communication (IPC) methods to tell sdpd what to advertise. Currently, this is done with the named pipe /var/run/sdp. BlueZ provides convenience functions written to make this process a little easier.

Registering a service with sdpd involves describing the service to advertise, connected to sdpd, instructing sdpd on what to advertise, and then disconnecting.

Describing a service

Describing a service is essentially building the service record that was parsed in the previous examples. This involves creating several lists and populating them with data attributes. Example 4.9 shows how to describe a service application with UUID 0xABCD that runs on RFCOMM channel 11, is named "Roto-Rooter Data Router", provided by "Roto-Rooter", and has the description "An experimental plumbing router"

Example 4.9: Describing a service

```
#include <bluetooth/bluetooth.h>
#include <bluetooth/sdp.h>
#include <bluetooth/sdp_lib.h>

sdp_session_t *register_service()
{
    uint32_t service_uuid_int[] = { 0, 0, 0, 0xABCD };
    uint8_t rfcomm_channel = 11;
    const char *service_name = "Roto-Rooter Data Router";
    const char *service_dsc = "An experimental plumbing router";
```

```
const char *service_prov = "Roto-Rooter";

uuid_t root_uuid, l2cap_uuid, rfcomm_uuid, svc_uuid;
sdp_list_t *l2cap_list = 0,
           *rfcomm_list = 0,
           *root_list = 0,
           *proto_list = 0,
           *access_proto_list = 0;
sdp_data_t *channel = 0, *psm = 0;

sdp_record_t record = sdp_record_alloc();

// set the general service ID
sdp_uuid128_create( &svc_uuid, &service_uuid_int );
sdp_set_service_id( &record, svc_uuid );

// make the service record publicly browsable
sdp_uuid16_create(&root_uuid, PUBLIC_BROWSE_GROUP);
root_list = sdp_list_append(0, &root_uuid);
sdp_set_browse_groups( &record, root_list );

// set l2cap information
sdp_uuid16_create(&l2cap_uuid, L2CAP_UUID);
l2cap_list = sdp_list_append( 0, &l2cap_uuid );
proto_list = sdp_list_append( 0, l2cap_list );

// set rfcomm information
sdp_uuid16_create(&rfcomm_uuid, RFCOMM_UUID);
channel = sdp_data_alloc(SDP_UINT8, &rfcomm_channel);
rfcomm_list = sdp_list_append( 0, &rfcomm_uuid );
sdp_list_append( rfcomm_list, channel );
sdp_list_append( proto_list, rfcomm_list );

// attach protocol information to service record
access_proto_list = sdp_list_append( 0, proto_list );
sdp_set_access_protos( &record, access_proto_list );

// set the name, provider, and description
sdp_set_info_attr(&record, service_name, service_prov, service_dsc);
.
.
```

Registering a service

Building the description is quite straightforward, and consists of taking those five fields and packing them into data structures. Most of the work is just putting lists together. Once the service record is complete, the application connects to the local SDP server and registers a new service, taking care afterwards to free the data structures allocated earlier.

```
    .
    .
    int err = 0;
    sdp_session_t *session = 0;

    // connect to the local SDP server, register the service record, and
    // disconnect
    session = sdp_connect( BDADDR_ANY, BDADDR_LOCAL, SDP_RETRY_IF_BUSY );
    err = sdp_record_register(session, &record, 0);

    // cleanup
    sdp_data_free( channel );
    sdp_list_free( l2cap_list, 0 );
    sdp_list_free( rfcomm_list, 0 );
    sdp_list_free( root_list, 0 );
    sdp_list_free( access_proto_list, 0 );

    return session;
}
```

The special argument BDADDR_LOCAL causes sdp_connect to connect to the local SDP server (via the named pipe /var/run/sdp) instead of a remote device. Once an active session is established with the local SDP server, sdp_record_register advertises a service record. The service will be advertised for as long as the session with the SDP server is kept open. As soon as the SDP server detects that the socket connection is closed, it will stop advertising the service. sdp_close terminates a session with the SDP server.

```
sdp_session_t *sdp_connect( const bdaddr_t *src, const bdaddr_t *dst, uint32_t
        flags );
int sdp_close( sdp_session_t *session );

int sdp_record_register(sdp_session_t *sess, sdp_record_t *rec, uint8_t flags);
```

4.5 Advanced BlueZ programming

In addition to the L2CAP and RFCOMM sockets described in this chapter, BlueZ provides a number of other socket types. The most useful of these is the Host Controller Interface (HCI) socket, which provides a direct connection to the microcontroller on the local Bluetooth adapter. This socket type, introduced in section 4.1, can be used to issue arbitrary commands to the Bluetooth adapter. Programmers requiring precise control over the Bluetooth controller to perform tasks such as asynchronous

device discovery or reading signal strength information should use HCI sockets.

The Bluetooth Core Specification[8, 9] describes communication with a Bluetooth microcontroller in great detail, which we summarize here. The host computer can send commands to the microcontroller, and the microcontroller generates events to indicate command responses and other status changes. A command consists of a Opcode Group Field that specifies the general category the command falls into, an Opcode Command Field that specifies the actual command, and a series of command parameters. In BlueZ, hci_send_cmd is used to transmit a command to the microcontroller.

```
int hci_send_cmd(int sock, uint16_t ogf, uint16_t ocf, uint8_t plen,
                 void *param);
```

Here, sock is an open HCI socket, ogf is the Opcode Group Field, ocf is the Opcode Command Field, and plen specifies the length of the command parameters param.

Calling read on an open HCI socket waits for and receives the next event from the microcontroller. An event consists of a header field specifying the event type, and the event parameters. A program that requires asynchronous device detection would, for example, send a command with ocf of OCF_INQUIRY and wait for events of type EVT_INQUIRY_RESULT and EVT_INQUIRY_COMPLETE. The specific codes to use for each command and event are defined in the specifications and in the BlueZ source code.

