

# Cricket v2 User Manual

Cricket Project  
MIT Computer Science and Artificial Intelligence Lab  
Cambridge, MA 02139  
<http://cricket.csail.mit.edu/>

July 2004



# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
1.1	System Overview . . . . .	9
1.2	Quick start . . . . .	11
1.2.1	Set up a communication terminal . . . . .	11
1.2.2	Communicate with the Cricket unit . . . . .	12
1.2.3	Configure a Cricket unit to be a beacon . . . . .	13
1.2.4	Test distance measurements . . . . .	13
1.3	Installing the Cricket embedded software image . . . . .	13
1.3.1	With TinyOS . . . . .	13
1.3.2	Without TinyOS . . . . .	14
1.4	Overview of this manual . . . . .	15
<b>2</b>	<b>Command Interface and Troubleshooting</b>	<b>17</b>
2.1	Cricket Hardware Details . . . . .	17
2.1.1	Diagnostic LEDs . . . . .	18
2.1.2	Test Switch . . . . .	18
2.1.3	Powering Crickets . . . . .	19
2.2	Serial Port Command Interface . . . . .	19
2.2.1	cricketd . . . . .	19
2.2.2	Cricket Command Interface . . . . .	21
2.2.3	Error Codes . . . . .	21
2.2.4	Default values . . . . .	21
2.2.5	Run mode command (MD) . . . . .	23
2.2.6	Configuration status command (CF) . . . . .	23
2.2.7	Get serial ID (ID) . . . . .	23
2.2.8	Get/Put space ID (SP) . . . . .	24
2.2.9	Get software version (VR) . . . . .	24
2.2.10	Distance to beacon (DB) . . . . .	24
2.2.11	Duration (DR) . . . . .	24
2.2.12	The uncorrected time of flight (TM) . . . . .	24
2.2.13	Get/Put coordinates command (PC) . . . . .	25
2.2.14	Get/Put the minimum and maximum beacon interval time (SL) . . . . .	25
2.2.15	Get/Put the ultrasound maximum time-of-flight (UL) . . . . .	25

2.2.16	Get/Put the software offset (OF)	26
2.2.17	Get system time (TS)	26
2.2.18	Get/Put temperature sensors (TP)	26
2.2.19	Test switch status (TB)	27
2.2.20	Get/Put display units (UN)	27
2.2.21	Get the beacon listing (LS)	27
2.2.22	Save settings to flash (SV)	28
2.2.23	Load settings from flash (LD)	28
2.2.24	Get/Put Output format (OU)	28
2.2.25	Customize output format (CO)	29
2.3	Troubleshooting and Deployment Hints	29
2.3.1	Problem: The listener or the beacon does not respond to any command	29
2.3.2	Problem: The listener does not report any events	30
2.3.3	Problem: The listener returns erroneous distances	30
2.3.4	Problem: The beacon blinks but does not respond to serial commands	30
2.3.5	Problem: The listener associates itself with the “wrong” space identifier	31
<b>3</b>	<b>Sample Application</b>	<b>33</b>
3.1	Setup	33
3.2	Source Code	33
3.3	Beacon Placement	34
3.4	Setting Parameters	34
3.5	Launching and Running BeaconConfigDemo	36
3.5.1	Launching cricketd	36
3.5.2	Launching BeaconConfigDemo	36
3.5.3	Running BeaconConfigDemo	37
3.5.4	Troubleshooting	40
<b>4</b>	<b>Developing Cricket Applications in Java</b>	<b>43</b>
4.1	Requirements	43
4.2	Architecture	43
4.3	Compiling Clientlib	44
4.4	Clientlib API	45
4.4.1	The cricketdaemon.clientlib.ServerBroker Class	45
4.4.2	The cricketdaemon.clientlib.Callback Class	47
4.4.3	The cricketdaemon.clientlib.data.CricketData Class	47
4.4.4	The cricketdaemon.clientlib.data.BeaconRecord Class	48
4.4.5	The cricketdaemon.clientlib.data.DistanceStat Class	49
4.4.6	The cricketdaemon.clientlib.data.Position Class	50
4.4.7	The cricketdaemon.clientlib.data.Sample Class	50
4.5	Using Clientlib: An Example	51
4.5.1	Source Code	51
4.5.2	Compiling ClientlibExample	54

4.5.3 Running ClientlibExample . . . . . 55



# List of Figures

1.1	A Cricket hardware unit; this unit can function as either a beacon or a listener under software control, and can also be used in a more symmetric way as both listener and beacon. . . . .	10
1.2	Example deployments of Cricket beacons. Multiple beacons may advertise the same space identifier if they are in the same space, but each beacon has a different position coordinate consistent with its location in space. . . . .	10
2.1	Cricket v2 hardware components and layout. . . . .	18
2.2	Serial port command API. . . . .	22
2.3	Incorrect placement of beacons to for boundary detection between two spaces. . . .	31
2.4	Correct beacon placement. . . . .	31
4.1	Software Architecture . . . . .	44
4.2	Clientlib Architecture . . . . .	45



# Chapter 1

## Introduction

This document describes the key features of **version 2 (v2)** of the **Cricket indoor location system**.<sup>1</sup> It provides information to develop Cricket applications and maintain a Cricket installation. You will find this document useful if you plan to:

- Write location-aware applications with Cricket to run on handhelds, laptops, and desktop under Linux or Windows.
- Deploy and maintain a Cricket system.

You will also find the document useful if you plan to write location-aware embedded wireless sensor computing applications on the Mote platform. Writing such applications will not be difficult because the Cricket embedded software is written in TinyOS [6], the software platform for the Motes.<sup>2</sup> In addition, you will find this document useful if you want to make modifications to the Cricket embedded software.

The best way to use this document is in conjunction with the Cricket hardware and software, so you can try things out while reading. Details on how to get the hardware and software are at <http://cricket.csail.mit.edu/>. That web page has pointers to commercially available Cricket hardware units from Crossbow Technologies (<http://www.xbow.com/>).

We start with a quick overview of the Cricket architecture. A more detailed technical description for Cricket v1 is in [5]. A more recent paper describing experiences with Cricket v1 and the design decisions made in v2 is in [1].

### 1.1 System Overview

Cricket is an indoor location system. It provides two forms of location information—**space identifiers** and **position coordinates**—and can be as accurate as between 1 cm and 3 cm in real deployments. Space identifiers are user- or application-specified names associated with spaces such

---

<sup>1</sup>In the rest of this document, when we refer to “Cricket” without a version number, we mean v2. Any references to v1 or other versions will be made explicitly.

<sup>2</sup>However, the current version of the Cricket v2 embedded software (2.0) does not expose the full TinyOS API required to easily write such applications. That will change in future versions of the Cricket software.



Figure 1.1: A Cricket hardware unit; this unit can function as either a beacon or a listener under software control, and can also be used in a more symmetric way as both listener and beacon.



Figure 1.2: Example deployments of Cricket beacons. Multiple beacons may advertise the same space identifier if they are in the same space, but each beacon has a different position coordinate consistent with its location in space.

as rooms or parts of rooms. The position coordinates are  $(x, y, z)$  Cartesian coordinates in some coordinate system.

The most common way to use Cricket is to deploy actively transmitting *beacons* on walls and/or ceilings, and attach *listeners* to *host devices* (handhelds, laptops, etc.) whose location needs to be obtained. See Figures 1.1 and 1.2.

Users or administrators configure beacons with space identifiers, and optionally with position coordinates (one can also use an auto-localization algorithm to assign position coordinates to beacons; we provide a simple one with the Cricket v2.0 software distribution). Each beacon periodically broadcasts its space identifier and position coordinates on a radio frequency (RF) channel, which listeners within radio range can receive. Each beacon also broadcasts an ultrasonic pulse at the same time as the RF message. Listeners that have line-of-sight connectivity to the beacon and are within the ultrasonic range will receive this pulse. (The exact values of the RF and ultrasonic ranges for the Cricket hardware are mentioned in the next chapter.)

Because RF travels about  $10^6$  times faster than ultrasound, the listener can use the *time difference of arrival* between the start of the RF message from a beacon and the corresponding ultrasonic pulse to infer its distance from the beacon. Every time a listener receives information from a beacon, it provides that information together with the associated distance to the attached host using

the API described in the next chapter. The listener (or software running on the host device) infers its position coordinates based on distances from multiple beacons whose positions are known, and software running on the host device can associate itself with the space corresponding to the nearest beacon.

The spaces advertised by Cricket beacons may be demarcated by physical boundaries such as walls, or may be virtual (*e.g.*, different parts of a room may correspond to different spaces). Cricket is designed to accurately demarcate virtual spaces that don't have any walls between them. Because ultrasound does not travel through walls, Cricket can easily demarcate spaces separated by walls.

Cricket scales reasonably well with large numbers and high densities of devices in close proximity (*e.g.*, many devices in the same room). Cricket is relatively easy to set up and start using, and relatively straightforward to maintain.

Because listeners need not transmit any information, Cricket makes it harder to track users if location privacy is a desired goal. This property of Cricket makes it different from some other indoor location systems like the Active Badge [7] and Active Bat [4] systems, in which tracking users is inherent.

Finally, note that the Cricket infrastructure is quite flexible, in that you can run the beacon on a moving device, and also run a more symmetric Cricket-based system where every node can be configured to simultaneously function as both a beacon and a listener.

## 1.2 Quick start

This section has instructions that will allow you to get started using Crickets. Use these instructions to test your Cricket hardware unit and to configure some of its parameters. You will need a serial communication terminal program such as `HyperTerminal` or `minicom`.

If you get a Cricket kit from Crossbow, the embedded software will already be installed on it. If not, then you need to install the embedded software first; see Section 1.3. The instructions for “quick start” in this section assume that the embedded software is already loaded on the Crickets.

**NOTE: The embedded software needs to be based on TinyOS 1.1.6. If you have an older version of TinyOS, you need to upgrade. Also, we have not tested Cricket with later versions of TinyOS; it might work, however. If you obtained a Cricket kit, verify that the CD that came with it has TinyOS 1.1.6 rather than an older version.**

Attach one of the Crickets to your computer's serial port. If your computer doesn't have a serial port, you can use a USB-to-serial converter.