4.6 Chapter Summary

This chapter has provided an introduction to Bluetooth programming with BlueZ. The concepts covered in chapter 2 were presented here in greater detail with examples on how to implement them in BlueZ. Many other useful aspects of BlueZ were left out for brevity. Specifically, the command line tools and utilities that are distributed

with BlueZ, such as hciconfig, hcitool, sdptool, and hcidump, are not described here. These utilities, which are invaluable to a serious Bluetooth developer, are already well documented. Only the simplest aspects of using the Service Discovery Protocol were covered - just enough to search for and advertise services. Additionally, other socket types such as BTPROTO_SCO and BTPROTO_BNEP were left out, as they are not crucial to forming a working knowledge of programming with BlueZ. Unfortunately, as of now there is no official API reference to refer to, so more curious readers are advised to download and examine the BlueZ source code[5].

[5]available at `http://www.bluez.org`

Chapter 5

The Bluetooth Location Infrastructure

A low cost and easy to deploy location awareness infrastructure requires a fast and reliable method to find nearby devices. Location aware computing provides applications with knowledge of the physical location where the computation is taking place, allowing applications to operate in a more context-sensitive fashion. However, to date, the infrastructure is expensive and difficult to deploy. Bluetooth is a stable, inexpensive, and mature technology upon which a location aware infrastructure can be built.

We propose placing Bluetooth devices key locations throughout a building, turning them into location beacons. The user is equipped with a locator device, usually a Bluetooth-enabled cell phone or PDA mobile device, which scans the environment for the location beacons. When a locator is within 10 meters, the location beacons respond, allowing the locator to estimate its position. In this chapter, we describe the deployment and evaluation of such a system.

5.1 Requirements

An effective location aware system should emphasize three main features. The first concerns the physical performance of the system. The second concerns protecting and preserving the privacy of the users of the system. The third concerns its practicality, with an emphasis on deployment, usage, and maintenance costs.

5.1.1 Performance

The performance of a location aware system can be characterized by how accurately it provides location measurements and how frequently it can update them. An ideal system would provide perfect accuracy that is continuously updated, with the time between updates being imperceptibly small. In practice, there is typically a price-performance tradeoff that limits the accuracy of a system. Although performance requirements will vary with the intended applications, we find it reasonable to require a system to have room-level precision maintainable at walking speed. This roughly translates to 3-meter accuracy with updates spaced a few seconds apart.

5.1.2 Privacy

Privacy is one of the most oft cited concerns in ubiquitous computing and could ultimately be the biggest factor in its success or failure. Formal studies[5, 28, 20] indicate that it is a major concern for consumers and participants in a location aware system. At first, it may not seem intuitive. After all, most people don't take special care to hide their daily movements, and typically don't mind if their friends, colleagues, or acquaintances happen to know where they are. But once the system makes it possible to immediately and precisely determine someone's location over an entire day, week, or month, we suddenly become much more wary of our situation, and are less willing to participate. We find it intrusive and Orwellian that someone else could track our

every movement, especially without our knowledge.

In light of these concerns, we stipulate that a well-designed location aware system should preserve the privacy of its users. Users should be able to choose whether or not to reveal their location and identity to others, and should not be actively tracked without explicit consent. It should not be *possible* for the system to track users without permission.

5.1.3 Practicality

Ultimately, in order for a location aware system to be useful and effective, it must also be practical. The infrastructure should be inexpensive and scale easily with the covered area. It should not be difficult or expensive (with respect to both time and money) to deploy or maintain. The devices that are to take advantage of this network should be able to do so easily, cheaply, and with as little modification as possible.

With the recent proliferation of mobile electronic devices, there has been a trend towards integration and unification of as much functionality as possible into a single device. We seldom see anyone carrying around a cell phone, PDA, digital camera, MP3 player, and pager all at once. Despite all of these being potentially useful devices, it is simply too troublesome to bother with each individual device. The hassles of charging the batteries, carrying them around, keeping track of where they are, far outweigh the benefits they provide. Instead, performance and features are often sacrificed for convenience and ease of use. For this reason, we place special emphasis on requiring as little additional hardware as possible.

5.2 Related Work

A number of systems providing support for location aware computing have already been created. Here, we briefly describe a few of the different approaches and analyze

them in the context of our requirements.

5.2.1 ActiveBat

The Active Bat sensing system[14, 31] pioneered the use of sound to provide location estimates in indoor environments. In the Active Bat system, transmitters are attached to mobile devices, and periodically emit a combination of RF and ultrasonic signals. An array of specially tuned and calibrated receivers placed in known positions around the environment record the emitted signals and relay them to a central station. To compute the distance from a receiver to the mobile device, the time-of-flight difference between the RF and ultrasonic signals is measured and multiplied by the speed of sound. Distance measurements to a mobile device from each receiver are collected and used to estimate the true position of the device to centimeter-level accuracy.

Since ultrasound does not pass through walls, glass, or other partitions, the Active Bat system adheres to our human notion of space and locality. If we are in a closed room, the system will never erroneously estimate that we are on another floor, standing outside the building, or even in an adjacent room. By the same token, if there is no direct path for sound to travel from the tracked object to the receivers (e.g. if the object is in a closed drawer), then it is much more difficult to provide an accurate location estimate.

When the Active Bat system was created, privacy in ubiquitous computing was not as significant a concern as it is today. This is reflected in its design, which allows a centralized system to have total knowledge of the locations of its participants, while the tracked objects themselves do not have this knowledge. In order for a mobile device to obtain knowledge of its own location, the centrally controlled system must transmit that information to the device.

Another drawback to Active Bat is that it does not scale well, both in terms of price and performance. As the number of tracked objects increases in a region,

their is greater contention for the RF and ultrasonic channels on which the system operates. Collisions can happen more frequently, degrading the performance of the system. The specialized hardware is currently prohibitively expensive to deploy and maintain on a large scale. Although the hardware costs could be brought down by mass manufacturing and integration with other devices, each receiver still needs to be carefully calibrated and maintained.

5.2.2 Cricket

The Cricket[27] location system, like Active Bat, uses time-of-flight measurements computed from the difference in arrival times of RF and ultrasonic signals to obtain distance measurements from one device to another. The main difference is that instead of the tracked objects actively transmitting information, it is the statically positioned beacons that actively transmit information in the Cricket system. Beacons can be programmed with a location coordinate, which is relayed during each of its transmissions. The mobile devices carry passive listeners that can receive the signals emitted by the beacons and perform distance and location estimates individually. By using the same technology and general methods as Active Bat, Cricket is also able to achieve centimeter-level accuracy.

From the very beginning, Cricket was designed to afford its users a sense of privacy. By creating the system in such a way that mobile devices never actively transmit, they made it much more difficult to track a user without permission. In some situations, such as an environment where the users trust the centralized system, it may be desired or beneficial to track users centrally, which can be accomplished by having mobile devices transmit their information individually.

One advantage of Cricket over Active Bat is that it is highly scalable with respect to the number of tracked objects it can support in a single space. Since additional listeners do not use any shared resources, there is no issue of channel contention or

signal collision. It becomes possible to support a large number of listeners in close proximity without any performance degradation. Like Active Bat, however, Cricket requires specialized hardware that is currently on the order of hundreds of dollars per listener and beacon, making it unaffordable for the general public.

5.2.3 802.11 signal strength

Many recent efforts in providing location aware services have focused on 802.11 technologies. The first of these, RADAR[4] operates on the same principle of having many beacons providing service to mobile devices. 802.11 base stations, which normally serve as network access points, are used as beacons and any 802.11 device capable of measuring signal strength can take advantage of the location aware services. Instead of using time-of-flight measurements to provide distance measurements, a RADAR device estimates its position by comparing base station signal strength measurements with an internal *radio map*. The radio map is constructed by obtaining signal strength measurements at every position in the desired area of coverage, presumably done by a system administrator at installation time. In typical office environments, RADAR is able to estimate a user's position to within approximately 3-meters. Recent progress[21, 22] in using Bayesian inference has improved this to meter-level accuracy.

In the original implementation, mobile devices broadcasted signals, which were collected and measured by base stations. In this form, RADAR closely resembled Active Bat, and consequently afforded no privacy for the individual user. It is certainly possible to shift the transmitter and receiver roles so that the mobile device acts as a passive listener measuring signals emitted by base stations.

802.11 signal strength based methods have the advantage that they can take advantage of widely deployed hardware that is already used for other purposes. In many cases, the base stations are already in place and can be used without modification.

The primary disadvantages are that such systems do not adapt well to dynamic environments, and that building the radio map is a significant undertaking. If a single base station moves, then hundreds of square meters of the radio map are affected. Ultimately, it seems impractical to provide location aware services on a large scale using radio maps and signal strength measurements.

5.2.4 Bluetooth

We are not the first to propose Bluetooth as a location tracking infrastructure. Anastasi et al[3] used statically positioned Bluetooth devices to constantly scan for other Bluetooth devices in the vicinity. Detected devices were then entered into a central database which was used to track the location of all moving Bluetooth devices. This approach is cost effective in that it makes use of readily available hardware. The disadvantages are that it allows for no privacy and requires specialized software on the trackers as well their connectivity to a centralized database. Numerous trackers constantly scanning for nearby devices is also wasteful of radio resources and can interfere with other technologies, such as 802.11, operating in the 2.4 GHz spectrum.

The Local Positioning Profile(LPP) [13] defines a standardized protocol for Bluetooth devices to exchange positioning data. A device whose location is known runs a Local Positioning (LP) Server, to which other Bluetooth devices can connect. LP Clients can request positioning information from LP Servers, which may be derived from preset configurations, GPS data, cellular data, or automatically generated, and then infer their own positions. The primary purpose of the LPP is to provide a means for devices to exchange data, and leaves much room for techniques to be developed for determining a device's actual position given the position of other devices. The LPP also does not take privacy into consideration, as it is required for both client and server to have knowledge of both Bluetooth device addresses, allowing a well-coordinated network of LP servers to track clients as they issue requests.

5.3 The Bluetooth location system

In this section, we describe the deployment of our infrastructure and analyze the basic technique for scanning for and discovering location beacons. We also show that using multiple co-located Bluetooth devices improves the reliability and robustness of our system.

The infrastructure for our system consisted of 30 D-Link DBT-120 USB Bluetooth adapters, purchased for US$30 each. Research groups in our building were asked to spare a single USB port in their computers. The beacons were then placed in computers approximately every 10 meters on six different floors. The Bluetooth device for each beacon was given the user friendly name of the form "OKN-*building*-*room*", where *building* and *room* indicated the building and room number of the beacon. The only software installed on the host machines were the device drivers. On average, configuring a machine to host one of our beacons took less than three minutes. The most time consuming part of the deployment was actually tracking down the system administrators for the machines we wanted to use, and obtaining their permission[1].

Client software was written for several types of locators and used to test the effectiveness of the infrastructure. A Linux client was used on laptops, desktops, and a HP iPAQ 5550. A C++ Symbian client was used on Nokia 6600 cellular phones. To choose an algorithm for determining the position of a locator, a series of experiments were conducted to analyze the process of device detection and communication in Bluetooth.

[1]Ironically, despite being a hub of research in computer science, many lab members had no idea what Bluetooth is and does, or how it would affect their computer. One researcher whose computer was not hosting a beacon expressed sincere concern that nearby Bluetooth devices would make his workstation vulnerable to crackers. Coincidentally, this incident occurred in the Theory of Computation research area.