### 1.2.1 Set up a communication terminal

**In Linux:**

- Open a terminal and run `minicom -o -s` as root or as a user with the right permissions to access the device (*e.g.*, `/dev/ttyS0`) to which the cricket is connected. By starting `minicom` with the `-o` and `-s` option you will see the configuration menu directly.

- Select **Serial port setup** and set the **Serial Device** to the port to which the Cricket is connected.
- Select **Bps/Par/Bits** to be **115200 8N1**. Hardware and software flow control can both be disabled.
- Save these settings.
- To see what you are writing on the screen, turn the echo function “on”.
- Now you are ready to send commands to the Cricket; see Section 1.2.2 to continue.

### **In Windows:**

1. Start HyperTerminal: In Windows XP, you can do this as follows:  
From **Start**→**All Programs**→**Accessories**→**Communications** select `HyperTerminal`.
2. HyperTerminal will prompt you for a **New Connection** name; type in any name you want (such as “Cricket”) and press **OK**.
3. In the next dialog box, select the **Connect using** option and set it to the communication port to which the Cricket is attached. For a direct serial connection, this port is likely to be COM1, COM2, or COM3; for a USB-to-serial connection, this port is probably COM4. Press **OK** after choosing the right option.
4. You now have to configure the serial communication protocol parameters. In the **Port Settings** dialog box, set **Bits per second** to **115200**, **Data bits** to **8**, the **Parity** to **None**, the **Stop bits** to **1**, the **Flow control** to **Xon/Xoff**. Press **OK**.
5. You can now send commands to the Cricket unit; see Section 1.2.2 to continue.

## **1.2.2 Communicate with the Cricket unit**

Once connected to the Cricket unit over a serial communication link using a terminal program like `HyperTerminal` or `minicom`, you can run the following test to see if the unit works.

1. Power the Cricket unit on. This switch is different from the “TEST” switch, which you don’t have to touch. (Both switches are marked on the board.)
2. Send the following command to the Cricket unit using the terminal program: **G CF <return>**.
3. A multi-line output should appear giving you information about the Cricket unit’s configuration.

### 1.2.3 Configure a Cricket unit to be a beacon

When you first get a Cricket unit, it will be configured as a listener. To configure it as a beacon:

1. Type **P MD 1** <return> in the terminal program.
2. Type **P SP test1** <return> to set the beacon's space ID to "test1" (or to any other string that you want). The maximum length of the space ID is 8 bytes.
3. Type **G CF** <return> to check the current configuration; you should find the new space ID and the beacon setting.
4. Save the new configuration to the flash by sending **P SV** <return>. If you don't save the configuration, the new settings will only remain until the Cricket unit is powered off.

### 1.2.4 Test distance measurements

1. Connect and turn on a Cricket configured as a listener to your host.
2. Turn on a Cricket configured as a beacon.
3. You should now see distance measurements appear in your terminal every time the listener hears from the beacon. The output has many fields; the "DB" field gives the distance to the beacon. The default units are centimeters (but can be changed to inches); the next chapter documents these and other options in detail.

If the above steps work, you can now use your Crickets to write and run applications! If the steps don't work, you may find the troubleshooting hints in the next chapter useful.

## 1.3 Installing the Cricket embedded software image

### 1.3.1 With TinyOS

This section describes how to program and/or modify the embedded code on the Cricket units. This chapter assumes some familiarity with the TinyOS platform; details on that platform are at <http://www.tinyos.net/>

You need a *programming board* (also called a "programmer") to program Cricket units with the TinyOS-based software image. Cricket can be programmed using the same programmers as Mica2 motes. We use the MIB510CA model.

The package providing the TinyOS-based embedded source code for Cricket contains only changed/added files. The files provided are in two separate directories in the top level `tos` directory:

<code>/apps/Cricket/</code>	Cricket application
<code>/apps/Makerules</code>	Makerules with the addition of the cricket platform.
<code>/tos/platform/cricket/</code>	Cricket platform directory

The current version of the Cricket embedded software works with TinyOS version 1.1.6. To install the development environment, install TinyOS 1.1.6 and copy the files included in the Cricket package to the corresponding TinyOS path.

The embedded source code for the Cricket units has two parts:

1. The Cricket platform.
2. The Cricket application.

The Cricket platform contains all the software differences that handle the differences between the Cricket hardware and the Mica2 hardware. The Cricket application is the software that incorporates the beacon and listener algorithms.

After installing TinyOS and copying in the Cricket package, install the Cricket application by typing:

```
# MIB510=<SERIAL_PORT> (e.g., /dev/ttyS0)
# make cricket install
```

The above lines assume you have a MIB510CA programmer connected to a serial port. Check the TinyOS documentation for details on other programmers.

**Note:** Currently, by default, TinyOS erases the Cricket flash each time it is reprogrammed. As a result, Cricket configuration parameters that were saved before the reprogramming action will be lost and reset to the default values.

**Note:** This document does not currently discuss how to modify the embedded software or how to write TinyOS-based Cricket applications. A future version of the document will address these issues.

### 1.3.2 Without TinyOS

This section describe how to program the software onto the Crickets using a already compiled version of the software.

You need a *programming board* (also called a “programmer”) to program Cricket units with the TinyOS-based software image. Cricket can be programmed using the same programmers as Mica2 motes. We use the MIB510CA model.

The embedded image is available from <http://cricket.csail.mit.edu> as a firmware only package (two compressed packages are available .zip and .tar.gz for your convenience and both contain the same binaries).

The package contain the following files:

- uisp.exe - The windows upload software

- uisp - The linux upload software
- main.srec - The cricket embedded software

Under Windows you need `cygwin` to be installed on your system. `cygwin` is available from <http://www.cygwin.com/>.

To install uncompress the package in a directory and execute the following command from that directory after connection the programmer and placing the Cricket on the programmer:

```
./uisp -dprog=mib510 -dserial=/dev/ttyS0 -dpart=ATmega128 --wr_fuse_e=ff  
--erase --upload if=main.srec
```

Where the `-dserial` parameter represent your COM port (e.g `/dev/ttyS0` is COM1).

**Note:** The Cricket should be connected using a straight cable and not a null modem cable.

## 1.4 Overview of this manual

The next chapter describes how to configure Crickets using the serial command interface and provides troubleshooting and deployment tips. Chapter 3 describes `BeaconConfigDemo`, a simple Cricket application that you can use to configure beacon coordinates and track a mobile device. This application is part of the Cricket software release. Chapter 4 describes how to write Cricket applications in Java using libraries that are part of the Cricket release (these libraries provide better abstractions than using the serial API directly).



# Chapter 2

## Command Interface and Troubleshooting

This chapter describes some salient hardware features of Cricket, the command interface to configure and read various parameters, and discusses troubleshooting and deployment issues.

The Cricket beacon and listener hardware are identical; they just run different software. In fact, there is only one embedded software image, and a runtime configuration switch (described below) determines whether a given unit is a beacon or a listener. By default, each Cricket node is configured to run as a listener. The Cricket embedded software runs in the TinyOS environment [6].

At least two Crickets are needed to operate the system, at least one beacon and at least one listener. In the current version of Cricket, the listener is usually attached to a host using a serial cable.<sup>1</sup>

The host device (to which the listener is attached) must run software to process the data obtained from the listener. One way to process this data on Linux computers (including handhelds) is to use `cricketd` (Section 2.2.1), which processes information obtained over the serial interface to obtain various location properties. In particular, `cricketd` runs on Familiar Linux [3] (on iPAQ handheld computers), as well as on standard laptop/desktop versions of Linux and Windows (under Cygwin [2]). We do not currently support `cricketd` on handhelds running Windows Pocket PC.

### 2.1 Cricket Hardware Details

Figure 2.1 shows the hardware components and board layout of the Cricket v2 units.

Cricket uses time-difference-of-arrival between RF and ultrasound to obtain distance estimates. Its radio runs at a frequency of 433 Mhz, with the default transmit power level and antennas providing a range of about 30 meters indoors when there are no obstacles.<sup>2</sup> The maximum ultrasound range is 10.5 meters when the listener and the beacon are facing each other and there are no obstacles between them. Our measurements show that the distance accuracy of the Cricket hardware is

---

<sup>1</sup>We expect to have a listener with a compact flash interface in the future.

<sup>2</sup>The radio range depends on the antenna and the kinds of obstacles between the two Crickets; its profile is complicated, non-isotropic, and asymmetric.

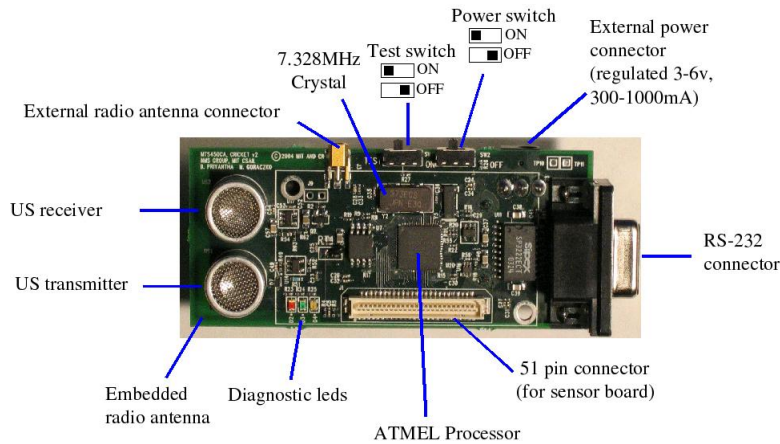


Figure 2.1: Cricket v2 hardware components and layout.

on the order of 1 centimeter at a distance of up to 3.5 meters, and 2 centimeters in the rest of the 10.5-meter range.

### 2.1.1 Diagnostic LEDs

The Cricket hardware has three LEDs, which you can use to infer a few things about the device’s state. At startup, all the LEDs light up for 500 milliseconds. At run time the LEDs light up as shown in the following table:

LED	Listener	Beacon
Green (D3)	lights up when a valid RF chirp arrives from a beacon	lights up for 150 microseconds during ultrasound transmission (may not be visible in bright light)
Red (D2)	lights up when a collision occurs (e.g., due to hidden terminal or improper beaconing)	lights up for 150 microseconds, when the beacon fails to grab a time slot to send a chirp
Yellow (D4)	lights up when an RF message is received but with no ultrasonic pulse	Not used yet

### 2.1.2 Test Switch

In addition to the power switch, each Cricket has a *test switch*. The behavior of the Cricket unit in each test switch setting is under software control.