5.3.1 Detecting beacons

A Bluetooth device inquiry, which is a broadcast of a predefined sequence of bits while hopping channels pseudorandomly, is used to detect nearby beacons. A response to an inquiry consists of a 48-bit Bluetooth address, a 24-bit device class field describing the device, and some synchronization information.

The ideal beacon would always listen for the inquiry sequence and respond almost immediately upon detection. A number of factors can cause a beacon to either not respond or to not detect an inquiry.

- Electromagnetic noise and interference with other devices in the 2.4 GHz range may hinder communications.

- A beacon cannot listen for an inquiry all the time. It must allocate time to listen for connection requests, and to participate in active connections.

- Upon first detecting a device inquiry, a beacon will always enter a backoff stage, in which it idles for 0 to 0.33 seconds randomly.

- A beacon, while listening for inquiries, will listen on one of 32 predefined channels at a time. During an inquiry, the locator will inquire on half of these channels for 2.56 seconds, switch to the other half for another 2.56 seconds, and then alternate two more times. Consequently, it is possible that a locator will not even inquire on the same channel on which a beacon is listening on for at least 2.56 seconds.

While nothing can be done about noise and interference from other radio sources[2], something can be done to improve the beacon detection speed. As can be seen from Figure 5-1, it can take 10 seconds for a locator to detect a beacon, and we have observed times when it has taken even longer.

[2]This is a problem addressed in version 1.2 of the Bluetooth specification, which allows for adaptive frequency hopping to avoid channels being used by co-located, interfering devices. However, existing Bluetooth 1.1 and 1.0 devices are not able to take advantage of this ability.

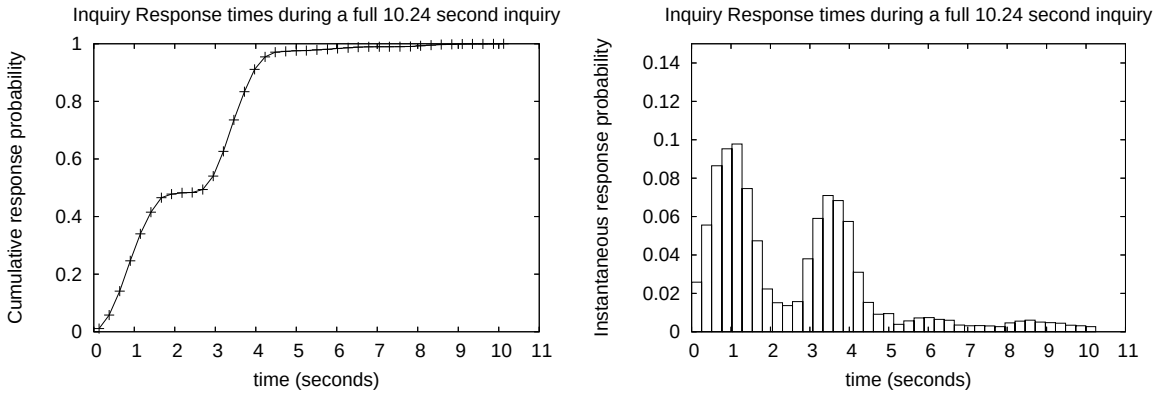


Figure 5-1: The locator divides the 32 inquiry channels into two disjoint sets of channels, say S and T, If a beacon happens to be listening on a channel in S, then it will be discovered in the first 2.56 seconds. Otherwise, it will not be discovered at least until the locator switches to set T. The graph on the left is the cumulative success while the one on the right is the instantaneous.

The Bluetooth specification is optimized for the situation where many devices are all in the same vicinity. Device inquiry is especially slow because of the pessimistic backoff algorithms used to minimize collisions. The recommended duration for a device inquiry is 10.24 seconds[8], which is longer than many applications can tolerate. As shown in Figure 5-1, however, more than half of the detected devices are detected in the first 2.56 seconds of the inquiry.

5.3.2 Two heads are better than one

To reduce the average time to detect a beacon, we placed two Bluetooth USB devices in the PC. The locator needs to wait for a response from only one beacon. Our experiments show that the beacons responded independently of each other, providing an ideal increase in response rate. The locator was placed approximately 8 meters from two co-located beacons, with a closed door, some wooden office furniture, and a metal filing cabinet in between the locator and the beacons. Additionally, a host of other active Bluetooth and WiFi devices were operating in the vicinity. By adding a Bluetooth device to the beacon, we significantly increased its tolerance for noise and

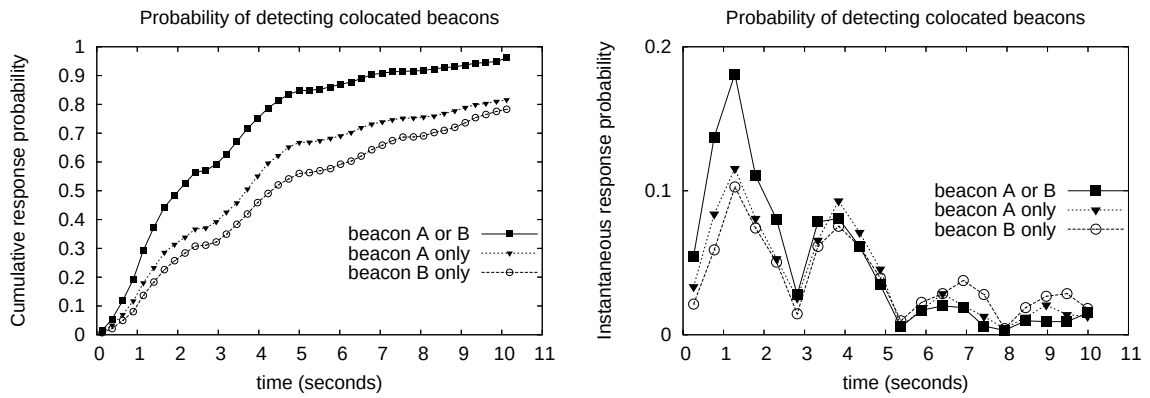


Figure 5-2: When beacons A and B are in the exact same location, the locator needs only to hear a response to an inquiry from either one.

interference. Unfortunately, this also has the effect of doubling the cost of a single beacon. These results are shown in Figure 5-2.

Similarly, when the locator was equipped with two Bluetooth devices, and performed inquiries with both devices simultaneously, location beacons were also discovered more quickly. These results are summarized in Figure 5-3. In order to achieve the improved response rate, however, the discoverability of the locator's Bluetooth devices had to be disabled, otherwise, performance actually decreased as the locator began responding to its own inquiries. Response rate of a single Bluetooth device in range was still not as fast as when a beacon is equipped with two Bluetooth devices and the locator with one. We attribute this to the backoff algorithm used during the inquiry scan process.

5.3.3 Adjusting Page Timeout

A beacon response to a device inquiry does not provide much to the locator other than the beacon's address. To obtain more information, such as the location of the beacon, the locator can perform a remote name request or establish a connection to the beacon using a transport protocol such as L2CAP or RFCOMM. A remote name request creates a temporary connection to retrieve a 248-byte data string that

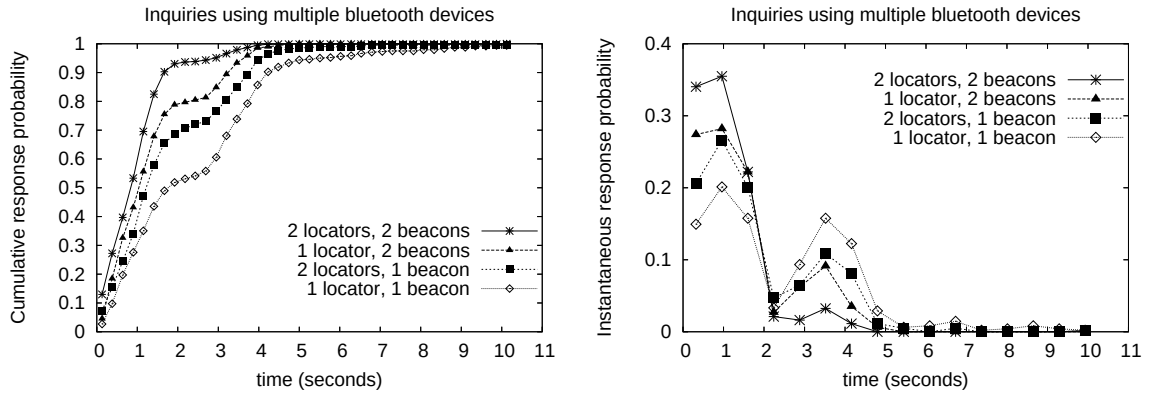


Figure 5-3: When a locator is equipped with multiple Bluetooth devices and uses them to perform simultaneous inquiries, devices in range are detected much more rapidly. Using two co-located beacons is even faster.

is usually interpreted as the user-friendly name of the target device, whereas a higher level connection allows the exchange of arbitrary data. Both of these actions involve *paging* the beacon, a time consuming process similar to a device inquiry, and reveal the locator's Bluetooth address. As noted earlier, a well coordinated network of beacons would then be able to track a locator using only its address.

The Bluetooth specification recommends 5.12 seconds as the timeout when paging[3] a remote device. Since paging is the most time consuming part of forming a connection, setting a page timeout effectively sets a connection timeout. To see if extending the page timeout significantly increased the chance of connecting to a device, we performed numerous remote name requests with the page timeout set to 20.48 seconds. Figure 5-4 shows that if the name of a remote device was resolved during the 20.48 second time period, it was resolved in the first 5.12 seconds 87% of the time. Note that some Bluetooth implementations, such as BlueZ for Linux, raise the default timeout significantly to increase the chance of successfully paging on the first try.

[3]Note that the specification recommends a time period twice as long for inquiries.

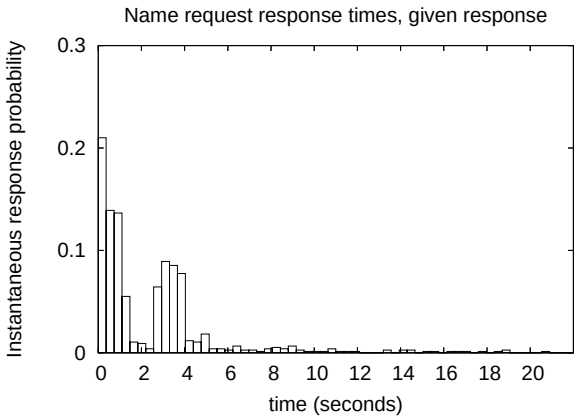


Figure 5-4: The name request process is similar in nature to the inquiry process, but uses a different set of 32 channels. If the name of a device is resolved, it is usually done so during the first 5.12 seconds of the name request - the amount of time it takes for the locator to iterate through both trains A and B.

5.3.4 Determining beacon position

The position of a detected beacon must be known in order for the beacon to be useful to the locator. This information could be stored on the beacon and replayed to querying locators[4] or the locator could maintain a lookup table storing the positions of all known beacons. The observations made from the above experiments suggest two different algorithms that a locator could use to efficiently detect and determine the position of nearby beacons. In both methods, the locator maintains a software cache that maps beacon addresses to locations. The first method incrementally builds the cache by querying unrecognized beacons, and the second method assumes the cache has already been built and will never change. Remembering that beacons are configured to have Bluetooth user-friendly names that correspond to beacon position, we can use remote name requests to query a beacon for its position.