Currently, this switch is used when the Cricket runs as a beacon. If the test switch is placed in the “ON” position before the Cricket is turned on, it will disable the onboard RS232 chip during start-up, to save energy while running. Even when the onboard RS232 chip is disabled, the Cricket serial port can be accessed using a programming board (see Section 1.3).

To enable the RS-232 chip again, restart the Cricket with the test switch in the OFF position.

### 2.1.3 Powering Crickets

Cricket comes equipped with a battery pack and an external power connector. It can be powered using two standard AA batteries or with an external power supply that provides 3-6 volts regulated at 300-1000 mA.

It is also possible to power Crickets using solar panels. Details are available at <http://cricket.csail.mit.edu/>.

## 2.2 Serial Port Command Interface

The Cricket listener provides data in ASCII format to software running on the attached host over an RS232 interface. The serial command interface parameters are:

Transmission speed	115200 bits/second
Data format	8 bits, no parity
Flow control	Xon/Xoff (“software”)
Stop bits	1

The ASCII-based serial port command interface is similar to the “AT” command interface that modems use. The command interface is accessible using standard utilities like `HyperTerminal` or `minicom`. Section 1.2 describes how to use these utilities to send commands to Cricket.

### 2.2.1 `cricketd`

`cricketd` is a daemon used to access the command interface over the network. `cricketd` binds to port 2948 and can be accessed using `telnet` (or another TCP application) by connecting to port 2948 on the machine running `cricketd` (connect to port 2948 on `localhost` if `cricketd` runs on the same machine).

```
# telnet <IP address of the machine running cricketd> 2948
```

When connected, the API can be used the same way as with `HyperTerminal` or `minicom`. `cricketd` supports multiple TCP clients and can provide location information to more than one application.

After connecting to `cricketd`, type “r” in the console (or send the ASCII character “r” over the TCP connection) to tell `cricketd` to start and stop sending the data flow to the console (or TCP connection). This command does not affect other connected clients.

### Compiling `cricketd`

In Linux or on Windows running a version `cygwin`  $\geq$  v1.5.10, run the following in `Cricket2.0/src/cricketd/`:

```
# autoconf
# ./configure
# make
```

To cross-compile for IPAQs, do the following:

1. Install toolchain 2.95 from handhelds.org (<http://handhelds.org/download/toolchain/cross-2.95-3.tar.bz2>).
2. Unpack file in the “/” directory.
3. Type:

```
# make clean; make -f Makefile.arm
```

### **Running cricketd:**

First, make sure you are running with root/administrator privileges.

`./cricketd -h` gives the listing of command line options.

`./cricketd -p <SERIAL_PORT> -s 115200` runs `cricketd`. `SERIAL_PORT` is usually `/dev/ttyS0` in Linux or cygwin when the Cricket unit is connected to the serial port, `/dev/ttyS3` when the Cricket is connected using a USB-to-serial converter, and `/dev/ttySA0` on iPAQ handhelds. Note that in version 1.5 and later of cygwin, `/dev/ttyS0` is used in place of COM1.

The above settings are not authoritative, however. Please consult your local system administrator if you need help in finding the correct serial port device.

### **Connecting, Sending, and Receiving Messages**

1. `telnet <HOST_IP_ADDR> 2948` (or open a TCP connection).
2. Type “r” to enable or disable the connection (or send “r” over your TCP socket).
3. After connecting (and powering the Cricket on), you should start receiving Cricket messages, if the host is connected to a listener. You may enter the Cricket command followed by “ENTER” to manipulate the Cricket unit.

You may find the README file included in the `src/cricketd` directory useful.

## 2.2.2 Cricket Command Interface

All commands have the same general format, using one of two directives. GET (“G”) returns the values corresponding to the parameters specified in the command. PUT (“P”) sets the value of the specified parameter to the specified command argument.

The general command format is:

`<directive> <command> <parameters>`

`<directive>` The character “G” or “P” for “get” or “put”.

`<command>` One of the commands from the “command” column in the table on the next page.

`<parameters>` The argument(s) to the command.

In response to any command, the Cricket listener or beacon echoes the specified command followed by the result:

`<command><result>`

## 2.2.3 Error Codes

The Cricket outputs the following set of error codes when a command sent using the serial port contains an error:

Error description	Code
Invalid command	Err 1
Missing parameters	Err 2
Value out of range	Err 3
Not defined	Err 4

## 2.2.4 Default values

Variable	Default values
Run mode (MD)	Listener
Space ID (ID)	NULL
Ultrasonic timeout (UL)	45000 $\mu$ s
Coordinates (PC)	(0,0,0)
Sleep range (SL)	668 and 1332 ms (average 1000 ms)
Offset (OF)	550 clock ticks
Temperature (TP)	enabled
Units (UN)	Metric and Celsius
Output format	3
Custom format	VR,ID,SP,DB,DR,TM,TS

The rest of this section describes the commands summarized in the table on the next page.

Parameters	Description	Access Mode				Command ASCII	Get Output	Put Input	Parameters Type ASCII
		Beacon Read	Listener Write	Write	Read				
Cricket run mode	Run as beacon or listener	*	*	*	*	MD	<mode>	<mode>	String[1]
Configuration	Current cricket configuration	*	*	*	*	CF	<multiline output>	N/A	N/A
Node ID	Unique node id	*	*	*	*	ID	<id>	N/A	String[23]
Space ID	User specified space id	*	*	*	*	SP	<name>	<name>	String[8]
Version	Software version	*	*	*	*	VR	N/A	N/A	N/A
Distance	Distance between the listener and the beacon	*	*	*	*	DB	<distance>	N/A	string[5]
Duration	Ultrasound time of flight	*	*	*	*	DR	<duration>	N/A	string[5]
Time of flight	Time of flight of the ultrasound (uncorrected)	*	*	*	*	TM	<timer>	N/A	string[5]
Coordinates	Coordinates of the node (x y z)	*	*	*	*	PC	<(x,y,z)>	<(x,y,z)>	3*string[6]
Sleep range	Sleep range between beacon chirp	*	*	*	*	SL	<low high units>	<low high units>	2*string[5],string[2]
Ultrasound lifetime	The ultrasound life before the wave attenuates	*	*	*	*	UL	<lifetime units>	<lifetime units>	string[5],string[2]
Offset	Software delay offset	*	*	*	*	OF	<offset>	<offset>	string[5],string[5]
System time	Report uptime in ms	*	*	*	*	TS	<timestamp>	N/A	string[5]
Temperature	Temperature on the node	*	*	*	*	TP	<temp units>	<value>	string[5],string[2]
Test switch	Status of the test switch	*	*	*	*	TB	<status>	N/A	string[2]
Units	Units used for display on valves	*	*	*	*	UN	<string>	<dist_units temp_units>	string[],2*string[1]
Listing	List all beacon heard	*	*	*	*	LS	<multiline output>	N/A	N/A
Save parameters	Save current parameters to flash	*	*	*	*	SV	N/A	N/A	N/A
Load parameters	Load parameters saved in the flash	*	*	*	*	LD	N/A	N/A	N/A
Output format	Output format	*	*	*	*	OU	<format>	<format>	string[5],string[5]
Custom format	Custom report output	*	*	*	*	CO	<custom expression>	<custom expression>	N/A

## 2.2.5 Run mode command (MD)

To place a Cricket device in “beacon” mode, set this parameter to 1. To place a Cricket device in “listener” mode, set this parameter to 2.

Example:

```
P MD 1<return>
```

Result:

```
MD BEACON
```

## 2.2.6 Configuration status command (CF)

The configuration command triggers a report that gives the values of all the Cricket parameters. An example of the output of this command, with explanations on the side, is shown below.

Example (get configuration):

```
G CF<return>
```

Result:

```
Cricket configuration:
Software version: 2.0 // Cricket software version
Mode: Listener // Running mode (Listener/Beacon)
Unique id: 1:c8:6a:b3:a:0:0:dc // Unique Cricket ID
Space id: MIT-6 // User-defined space ID
Uptime: 16:32:14 // Uptime of Cricket from last power cycle (hh:mm:ss)
Ultrasound attenuation time(us): 45000 // Time Cricket should wait for the ultrasound to attenuate
Timer Offset(us): 550 // Offset to compensate for software processing time
Minimum beacon interval(ms): 668 // Minimum wait time between beacon messages
Maximum beacon interval(ms): 1332 // Maximum wait time between beacon messages
Average beacon interval(ms): 1000 // Average interval between beacon messages
Compensation value(us): 48 // Time in us that one bit takes to travel over the Cricket radio
Distance Units: Metric // Units used to display the distance measurement (DB)
Local temperature value (Celsius): disabled // Temperature from the onboard sensor
Speed of sound value(m/s): not used // Speed of sound calculated based on the temperature
Test switch status: On // Position of the test switch on the Cricket
Event output format: 3 // Event reporting format
Output variable(s): VR ID DB DR SP TM TS // Variables output (configurable using the CO command)
```

```
Massachusetts Institute of Technology
http://nms.csail.mit.edu/cricket
```

## 2.2.7 Get serial ID (ID)

The Cricket ID is a 64-bit number that is permanent and cannot be changed; it is taken from a serial number DS204 chip and is analogous to the MAC address on a network card. This number is returned in the form of 8 hexadecimal numbers.

Example:

G ID<return>

Result:

ID 81:23:a1:34:01:43:12:e3

### **2.2.8 Get/Put space ID (SP)**

The space ID is an 8-byte string that can provide more information about a beacon. The space ID has to be set by the user at least once to have a value different from NULL.

Example (set space ID to “MIT-6”):

P SP MIT-6<return>

Result:

SP MIT-6

### **2.2.9 Get software version (VR)**

Return the software version of the Cricket being queried. The current version is 2.0.

### **2.2.10 Distance to beacon (DB)**

The listener reports the distance to a beacon each time the listener hears an RF message and a concurrent ultrasonic signal from that beacon. The reported distance is in the units set using the UN command (Section 2.2.20).