- Method A Perform a device inquiry for 10.24 seconds. After the first 2.56 seconds, the inquiry is canceled as soon as a device is discovered whose name is

[4]This is exactly what LPP[13] is designed to do - provide a standardized method for the transfer of positioning information from the beacon to the locator. However, at the time of writing, LPP was still in draft form and we found other methods much simpler.

not in the software cache. Remote name requests with a page timeout of 5.12 seconds are sent to all unrecognized beacons and the responses cached. Repeat.

- Method B A software cache containing all known Bluetooth beacons in the building is is preloaded into memory. The locator repeatedly performs device inquiries, and never issues any remote name requests. Unrecognized Bluetooth devices are ignored. Repeat.

An experiment was conducted to evaluate these two methods along with a third method, which we called the naive method.

- Method C (naive) Perform a device inquiry for 10.24 seconds. Remote name requests with a page timeout of 20.48 seconds are sent to all unrecognized beacons and the response cached. Repeat.

Three locators, each using one of the three methods, were carried around the building for forty minutes, collecting localization data. Our results, summarized in Table 5.1, show that method B was by far the fastest. Out of 34 beacons detected by locator B, it was either the first or only locator to detect the beacons 30 times. Locator A consistently detected beacons faster than C, but slower than B, and locator C was usually the last to detect a beacon.

For each beacon that a locator discovered, we averaged the difference between the time that it was discovered and the time that it was first discovered by any of the other two locators. We found that on average, method C was 19.5 seconds slower to detect a beacon than the first method (not necessarily B), method A was 8.5 seconds slower than the first method, and method B had only a 0.9 second delay on average.

Method B had the advantage of not needing to perform any remote name requests at all. In areas dense with Bluetooth devices, the algorithm could safely ignore unrecognized devices. Additionally, as it doesn't establish any connections, method

locator	only to detect	first to detect	second to detect	third to detect
A	0	3	19	2
B	10	20	3	1
C	0	2	3	16

Table 5.1: Locator B was almost always the first or only locator to detect a beacon. Locator A was usually second to detect a beacon, and locator C was usually last.

B can guarantee complete anonymity. The only disadvantage is that the software cache must be obtained from somewhere else.

We find that method A is useful in situations where positioning is desired in an unfamiliar environment, where the software cache for method B could not be updated before entering the area. In a known environment, however, method B is faster in all respects. In no circumstances is method C to be preferred.

5.3.5 Estimating locator position

Once a locator has detected one or more beacons and determined their positions, the locator can then estimate its own position. The simplest approach is to conclude that the locator is somewhere within the geometric intersection of the areas in range of each detected beacon (see Figure 5-5). In this way, the precision with which a locator can determine its position is directly related to the number of beacons it detects and the placement of each beacon.

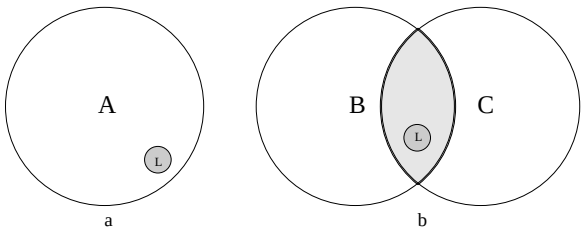


Figure 5-5: a) When the locator L can only detect one beacon, it can only conclude that it is somewhere within the circle b) When two beacons are detected, much greater resolution is achievable, and the locator can conclude it is somewhere in the shaded region

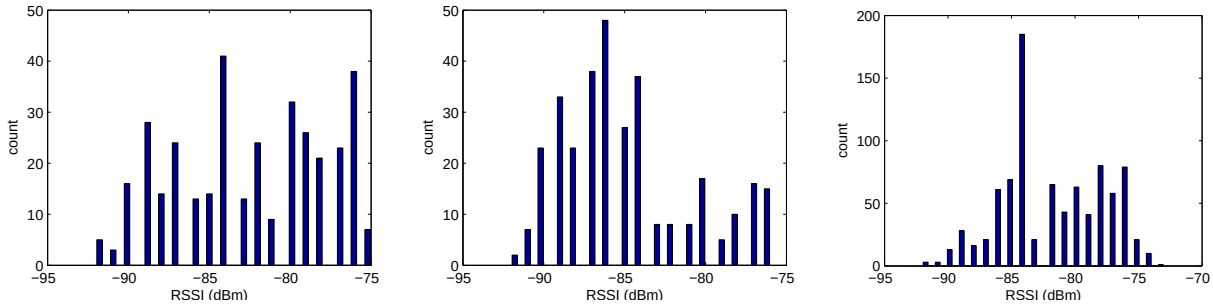
5.3.6 Signal Strength

The method used in the previous section to estimate the locator position is unsatisfying, in part because of its simplicity, and in part because of its low resolution. The Bluetooth 1.2 specification[9] supports device inquiries that report signal strength of discovered devices. Given the intuition that signal strength and distance from a device share an inverse relationship, it is natural to ask if signal strength could be used to refine an estimate of the locator's position.

The radio maps used by 802.11 location systems such as RADAR[4] consist of signal strength measurements made at a number of different locations. Since 802.11 and Bluetooth both operate in the same frequency band, it seems reasonable to conclude that the same inference techniques that work for 802.11 systems also work for Bluetooth. The major drawbacks to these methods is that they make very strong assumptions about the RF conditions in the areas mapped, and that they require a labor intensive off-line training process to initialize the radio maps. These methods are highly sensitive to changes in network topology; the maps must be updated every time some furniture is moved in order to stay accurate.

A weaker approach to take is to compute a distance estimate to each beacon in range based on the signal strength measured from that beacon, and to then compute a position estimate from those distance estimates. This is a well studied problem in radio literature[11]. In theory, by obtaining three distance measurements, trilateration can be used to calculate locator position. More than three distance measurements yields an overdetermined system of linear equations that can be solved with least squares techniques.

In practice, however, signal strength is a poor choice to use for estimating distance. Especially indoors, a multitude of factors affect signal strength measurements. Large shadowing and high multipath effects can amplify or severely diminish a signal. Radio signals in the 2.4 GHz frequency range that Bluetooth operates in are highly



(a) 3 meters (b) 6 meters (c) 9 meters

Figure 5-6: Signal strength readings taken at three different distances. In all three cases, the distribution is highly varied and non-Gaussian.

susceptible to occluding structures such as furniture, walls, water, and people (since people are mostly water). The result is a signal with significant non-Gaussian noise that is poorly correlated with distance. Figure 5-6 shows three histograms of signal strength measurements for several beacons at three different distances. The beacons were all in different locations, and the measurements were collected over a period of days and then aggregated. The distances were arbitrarily chosen.

Due to the non-Gaussian structure of the signal strength noise, standard noisy inference techniques such as Kalman filtering are not well suited for estimating distances. Additionally, it does not seem likely that any probabilistic inference methods can provide an accurate estimate of the distance to a single beacon using only signal strength measurements for that beacon. Although probabilistic methods are well suited for use in radio maps, as described earlier, there is too much noise in an individual signal strength measurement to obtain an precise measurement.

5.4 Discussion

By taking advantage of an existing computational infrastructure, we were able to deploy our system on a building-wide scale with a minimal investment of capital

and labor. We did not need to mount any hardware in special places, run cables, or displace any existing equipment or furniture. The only physical change to the environment needed was the addition of 30 small USB devices. Locators did not require specialized hardware, and could immediately determine their position to an accuracy of 3-10 meters, depending on beacon density.

Unfortunately, we underestimated the high maintenance costs associated with relying on other research groups to host our beacons. Host machines were frequently moved, reformatted with a new operating system (without the Bluetooth device drivers), or decommissioned. Without a system to notify us of these changes, we were not able to efficiently determine when beacons were disabled. After six months of operation, fewer than half of the original beacons were still operating. This problem was exacerbated due to the fact that each research group administered its own machines, each of which might be running a different operating system (the most common being OS X, Windows 2000/XP, Debian Linux, and Fedora Linux).

A second disadvantage was that beacons could only be placed in areas where computers with USB support were already running. This was not an issue in research areas, where there is a high density of desktop computers, but quickly became a problem when we wanted to expand our infrastructure to areas such as the cafeteria, classrooms, and bathrooms. In those areas, the cost of deploying a single beacon rose by an order of magnitude.

The crippling factor was the system's reliance on host computers, which were only necessary to provide power to the beacons and initialize them. We noted that host-beacon communication was only required during the initialization process, and that at all other times, the host computer was merely an expensive power source.

We are in the process of constructing a second infrastructure to address the problems encountered in the first system. In the second version of our system, we continue to use USB Bluetooth adapters, which are now widely retailed for $US20 each. In

addition to the Bluetooth adapters, we purchased 30 four-port USB hubs with external AC adapters, available for $US10 each. To initialize a beacon, shown in Figure 5.4, we first connect it to the USB hub, which is in then connected to a typical 120V power outlet. The hub is connected to a laptop computer, which automatically initializes the beacon and allows it to be discovered by the locators. In our system, a single laptop running Debian Linux was used to initialize all the beacons. In the laptop's default configuration, connecting and disconnecting the USB hub was sufficient to initialize the beacon. A small status light on the D-Link adapter automatically indicates whether it is initialized or not.

A system built in this manner no longer relies on host computers to be present. The only sub-infrastructure it requires is an electrical power system, which is readily available in all places it would be deployed. Beacons can be deployed in a much wider variety of locations in a much simpler fashion that does not require significant centralized coordination.

The major disadvantage of this system is its susceptibility to power failure. If a beacon loses power, even momentarily, then it must be re-initialized. We currently regard this feature as unplanned maintenance, similar to technical difficulties with elevators or access card readers. Since our building rarely loses electrical power more than once or twice a year, we do not view this as a significant problem. In extremely large scale deployments spanning multiple buildings and city blocks, a single power failure could require a significant recovery time. If the system were to be deployed on such a large scale, however, it would be relatively inexpensive to design and manufacture a self-initializing beacon.

Figure 5-7: A beacon in the current version of our system consists of a USB hub (top right), AC adapter (left), and USB Bluetooth adapter (bottom right).

5.5 Chapter Summary

This chapter presents an infrastructure for location aware computing that is inexpensive and trivial to deploy. Previous systems rely on specialized hardware, are prohibitively expensive to deploy and maintain, or do not scale well with the area serviced. Our system uses commodity electronics that are already widely available, and easily scales with size. Locator devices such as cell phones and PDAs often do not need any hardware modification at all to take advantage of the system. With sufficient beacon density, a locator is able to determine its position to room-level accuracy, or approximately 3-meters.

The first version of the system suffered from high maintenance and poor reliability. Relying on PCs to host the beacons used in the infrastructure meant the beacons were disabled or rendered ineffective as soon as a PC was turned off, reformatted, or physically moved. These problems were addressed and a second system, which is not as susceptible to these issues, is currently being deployed.

Appendix A

Installing PyBluez

This appendix describes how to obtain and install PyBluez.