### **2.2.11 Duration (DR)**

The duration is reported by the listener under the same conditions as in “DB”, above. The duration represents the time-of-flight of the ultrasonic pulse, compensating for the various time offsets for accuracy. This value can be used to calculate a distance with with more precision than the DB distance, because the latter uses a (temperature-compensated) value for the speed of sound, which may introduce some error. For example, to determine which of multiple beacons is closest to the listener, compare the DR values. The units of the reported duration is microseconds.

### **2.2.12 The uncorrected time of flight (TM)**

This value is the same as the duration (DR) but without any compensation. It is also reported by the listener each time a distance measurement arrives.

### 2.2.13 Get/Put coordinates command (PC)

The  $x$ ,  $y$ , and  $z$  coordinates can be preset for a beacon. Each value ranges between 0 and 65536 and are in the units specified by the “P UN” command.

Example (set the coordinates to (32,2,3)):

```
P PC 34 2 3<return>
```

Result:

```
PC (32,2,3)
```

### 2.2.14 Get/Put the minimum and maximum beacon interval time (SL)

The minimum and maximum sleep times between beacon chirps can be set using the SL command. The valid range for the minimum value is 200 to 65536 ms and the maximum value needs to be higher than the minimum by at least twice the attenuation time of the ultrasound (45 ms), but cannot exceed 65536 ms. The average beacon chirp interval depends on the minimum and maximum values, because the sleep time is randomly chosen to be in this range.

Example (set the interval to minimum of 500 ms and a maximum of 1500 ms):

```
P SL 500 1500<return>
```

Result:

```
SL 500 1500
```

The average sleep time in this example is  $(500 + 1500) / 2 = 1000$  ms.

### 2.2.15 Get/Put the ultrasound maximum time-of-flight (UL)

This parameter is used by the beacons to wait for the ultrasound to attenuate. The beacon waits for at least this much time (in microseconds) after hearing another beacon’s chirp before attempting a chirp. The default value is 45000 (45 ms). Valid values are from 0 to 65536 (65 ms).

Example (set ultrasound lifetime to 40000 microseconds):

```
P UL 40000<return>
```

Result:

```
UL 40000
```

### 2.2.16 Get/Put the software offset (OF)

The software offset can be changed from the preset value (550  $\mu$ s) to any value between 0 and 65536. This offset compensates for processing time during the reception of beacon messages. It depends on the processor speed.

Example (set offset to 500  $\mu$ s):

```
P OF 500<return>
```

Result:

```
OF 500
```

**Note:** This command only useful for user change the Cricket embedded software installed on the Cricket node. You should not have to change this parameter, and if you do, it has to be changed with extreme care.

### 2.2.17 Get system time (TS)

Report the elapsed time (32 bits long) since the last power-up of the Cricket unit. This time is reported in milliseconds.

Example (get the system time):

```
G TS<return>
```

Result:

```
TS 32983
```

### 2.2.18 Get/Put temperature sensors (TP)

The temperature command reports the temperature of the Cricket unit. It is also used to enable the temperature sensor in the first place. When the temperature sensor is enabled, the embedded Cricket software calculates distances using the speed of sound at the ambient temperature. The temperature is returned in the units specified by the UN command (Section 2.2.20). By default the temperature is specified in degrees Celsius.

By sending “P TP 1/0”, you can enable(1)/disable(0) the temperature sensor.

Example (enable the temperature sensors):

```
P TP 1<return>
```

Result:

```
TP 1
```

### 2.2.19 Test switch status (TB)

The TB value represents the test switch position on the Cricket beacon that caused this report. When queried using the G directive (“G TB”), the TB value that’s returned represents the status of the test switch on the Cricket unit connected to the serial port.

Example:

```
G TB<return>
```

Result:

```
TB On
```

### 2.2.20 Get/Put display units (UN)

The display units parameter is used to set or get the units used to display the information reported by the Cricket over the command interface. The command take two parameters: the first one is the distance units and the second the temperature units. The distance units can be “1”, meaning Metric (centimeters), or “2”, meaning Imperial (inches). The temperature can be “1”, in degrees Celsius or “2”, degrees Fahrenheit. When queried, the command returns the current settings.

Example (put Imperial and Celsius units):

```
P UN 2 1<return>
```

Result:

```
UN Imperial and Celsius
```

**Note:** The default setting of the units is “Metric” and “Celsius”.

### 2.2.21 Get the beacon listing (LS)

The listing command lists all beacons heard (up to a maximum of 15 beacons) in the past 15 seconds and the last 5 distances measured for each of these beacons. Each output line reports the unique ID of the beacon heard and the last 5 distance measurement received for that beacon.

```
LS 0: 1:f2:66:b3:a:0:0:df 165 165 169 175 180  
LS 1: 2:f2:32:67:a:0:0:45 40 40 40 40 40  
LS 2: a:62:0:b3:a:0:0:21 17 16 15 12 10
```

**Note:** Currently only the ID and the distances are reported. The distances reported are in the units set by the UN command (Section 2.2.20).

### 2.2.22 Save settings to flash (SV)

The SV command is used to save the current settings into the flash. The red light will light up during the “save” operation. The saved parameters will be loaded each time the Cricket powers up or when the “P LD” command is sent to the Cricket unit.

**Note:** The flash is erased each time the Cricket is reprogrammed with new software. Non-default configuration parameters that were previously saved will *not* be retained across reprogramming actions.

### 2.2.23 Load settings from flash (LD)

The LD command reloads the parameters from the flash.

**Note:** The LD command is equivalent to restarting the unit, and the same rules apply to the test switch in terms of enabling or disabling the serial port (see Section 2.1.2).

### 2.2.24 Get/Put Output format (OU)

Cricket v2 has a few pre-programmed output formats for backward-compatibility with Cricket v1 (this feature is probably not useful for most users). By using the OU command the output format can be set to be compatible with previous Cricket versions, as follows:

Parameters	Description
0	No output at all
1	Cricket version 1 with decimal distances
2	Cricket version 1 with hexadecimal distances
3	Cricket version 2 output format (default)

The output format for each parameter is as follows:

Output 1:

```
$Cricket2,ver=3.0,id=45,dist=17,duration=1435,time=1195
```

Output 2:

```
$Cricket2,ver=3.0,space=MIT-0,id=45,dist=11,duration=1435
```

Output 3:

```
VR=2.0,ID=1:5e:3c:3c:a:0:0:ba,SP=MIT-0,DB=14,DR=423,TM=1045,TS=207040
```

**Note:** The version number (VR) in output 1 and output 2 do not reflect the same version scheme as in the current version (output 3) of the software. They should only be used by legacy Cricket applications that were created for Cricket v1.

## 2.2.25 Customize output format (CO)

The custom output command is used to change format of the information reported from the listener via the command interface. It allows the user to specify which parameters should be reported. The parameters are the names of the commands from the tables.

For example, by sending “P CO DB VR” over the command interface, the reports from the listener will look like:

```
VR=2.0,DB=98
```

**Note:** The user cannot specify the order in which the parameter values should be reported. We provide two special settings:

- \* corresponds to every available parameter symbol.
- – corresponds to the default reporting line.

For example, by sending “P \*” the report of the distance will look like

```
VR=2.0,ID=01:dd:be:be:09:00:00:95,SP=MIT-2,DB=224,DR=6479,TM=6789,TS=455424
```

**Note:** The custom output command works only when the output format is set to 3. The example also assumes that all the options are enabled on the beacon and that the listener can measure distances from the beacon.

## 2.3 Troubleshooting and Deployment Hints

This section tells you how to resolve some common problems.

### 2.3.1 Problem: The listener or the beacon does not respond to any command

- Verify that TinyOS version 1.1.6 has been installed and the Cricket embedded software loaded. If not, do so using a programming board.
- Verify that the serial parameters are set to 115200 baud 8N1.
- Check that the Cricket is ON.
- Check that the batteries are not dead.
- Make sure no other instance of a program like `minicom` or `cricketd` is already running, which has locked access to the serial port.

### **2.3.2 Problem: The listener does not report any events**

- Check if the green light on the listener lights up from time to time; if not, check if the beacon is ON and that its green light blinks.
- Check if the output format (using the OU command) is set to something different than 0; type “P OU 3” to set it to report information. If this step works, typing “P SV” will save the configuration.

### **2.3.3 Problem: The listener returns erroneous distances**

If the reported distances are wrong by more than a few centimeters, then something is amiss.

- If the results are wrong for every beacon, the most likely problem is that the batteries on the listener are weak.
- If the wrong distance comes from a beacon that was previously correct, then that beacon’s batteries are probably weak.
- Check the setting of the units in which the distance reports are being made. You can use “G UN” or “P UN ... ..” to set these units. See Section 2.2.20.
- If the listener reports erroneous distance at certain locations but not others, then you may have sources of interference on the path from beacon to listener. Some objects block ultrasonic signals and some others reflect them; some even generate ultrasonic signals at the same frequency as Cricket. All these objects cause the Crickets to report wrong distances.

Obstructions close to the beacon’s ultrasonic transmitter or listener’s ultrasonic receiver cause the biggest problems. These obstructions could be walls or doors (through which ultrasound does not pass), or could be people. Large metallic plates or cabinets on the path from beacon to listener can disrupt distance estimation by affecting both ultrasound and RF propagation. We have also found that some fluorescent lamps generate 40 kHz ultrasonic waves that can interfere with the Crickets.

Don’t place beacons too close to large objects; if a large object is within 10 or 15 centimeters of a beacon, that beacon’s transmissions may be blocked. Dealing with interfering fluorescent lamps could be harder; however, we have found that the interference usually comes from lamps that are close to dying, so you might think of this interference as an early-warning system to replace the lamp!

### **2.3.4 Problem: The beacon blinks but does not respond to serial commands**

If the Cricket was turned on with the test switch in the “ON” position, then the serial port is disabled to save energy while running. You have two options if you want to communicate with the beacon over the command interface:

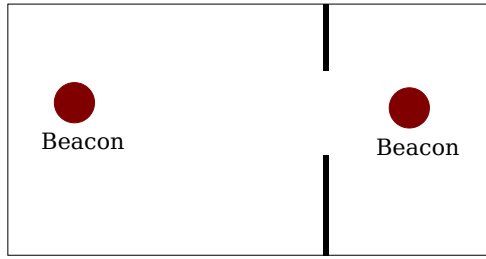


Figure 2.3: Incorrect placement of beacons to for boundary detection between two spaces.

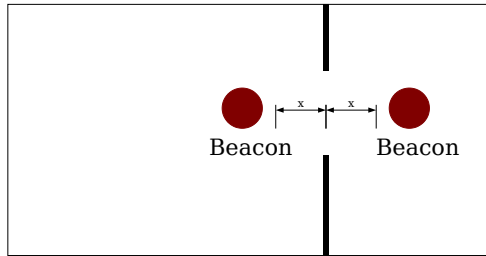


Figure 2.4: Correct beacon placement.

- Restart the Cricket with the test switch in the “OFF” position. Note that this configuration consumes more power.
- Connect the Cricket to a MIB510CA programmer. The serial port is then accessible without using the on-board RS232 controller.

### 2.3.5 Problem: The listener associates itself with the “wrong” space identifier

When using Cricket beacons to delimit spaces that are not separated by a wall, both RF and ultrasound from the beacons in the two different spaces may traverse the other space. Because the listener associates itself with the space advertised by the nearest beacon, you need to place beacons with some care to achieve proper spatial demarcation. Figure 2.3 illustrates a bad configuration of beacons, where a listener on the left side of the partition could be closer to the beacon on the right, causing it to associate itself with the wrong space.

Place the beacons corresponding to the different spaces at equal distances from the boundary between the spaces, as shown in Figure 2.4.



# Chapter 3

## Sample Application

We have developed a sample Cricket application (which we will also call the “demo” program), `BeaconConfigDemo`, that lets a user make simple drawings using Cricket. This application demonstrates the following features of the Cricket location system:

- Assisted configuration of an adhoc beacon coordinate system
- Accuracy of distance measurements
- Accuracy and latency of real-time tracking of the listener’s position within the adhoc beacon coordinate system

This chapter explains how to setup, launch and use the `BeaconConfigDemo` application.

### 3.1 Setup

You need Java installed on your system. Please download the latest Java runtime environment (JRE) or software development kit (SDK) (version  $\geq$  1.4) from <http://www.javasoft.com/>

On Windows systems, you need `cygwin` to run `cricketd` (see Chapter 2) and to run the scripts provided with the demo. Please download the latest version of `cygwin` from <http://www.cygwin.com/>

### 3.2 Source Code

The source code for `BeaconConfigDemo` is located under the Cricket distribution directory `src/app/beaconconfig/`

A set of pre-compiled binaries is already included in the Cricket distribution. If you wish to compile the source code, simply run `make` in the source directory. You’ll need the Java SDK on your machine to compile the code.

### 3.3 Beacon Placement

The BeaconConfigDemo application requires at least three beacons mounted on a flat surface (either floor or ceiling or along the sides of walls). The listener must be able to listen to *all* the beacons at once when placed under each beacon. The typical beacon operating range of a Cricket beacon is  $\approx 10$  meters (with the antenna and power levels that are set by default).

Don't place a large number of beacons ( $> 8$ ) within a given area as that will cause excessive contention and could increase the latency of position tracking in this demo.

Finally, if four or more beacons are used, the beacons cannot be placed such that any four of them are on the same circle (in particular, you cannot place them at the four corners of a rectangle or square).<sup>1</sup>

### 3.4 Setting Parameters

Next, edit the parameters in NMSDemo/2004/BeaconConfigDemo/launchall.sh You will probably need to change the CRICKET\_IPADDR to 127.0.0.1 and may need to change NUMBEACONS.

```
MODE='`normal`' or '`simulation`'  
NUMBEACONS=4  
CRICKETD_IPADDR=10.0.0.100  
CRICKETD_PRINT_ENABLE='`-r`'  
REMOTE_ENABLE='`-r`'  
ENOUGHSAMPLES=2  
SEPDIST_CM=10  
STDDEV_TOLERANCE_CM=10  
FILTER_WINDOW_MS=4000  
WIN_SIZE=500  
GRID_SIZE_CM=61  
SAMPLEMONHISTORY_MS=5000  
SAMPLEMONMAXY_CM=400  
STREAMERSAMPLINGPERIOD_MS=50  
STREAMERDELAY_MS=3000  
JAVAHOTSPOT_ENABLE='`-server`'
```

MODE sets normal or simulation mode (see Section 3.5.3)

NUMBEACONS sets the number of active beacons being used.

CRICKETD\_IPADDR sets the IP address of the listener's host device that is running cricketd (see Section 2.2.1). Use 127.0.0.1 if cricketd is running on localhost.

---

<sup>1</sup>When the beacons are all on the same circles, the resulting system of equations is degenerate and the listener will show up at a position on the imaginary plane!

`CRICKETD_PRINT_ENABLE` enables CricketDaemon to print the raw information supplied by `cricketd` and lets you see the activity in the data path between `cricketd` and `CricketDaemon`. This is useful for trouble shooting purposes. When the beacon reception rate is high, the screen output may cause a very high CPU load. This option may be disabled by prepending the line with a '#' symbol.

`REMOTE_ENABLE` enables connection with `BeaconConfigRemote` (see Section 3.5.3). If this option is enabled AND `BeaconConfigRemote` is not running, `BeaconConfig` may "hang" due to network timeouts. This option may be disabled by prepending the line with a '#' symbol.

`ENOUGHSAMPLES` is used to filter corrupted data. Beacons with fewer than `ENOUGHSAMPLES` within a history window (defined by `FILTER_WINDOW_MS`) are ignored. This is useful to filter invalid beacons as a result of noise generated by poor serial port connections or RF errors that corrupt the beacon information. Usually, a value of 2-3 is enough for effective filtering. This filter has no effect on the accuracy of the distance measurement between a beacon and a listener.

`SEPDIST_CM` defines the *maximum separation distance* (in centimeters) between the listener and a beacon during the beacon coordinate configuration phase (see Section 3.5.3). That is, the listener must be held within `SEPDIST_CM` cm directly below a beacon during calibration. When `SEPDIST_CM` is large relative to the distances between beacons (1.5x or greater), the user should move swiftly from one beacon to another during the calibration process. Otherwise, `BeaconConfig` might start a measurement while the listener is not directly underneath a beacon and cause errors in the beacon coordinate computation.

`STDDEV_TOLERANCE_CM` defines the threshold in which distance readings are considered stable the beacon coordinate configuration phase (see Section 3.5.3). Distance estimation with respect to a beacon is considered stable if the readings collected within the past `FILTER_WINDOW_MS` milliseconds has a standard deviation below `STDDEV_TOLERANCE_CM`. During tracking phase, sample distances that are greater than  $3 * \text{STDDEV\_TOLERANCE\_CM}$  will also be filtered.

`FILTER_WINDOW_MS` specifies the window size (in milliseconds) of the sliding-window filter used to reject erroneous distance readings from the Cricket listener. Higher value improves accuracy but increases the position tracking latency. For example, a value 10000 will cause the program to keep a history of all the distance samples collected over the past 10 seconds. For Cricket v2, the recommended value is 4000. The mode value of the distances collected in the history window is used to estimate the true distance between a beacon and the listener.

`WIN_SIZE_PIX` sets the size of the `BeaconConfig` GUI (square) window in pixels.

`GRID_SIZE_CM` sets the grid size in centimeters. The GUI will automatically adjust the drawing scale so the size of the grid as appear on the screen may vary.

SAMPLEMONHISTORY\_MS sets the size of the sample monitor (in the BeaconFinderApp window) history window in milliseconds.

SAMPLEMONMAXY\_CM sets the max value for the y-axis (cm) in the sample monitor (in the BeaconFinderApp window).

STREAMERSAMPLINGPERIOD\_MS sets the density of the streamer. The streamer draws the trail of the Cricket listener during tracking mode. A smaller period will draw more dots per time unit for the trail.

STREAMERDELAY\_MS defines how long the streamer trail dots stay on the screen before disappearing.

JAVAHOTSPOT\_ENABLE enables Sun's Java HotSpot Server mode. When enabled, the jvm will perform dynamic just-in-time compiling to increase performance. However, the start up time will be quite slow. In general, the applications will run sluggishly in the first minute or so (depending on the computer speed) until the compilation is complete. After the first minute, the CPU load should drop significantly. This option may be disabled by prepending the line with a '#' symbol.

## 3.5 Launching and Running BeaconConfigDemo

The BeaconConfigDemo application has two operating phases. The first phase configures an ad hoc beacon coordinate system for the set of active beacons, and the second phase tracks the listener's position in real time. BeaconConfigDemo then uses the position tracking to let user draw polylines, rectangles, and circles in its window.

### 3.5.1 Launching cricketd

To run BeaconConfigDemo, cricketd must be running on the listener's host device (whose IP address is CRICKETD\_IPADDR). To compile and run cricketd, see Section 2.2.1.

### 3.5.2 Launching BeaconConfigDemo

At the shell prompt, type the following commands:

```
# cd NMSDemo/2004/BeaconConfigDemo
# ./launchall.sh
```

Two programs will be launched. The first is called the Beacon Finder, which displays the distance measurement statistics for each beacon. The second application is called Beacon Configuration, which executes the two phases of configuring the adhoc beacon coordinate system and position tracking.

Use the `Beacon Finder` to verify that the listener is within range of all beacons. Do this by holding the listener underneath each beacon and verify that 1) it has a row entry for `NUMBEACONS` different beacons and 2) none of the entries have excessively long last-update times ( $> 5000\text{ms}$ ).

Otherwise, some beacons are out of range of the listener and cause the beacon coordinate configuration phase to fail. Please adjust the beacon placement until all the beacons are within range.

Below the numerical statistics are two sub-windows. The first sub-window is a graphical representation of the standard deviation of the measured distances of each beacon within a history window defined by `FILTER_WINDOW_MS`. The second sub-window is a time “sweep” stick plot of the distance measurements (stick height) in the history window. The color of each stick is encoded to correspond to the beacon from which the distance sample is measured.