Requirements

- GNU/Linux Operating System
- Bluez libraries $\geq$ v 2.11
- C development and compilation environment
- Python $\geq$ v 2.3
- `distutils` Python extension module

Obtaining PyBluez

PyBluez is distributed at `http://org.csail.mit.edu/pybluez` The latest versions and updates can be obtained on the downloads page linked at that web site. A direct link to the latest version of PyBluez (0.2 at the time of writing this document) is

`http://org.csail.mit.edu/pybluez/release/pybluez-src-latest.tar.gz`

Building PyBluez

After downloading the latest distribution, unpack the `.tar.gz` file and issue the command

```
# python setup.py build
```

This will invoke the Python `distutils` module and automatically configure and compile PyBluez.

Installing PyBluez

Installing PyBluez onto a host machine requires superuser privileges. To install the compiled module, issue the following command

```
# python setup.py install
```

This will ensure that the PyBluez extension module has been built correctly and install it as a third-party extension module on the host system. To test the installation, invoke the Python interpreter and import the `bluetooth` module. For example,

```
# python
Python 2.3.4 (#2, Dec  3 2004, 13:53:17)
[GCC 3.3.5 (Debian 1:3.3.5-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import bluetooth
```

If the import completes without any exceptions being raised, then PyBluez was successfully installed and is ready for use.

Bibliography

[1] Lauri Aalto, Nicklas Göthlin, Jani Korhonen, and Timo Ojala. Bluetooth and WAP push based location-aware mobile advertising system. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 49–58, 2004.

[2] AIM Inc. *Radio Frequency Identification - RFID. A Basic Primer v 1.1*, September 2004.

[3] G. Anastasi, R. Bandelloni, M. Conti, F. Demastro, E. Gregori, and G. Mainetto. Experimenting an indoor bluetooth-based positioning service. In *Proceedings of the International Conference on Distributed Computing Systems Workshops*, pages 480–483, May 2003.

[4] P. Bahl and V. N. Padmanabhan. Radar: An in-building RF based user location and tracking system. In *Proceedings of IEEE INFOCOM 2000*, pages 775–784, March 2000.

[5] Louise Barkhuus and Anind Dey. Location-based services for mobile telephony: a study of users' privacy concerns. In *Proceedings of Interact 2003*, pages 709–712, 2003.

[6] bluejackQ. *bluejackQ*, July 2004. `http://www.bluejackq.com`.

[7] Bluetooth SIG. *Assigned Numbers - Bluetooth Baseband*, 2003. `https://www.bluetooth.org/foundry/assignnumb/document/baseband`.

[8] Bluetooth Special Interet Group. *Bluetooth Profile, Specification of the Bluetooth System, Version 1.1*, February 2001.

[9] Bluetooth Special Interet Group. *Bluetooth Profile, Specification of the Bluetooth System, Version 1.2*, November 2003.

[10] Robert Grover Brown. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons, third edition, 1997.

[11] James J. Caffery. *Wireless location in CDMA cellular radio systems*. kluwer international series in engineering and computer science. Kluwer Academic, Boston, Massachusetts, 2000.

[12] Ivan. A. Getting. The global positioning system. *IEEE Spectrum*, 30(12):36–47, Dec 1993.

[13] Bluetooth Special Interest Group. *Local Positioning Profile, Version 0.95*, July 2003.

[14] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. *Wirel. Netw.*, 8(2/3):187–197, 2002.

[15] Jason I. Hong and James A. Landay. An architecture for privacy-sensitive ubiquitous computing. In *MobiSYS '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 177–189, New York, NY, USA, 2004. ACM Press.

[16] In-Stat/MDR. *Bluetooth 2004: Poised for the Mainstream*, July 2004. `http://www.instat.com/r/nrep/2004/IN0401211MI.htm`.

[17] The Institute of Electrical and Electronics Engineeers, Inc. *IEEE Standard 802.11 - Wireless LAN Medium Access Control (MAC) and Physical Layer (PHS) specifications*, 1999.

[18] Internet Engineering Task Force. *A UUID URN Namespace*, 2004. `http://www.ietf.org/internet-drafts/draft-mealling-uuid-urn-05.txt`.

[19] Internet Engineering Task Force. *IETF RFC Page*, May 2005. `http://www.ietf.org/rfc.html`.

[20] Eija Kaasinen. User needs for location-aware mobile services. *Personal and Ubiquitous Computing*, 7(1):70–79, 2003.

[21] Andrew M. Ladd, Kostas E. Bekris, Algis Rudys, Lydia E. Kavraki, Dan S. Wallach, and Guillaume Marceau. Robotics-based location sensing using wireless ethernet. In *Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 227–238, 2002.

[22] Andrew M. Ladd, Kostas E. Bekris, Algis Rudys, Dan S. Wallach, and Lydia E. Kavraki. On the feasibility of using wireless ethernet for indoor localization. volume 20, pages 555–559, 6 2004.

[23] Brent A. Miller and Chatschik Bisdikian. *Bluetooth Revealed*. Prentice Hall, Upper Saddle River, NJ, second edition, 2002.

[24] Michael Minges. Is the internet mobile? Measurements from Asia-Pacific. In *Proceedings of the International Telecommunications Societ Asia-Australian Regional Conference*, June 2003.

[25] M. Nilsson and J. Hallberg. Positioning with Bluetooth, IrDA, and RFID. Master's thesis, Luelå University of Technology, 2002.

[26] David C. Plummer. *An Ethernet Address Resolution Protocol*, November 1982. `http://www.ietf.org/rfc/rfc0826.txt`.

[27] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 32–43, 2000.

[28] Einer Snekkenes. Concepts for personal location privacy policies. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 48–57. ACM Press, 2001.

[29] Guido van Rossum. *Extending and Embedding the Python Interpreter*. Python Software Foundation, 2005. `http://docs.python.org/ext/ext.html`.

[30] Guido van Rossum. *Python/C API Reference Manual*. Python Software Foundation, 2005. `http://docs.python.org/api/api.html`.

[31] R. Want, A. Hopper, V. Falc ao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102, Jan 1992.

[32] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265, September 1991.