### 3.5.3 Running BeaconConfigDemo

#### Phase 1: Configuring the Beacon Coordinate System

This phase discovers the coordinates of each active beacon being used in the demo. The user needs to hold the listener still within `SEPDIST_CM` (default is 5 meters, see Section 3.4) directly below 3 different *reference beacons*, perpendicular to the plane on which the beacons are deployed. Only 3 reference beacons need to be measured regardless to the number of active beacons being used. The action of steadily holding a listener underneath a reference beacon lets the system measure the distances to all other beacons with respect to the reference beacon.

In general, it does not matter which three among the set of installed beacons are selected for reference. However, coordinate accuracy should improve if the position of the three selected beacons circumscribe the tracking area.

The coordinate system axes are defined by the order of listener placement underneath each of the three reference beacons. The first reference beacon that is being “calibrated” by the listener defines the origin of the coordinate system in the lower left corner. The distance measurements can be noisy and the user must hold the listener still (or tilting the listener slightly) until after the standard deviations of the distance measurements for all of the beacons fall below `STDDEV_TOLERANCE_CM` (default 10 cm). When this happens, the `Beacon Configuration` window will then show a dot (and sound a beep), representing the origin defined by this beacon. Note that a dot will *not* show the distance measurements if any one of the beacons are faulty (i.e. either the distances have high variations or the beacon is out of range or if there is not enough samples to make a distance estimate).

Repeat the same process for the second reference beacon. When done, the `Beacon Configuration` window will display a second dot (and sound two beeps), representing the position of the second reference in the coordinate system. Thus, the line extended between the first and second beacon becomes the horizontal axis of beacon coordinate system.

Repeat the process again for the third reference beacon. When done, the coordinate system configuration is complete and the `Beacon Configuration` window will display all the dots representing the positions of all the beacons that have been configured. The third reference beacon does *not* define the y-axis. Rather, it defines the direction of the positive y-axis. This should

become intuitive after the user gains experience with performing the steps described here.

We summarize the steps for configuring the beacon coordinate system below:

1. Lay out 3 or more beacons on a flat surface (ceiling or floor, etc.). If 4 beacons are being used, make sure they don't form a square or rectangle.
2. Select 3 beacons as references.
3. Steadily hold the listener directly above (below) a reference beacon for a few seconds until all the error bars in the `Beacon Finder` window falls under the indicated threshold. When a new dot appears in the `Beacon Configuration` window, move to next step.
4. Repeat step 3 two more times to configure the other two reference beacons. The order in which you calibrate the reference beacons defines the directions of the coordinate system axes.
5. Once all the refereces have been calibrated, the `Beacon Configuration` program enters tracking mode, which shows the listener's position with respect to the beacons.

## Phase 2: Tracking/Drawing Mode

After the beacon coordinates have been configured, the `Beacon Configuration` program will automatically enter the tracking mode, which displays the listener's position (represented by a red square) in real time. The user may now use the listener device as a virtual mouse in free space to draw various objects such as polylines, circles and rectangles. The drawing interface is based on a "pen up" and "pen down" model, much behaves much like an old-fashioned plotter. For example, click on the check box for `Line` once will anchor the first *end* point in the line. Click on the check box again (after moving the listener to a different position) will anchor an *intermediate* point on the polyline. The second *end* point of the polyline is anchored when the user clicks on "PenUp".

To erase a drawn object, move the mouse to high-light the object to be erased and click on the right mouse button. Clicking on the "EraseAll" button would wipe all the drawn objects from the screen.

This drawing program can be used to capture the general shape and position of various objects (e.g. furniture) in the environment by outlining the objects with the listener. If the listener can be set steadily for a length of time during the outlining process, one can produce an amazingly accurate capture of the environment!

Due to the sliding window distance filtering algorithm, there is a lag in tracking the listener's position. Depending on the beacon range, the level of contention, the noise in the distance readings, and various other factors, this lag may increase up to the size of the sliding window, which has default value of 4 seconds.

The "Aggressive" option bypasses the sliding window and enables `CricketDaemon` to compute position estimates based on the very last distance sample measured from each beacon. The tracking latency is reduced at the cost of increased errors from the occasional incorrect distance measurement

When enabled, the “Streamer” option draws the trail of the Cricket listener during tracking mode.

Finally, there is an option to display the distance *annunus*, which graphically represents the distance measured by the listener with respect to each beacon. The thickness of the annunus represents one half standard deviation of the measured distance. Thus, the intersection of the annuni graphically depicts the approximate position of the listener. The annunus display offer some intuition about how the position estimation works in real time.

## Running the BeaconConfigDemo Remote User Interface

There is a program that allows the user to *remotely* control the user interface of the Beacon Configuration window. This remote control is especially useful when the listener’s host device is a handheld and its distance readings are being forwarded to a desktop or laptop computer that is running the BeaconConfigDemo (see Section 2.2.1 for more explanation about this usage model).

The remote control application should be launched on the device that will be running the remote. To launch the remote, run

```
cd NMSDemo/2004/BeaconConfigDemo
./launchremote.sh \texttt{IPADDR}
```

where IPADDR is the IP address of the computer running the Beacon Configuration program.

## Simulation Mode

One can run the entire BeaconConfigDemo in *simulation mode*. In this mode, everything works the same way as described above except that the user controls the listener’s position in a virtual environment. To launch BeaconConfigDemo in simulation mode, run

```
cd NMSDemo/2004/BeaconConfigDemo
set the MODE line in launchall.sh to ``simulation``
./launchall.sh
```

In addition to the program windows described in earlier sections, there is now a new window that displays the virtual environment. The environment contains 5 beacons and a listener (represented by a short black segment). When a beacon transmits a signal, a red circle centered at the transmitting beacon will appear.

The user can use the following set of keys to navigate the listener in the virtual environment.

- i north
- k east
- m south

j west

u up (altitude)

d down (altitude)

For example, to calibrate the beacon coordinates, the user should

1. move the listener first to Beacon A and wait until a dot appears in the Beacon Configuration window
2. move the listener to Beacon C, wait until another dot appears,
3. move the listener to Beacon D, wait until the third dot appears

Then the user can move the listener in the virtual environment and watch how its position is being tracked in the Beacon Configuration window. The `beacon.conf` file sets the location of the beacons in the virtual environment window.

### 3.5.4 Troubleshooting

The following commands are useful for low-level debugging.

1. `telnet localhost 5001` and type “register”, ENTER (taps data from CricketDaemon—you should see readable numbers and characters)
2. `telnet localhost 2948` and type 'r' and ENTER (taps data from `cricketd`—you should see readable numbers and characters)
3. hyperterminal to COM1 (115200 baud 8N1) (taps data directly from listener)

Some common causes of errors are:

1. Low battery
2. Loose cable
3. Network connectivity or incorrect `CRICKETD_IPADDR` setting in `launchall.sh`
4. Running an incompatible version of cygwin (`> 1.5.10`). (Currently, there are some issues with cygwin. To detect this error, try telnetting from a Linux machine to `cricketd` running in the cygwin/Windows. If the Linux terminal produces garbage output, then the `BeaconConfigDemo` will crash, even if it runs on the same machine running `cricketd`. We are working to resolve this issue.)

5. Some lighting fixtures (especially fluorescent tubes that are about to die) may produce noise in the ultrasonic range that interferes with Cricket and cause distance measurements to fluctuate rapidly. Turn off these fixtures to see if the distance measurements becomes more stable.
6. Although Cricket does not require line-of-sight for distance ranging, objects in the environment may deflect signals and cause a systematic error in distance ranging.



# Chapter 4

## Developing Cricket Applications in Java

The software package includes a library to help developers create Cricket applications in Java. This chapter describes our software architecture and our Java Cricket client library API called `Cricketlib`. At the end of the chapter, we show how to create a simple Cricket application in Java using `Cricketlib`. We recommend developers to use this sample application as a template to create their own Cricket applications in Java.

### 4.1 Requirements

Java must be installed on your system. Please download the latest Java software development kit (SDK) (version  $\geq 1.4$ ) from

<http://www.javasoft.com>

Our code contains `Makefile` and shell scripts to automate the compilation and program loading steps. They run readily in Linux. But to run them on a Windows system, `cygwin` needs to be installed. Please download and install the latest version of `cygwin` from

<http://www.cygwin.com>

### 4.2 Architecture

Figure 4.1 illustrates the Cricket software architecture. At the lowest layer, `cricketd` allows a Cricket host device to access the Serial Port API to configure low-level Cricket parameters and obtain raw measurements from the Cricket hardware device (see Section 2.2). Our software package includes a `CricketDaemon` server application that connects to `cricketd` to filter and process raw Cricket measurements to infer the listener's spatial location and compute its position coordinates. Java applications may access the processed location information via the Java Cricket client library (`Clientlib`), which interfaces between the application and the `CricketDaemon`. At runtime, one `CricketDaemon` processes location information for exactly one Cricket device. A

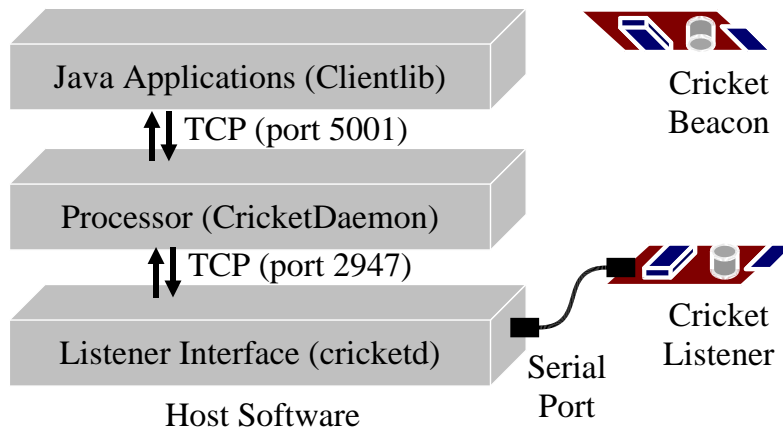


Figure 4.1: Software Architecture

CricketDaemon may serve location information of a Cricket device to numerous applications at the same time.

### 4.3 Compiling Clientlib

The Clientlib is part of CricketDaemon. A precompiled binary (cricketdaemon.jar) is located in

NMSDemo/2004/BeaconConfigDemo

You may modify the CricketDaemon source code for your own experimentation. To compile CricketDaemon, do:

```
cd src/cricketdaemon
make clean
make
```

You should change the ROOT parameter to point to the location of the Cricket software package. Also, change the SEP parameter as instructed in the Makefile. The compiled cricketdaemon.jar binary will be located in the bin directory.

*By default the scripts in our software package uses the binaries and configuration files (beacons.conf) contained in the BeaconConfigDemo directory. Be sure to copy the cricketdaemon.jar binary from the bin directory to the BeaconConfigDemo directory.*

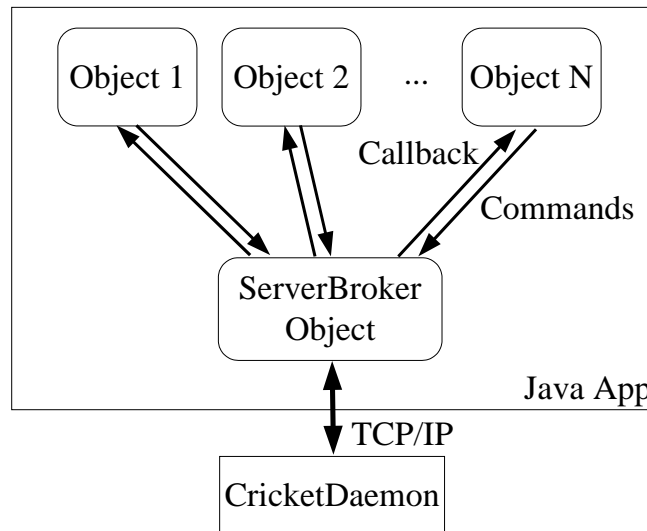


Figure 4.2: Clientlib Architecture

## 4.4 Clientlib API

The Java Cricket Client library (Clientlib) uses callbacks to feed location information to the application. As shown in Figure 4.2, a Java application instantiates one `ServerBroker` object. The `ServerBroker` object is an independent thread that interfaces between a set of callback handlers in the Cricket application and the `CricketDaemon`. At least one object in the Java application implements a callback handler and registers with the `ServerBroker`. In the callback registration, the callback handler object specifies a *callback mask* that selects the type(s) of location updates that should trigger the callback handler being registered. The `ServerBroker` dispatches the callback handlers whenever it receives a location update from the `CricketDaemon` that matches the callback mask.

### 4.4.1 The `cricketdaemon.clientlib.ServerBroker` Class

The `ServerBroker` class implements the following public methods:

```
public ServerBroker()
public ServerBroker(String addr, int portNum)
```

These are constructors for `ServerBroker`. The default constructor attempts to connect to the `CricketDaemon` running on the `localhost` via port 5001. Otherwise, the IP address of the host running `CricketDaemon` and the port it binds to may be specified through the `addr` and `portNum` parameters via the alternate constructor.

```

public boolean clearBeaconTable()
public boolean clearBeaconTableEntry(String space, int id)
public boolean setBeaconTableEntry(String space, int id,
    double x, double y, double z)

```

CricketDaemon implements a *beacon table* that maps a uniquely-identified beacon to an arbitrary coordinate value. By default, the beacon table is empty and CricketDaemon uses coordinate values advertised by the beacon if it cannot find a matching entry in the beacon table. But if a beacon does not advertise its coordinates and it has no entry in the beacon table, the distance measurements from it will not be used to compute the position coordinates of the listener.

The *\*BeaconTable\** methods manipulate the beacon table. A beacon is uniquely identified by its *space* and *id* values.<sup>1</sup> As the name of the methods suggests, the `clearBeaconTable` method clears the entire beacon table, `clearBeaconTableEntry` method clears an entry for a specific beacon and `setBeaconTableEntry` maps a beacon with *space* and *id* to coordinates (*x*, *y*, *z*).

The return `boolean` value indicates whether the command succeeded (`true`) or failed (`false`).

```

public boolean setPositionAggressive(boolean v)

```

By default, CricketDaemon uses the mode distance value collected from each beacon within its history window (see `FILTER_WINDOW_MS` in Section 3.4) to compute the position coordinates of the listener. When this method is invoked with `v=true`, CricketDaemon will use only the last measurement taken from each beacon heard in the history window to compute the position of the listener. This will reduce the latency of position tracking at the expense of using unfiltered distance estimates to compute the location of the listener.

The return `boolean` value indicates whether the command succeeded (`true`) or failed (`false`).

```

public synchronized void register(Callback c, BitSet fieldMask)
public synchronized void deRegister(Callback c, BitSet fieldMask)

```

These methods registers/deregisters a callback object with the `ServerBroker`, where *c* is a callback object that implements the `Callback` interface and *fieldMask* specifies the type(s) of location update that should trigger a callback. Table 4.4.1 lists the available callback type bits.

*By default, all distance/coordinate values are in centimeter units, unless the unit is changed via the Serial Port API described in Section 2.2.*

The `POSERR` type bit requires further explanation. `ServerBroker` invokes the callback handler method (see next section) with the `POSERR` bit set when the position solver in `CricketDaemon` cannot solve the listener's position. In the current implementation, the position solver requires at least 3 beacons to be within the listener's range and at least 3 of those beacons need to have sufficiently low standard deviations among the distance samples collected in the current history window (see `STDDEV_TOLERANCE_CM` in Section 3.4). If either of these conditions are not met, the callback handler will receive a callback with `POSERR` value set in the callback mask.

---

<sup>1</sup>In the near future, the API might change so that a beacon is uniquely identified by the unique hardware identifier value .

TYPE	DEFINITION
ALL	selects all types
SPACE	current spatial location
DEVICECOORD	current position coordinates
BEACONSHEARDLIST	set of beacons heard in history window
BEACONSTAT	distance measurement to a beacon
POSERR	exceptions in position computation

Table 4.1: Callback types.

```
public void run()
public void start()
```

All applications should call either the `run()` or the `start()` method to start the callback loop in `ServerBroker`. The `run()` method blocks while `start()` forks a new thread to run the callback loop.

#### 4.4.2 The `cricketdaemon.clientlib.Callback` Class

As mentioned above, all applications should have at least one callback handler object that implements the `Callback` interface in order to receive location updates from the `CricketDaemon`.

```
void callback(CricketData data, BitSet mask);
```

This method gets invoked whenever `CricketDaemon` updates a location value type that matches a type bit that has been set in the `fieldMask` parameter when the callback was registered. The updated value is packaged in the `CricketData` object. The callback mask `mask` specifies which type of value is available in `CricketData`.

Beware that the same `data` object gets passed to every callback handler. As a result, any modifications made to `data` becomes visible to another callback handler. Ideally, the `CricketData` object should be immutable. But it is not immutable for efficiency reasons. Thus, *be warned that modifications to `data` may cause unexpected errors in your applications.*

#### 4.4.3 The `cricketdaemon.clientlib.data.CricketData` Class

The callback handler accepts a `CricketData` object, which contains updates to location values computed by the `CricketDaemon`. `CricketData` implements the following observer methods for accessing location values.

*Before invoking an observer method, the callback handler should verify that the corresponding type bit is set in the callback mask. If the type bit is not set in the callback mask, the corresponding observer methods in `CricketData` will return null or an invalid value.*

```
public long getHWTimeStamp()  
public long getTimeStamp()
```

These methods return a millisecond time stamp value of the latest distance measurement that caused `ServerBroker` to generate the current location update. `getHWTimeStamp` returns the hardware time stamp from the Cricket device while `getTimeStamp` returns the time stamp from the host device running `CricketDaemon`. The `getHWTimeStamp` is more accurate because the high-level host device time stamp suffers from jitters that last up to several hundred milliseconds. The less accurate `getTimeStamp` method is provided to allow applications to set timers that use the host device's local clock.

```
public String getCurrentSpace()  
public BeaconRecord getCurrentSpaceObject()
```

A callback handler may invoke these methods when the `SPACE` type bit is set in the callback mask. The `getCurrentSpace` method returns the string describing the current space in which the listener is located (i.e., the string advertised by the beacon that is currently closest to the listener). The `getCurrentSpaceObject` method returns a `BeaconRecord` object that contains various attribute values that describe the beacon that is currently closest to the listener.

```
public ArrayList getBeaconsHeard()
```

A callback handler may invoke this method when either or both of `BEACONSHEARDLIST` or `BEACONSTAT` type bits are set in the callback mask. It returns an `ArrayList` of `BeaconRecords` containing various attribute values that describe the set of beacons that the listener has detected in `CricketDaemon`'s current history window.

```
public Position getDevicePosition()
```

A callback handler may invoke this method when either or both of the `DEVICECOORD` or `POSERR` type bits are set in the callback mask. It returns a `Position` object that describes the current location coordinates of the Cricket listener.

#### **4.4.4 The `cricketdaemon.clientlib.data.BeaconRecord` Class**

The `BeaconRecord` class contains the following attribute values associated with a beacon.

```
public String uniqueName;  
public String space;  
public Long lastUpdate;  
public DistanceStat distStat;  
public Position pos;
```

`uniqueName` is a string that uniquely identifies the beacon. Currently, this is a concatenation of the `space` and an integer. In the near future, this will be the unique hardware identifier value described in Section 2.2.

`space` is the space string advertised by the beacon.

`lastUpdate` is a millisecond hardware time stamp of the last distance measurement heard from this beacon.

`distStat` is a `DistanceStat` object representing the statistics of the distance samples from this beacon in `CricketDaemon`'s history window.

`pos` is a `Position` object representing the coordinate location of the beacon. It is null if the beacon does not advertise its coordinates and if there is no entry for this beacon in the beacon table (see 4.4.1).

#### 4.4.5 The `cricketdaemon.clientlib.data.DistanceStat` Class

`DistanceStat` represents the statistics of the distance samples from this beacon in `CricketDaemon`'s history window. It contains the following fields:

```
public double dist;  
public double median;  
public double mean;  
public double mode;  
public double max;  
public double min;  
public double stddev;  
public double variance;  
public ArrayList samples;
```

`dist` is the filtered distance estimate computed by `CricketDaemon`. By default, this is the mode value of the distance samples collected in the history window.

`median` is the median value of the distance samples collected in the history window.

`mean` is the mean value of the distance samples collected in the history window.

`mode` is the mode value of the distance samples collected in the history window.

`max` is the maximum value of the distance samples collected in the history window.

`min` is the minimum value of the distance samples collected in the history window.

`stddev` is the standard deviation of the distance samples collected in the history window.

`variance` is the variance of the distance samples collected in the history window.

`samples` is an `ArrayList` of `Samples` containing all the measured distance samples from a beacon currently in `CricketDaemon`'s history window.

#### 4.4.6 The `cricketdaemon.clientlib.data.Position` Class

`Position` represents a coordinate location. It contains the following fields:

```
public double x, y, z;
```

where `x`, `y`, `z` are the (x, y, z) coordinate values (default unit=centimeters, unless changed by the Serial Port API described in Section 2.2). More specifically, it has the same unit as the distance unit that the Cricket device is configured to use. In addition, the `Position` class implements the following methods:

```
public boolean equals(Object o)
```

Returns true if object `o` is a `Position` with the same (x, y, z) coordinate.

```
public double dist(Position o)
```

```
public double dist(double ox, double oy, double oz)
```

These methods compute and return the distance between position `o` or the specified coordinates (`ox`, `oy`, `oz`) and the position represented by this object.

```
public boolean invalid()
```

Returns true if this position is invalid. (Typically an invalid listener position value is also indicated by the `POSERR` bit in the callback mask.)

```
public static Position weightSum(double w1, Position p1,  
                                double w2, Position p2)
```

Returns a `Position` that is a weighted sum of two positions:  $w1 * p1 + w2 * p2$ .

#### 4.4.7 The `cricketdaemon.clientlib.data.Sample` Class

`Sample` represents a distance measurement sample with respect to a beacon.

```
public long time;
```

```
public double dist;
```

`time` is the hardware time stamp of the measurement generated by the Cricket device.

`dist` is the measured distance (default unit=centimeters, unless changed by the Serial Port API described in Section 2.2).

## 4.5 Using Clientlib: An Example

Our software package includes a template application called `ClientlibExample` to help developers create Cricket applications in Java. In this section, we will explain how to use this template application to create a Java application that prints the current space and coordinate location of the listener. You can find the `ClientlibExample` source code under the path:

```
Cricket/src/app/cricketlibexample
```

### 4.5.1 Source Code

The `ClientlibExample.java` file contains all the source code for the sample application. For convenience, we list the source code here. The code includes instructive comments that explain how to use this template to create your own Cricket application.

```
package clientlibexample;

import gnu.getopt.*; // OPTIONAL: package that process command line args
import java.util.*;

// All Cricket apps should include the following two paths
import cricketdaemon.clientlib.*;
import cricketdaemon.clientlib.data.*;

/**
 * An example of how to use the CricketDaemon Java clientlib.
 *
 * It simply prints a line containing space and position
 * information whenever they are updated by the CricketDaemon
 * processing stack.
 *
 * @author Allen Miu
 */

// A Cricket callback handler class implements
// the Callback interface
class ClientlibExample implements Callback
{

/**
 * Handle to cricket
 */
Broker cricket;
```

```

/**
 * Data structures
 */
String currentSpace = null;
Position currentPosition = null;

public ClientlibExample()
{
/**
 * Established TCP connection to the CricketDaemon running on
 * localhost. The ServerBroker will trigger a callback
 * whenever the CricketDaemon decides to push information out to
 * the clients. An application may the alternate
 * ServerBroker constructor:
 *
 * public ServerBroker(String addr, int portNum)
 *
 * to connect to a CricketDaemon running on <code>addr</code>
 * (may be numeric IP or hostname), using portNum (normally,
 * should use 5001).
 */
    cricket = new ServerBroker();

/**
 * Create a bitmask specifying the type of Cricket information
 * that should trigger a callback on this object. The following
 * table shows all the valid bit types:
 * ALL                selects all types
 * SPACE              current spatial location
 * DEVICECOORD        current position coordinates
 * BEACONSHEARDLIST  set of beacons heard in history window
 * BEACONSTAT         distance measurement to a beacon
 * POSERR             exceptions in position computation
 */
    BitSet mask = new BitSet();
    mask.set(Broker.SPACE);
    mask.set(Broker.DEVICECOORD);
    mask.set(Broker.POSERR);

/**
 * Register this object's callback handler with ServerBroker.
 * An application can create and register multiple handlers.
 * Here, we register only one handler.
 */
    cricket.register(this, mask);
}

```

```

    cricket.start(); // fork thread, runs forever
    System.out.println("ClientlibExample created");
}

/**
 * The ServerBroker object invokes the callback handler
 * with location updates contained in <code>data</code> and
 * <code>mask</code> indicates which location type(s) is(are)
 * readable in \texttt{CricketData}. All Cricket callback handler
 * classes must implement this method.
 */
synchronized public void callback(CricketData data, BitSet mask)
{
    boolean update = false;

    // check for space updates
    if(mask.get(Broker.SPACE)) {
        BeaconRecord cs = data.getCurrentSpaceObject();
        if(cs != null && cs.uniqueName != null) {
            if(currentSpace == null || !(currentSpace.equals(cs.uniqueName))) {
                // if space string is different from before, update it
                update = true;
                currentSpace = cs.uniqueName;
            }
        }
    }

    // check for coordinate updates
    if(mask.get(Broker.DEVICECOORD) || mask.get(Broker.POSERR)) {
        Position p = data.getDevicePosition();
        if(p != null) {
            if(currentPosition == null || !(currentPosition.equals(p))) {
                if(p.invalid()) {
                    // CricketDaemon cannot produce a position estimate
                    // because it does not hear enough different beacons
                    System.err.println("ignoring invalid position");
                }
                else {
                    // if the coordinates are different from before,
                    // update them
                    update = true;
                    currentPosition = p;
                }
            }
        }
    }
}

```

```

    }

    if(update)
        printState();
}

/**
 * Prints the current space and coordinates of the Cricket listener
 */
private void printState()
{
    System.out.println("space="+currentSpace+" pos="+currentPosition);
}

public static void main(String[] args)
{
    /* OPTIONAL: example of gnu-style getopt to parse cmd line args */
    Getopt g = new Getopt("ClientlibExample", args, "c:");
    int c;
    try {
        while((c = g.getopt()) != -1) {
            switch(c) {
                case 'c':
                    System.out.println("wassup! "+g.getOptarg());
                    break;
            }
        }
    }
    catch (Exception e) {
        //usage();
    }

    // The ClientlibExample constructor launches the ServerBroker
    // thread so all we have to do is to invoke it.
    ClientlibExample client = new ClientlibExample();
}
}

```

## 4.5.2 Compiling ClientlibExample

The ClientlibExample application includes a Makefile to compile the source code. Before compiling, modify the ROOT and SEP variables as instructed in the Makefile. The binary will be located in

```
src/app/clientlibexample
```

The most important detail in compiling a Java Cricket application is to include the `cricketdaemon.jar` in your classpath. You can find this file in

```
NMSDemo/2004/BeaconConfigDemo/cricketdaemon.jar
```

In addition, you may find the `gnu.getopt.jar` file in the `lib` directory if the package is used to process command line arguments.

For example, you may compile `ClientlibExample` by

```
cd src/app/clientlibexample
export CD="../../NMSDemo/2004/BeaconConfigDemo/cricketdaemon.jar"
export GO="../../lib/gnu.getopt.jar"
javac -classpath "$CD:$GO" *.java
```

Then, create a jar file by

```
cd src/app
jar cvf clientlibexample.jar clientlibexample/*.class
mv clientlibexample.jar clientlibexample
```

### 4.5.3 Running ClientlibExample

To run the compiled `ClientlibExample` application, we must include the following files in the classpath:

```
cricketlibexample.jar
cricketdaemon.jar
gnu.getopt.jar
```

The `ClientlibExample` application includes a script called `launch.sh` that shows how to include these files in the classpath and launch the application. Please modify the `ROOT` variable as instructed in the script file.

Before running `ClientlibExample`, you need to first run `cricketd` and `CricketDaemon`. Furthermore, at least 3 beacons must be placed according to the instructions in Section 3.3 and their coordinates must be configured.

Please refer to Section 2.2.1 for instructions to run `cricketd`. To launch `CricketDaemon` and configure beacon coordinates, please follow the instructions in Chapter 3.

*Note: As explained in Section 4.2, applications that use processed location information of the same listener device should connect to the same `CricketDaemon` server. In our example, both `BeaconConfigDemo` and `ClientlibExample` connects to the same instance of `CricketDaemon` server.*

Alternatively, you may hard code the beacon coordinates in a file and use the `launch-cricketdaemon.sh` script to run `CricketDaemon`. Please modify the user-defined

parameters in the script before running the script. Section 3.4 explains the function of the parameters in the script. The hard coded coordinate values are specified by the file indicated in the `COORDINATE_FILE` parameter. Please see `bin/beacons.conf` for a sample coordinate file.

To summarize, you may launch `ClientlibExample` (after running `cricketd` on the listener's host device and modifying the parameters in the `launch*.sh` scripts) by doing the following:

```
cd NMSDemo/2004/BeaconConfigDemo
./launchall.sh &
cd src/app/clientexampleapp
./launch.sh
```

For the “hard-coded coordinates” approach, modify `bin/beacons.conf` and do:

```
cd src/app/clientexampleapp
./launch-cricketdaemon.sh &
./launch.sh
```

The following is a snippet of the output from executing the `ClientlibExample` application. The first three lines shows that the `ServerBroker` correctly established a connection with the `CricketDaemon`.

```
ServerBroker::connect() connecting to CricketDaemon at 127.0.0.1
CricketDaemon accepting new connection from /127.0.0.1:32864
ServerBroker::connect() connected
ClientlibExample created
space=C-32 pos=null
space=C-32 pos=( 110.68068181818181 51.996363636363654 74.20246458136243 )
space=C-32 pos=( 105.02765151515152 49.79030303030304 70.9467680654887 )
space=A-32 pos=( 105.02765151515152 49.79030303030304 70.9467680654887 )
space=A-32 pos=( 99.99583333333334 47.826666666666675 67.46006180099782 )
space=A-32 pos=( 84.18386363636364 39.73272727272729 70.51112894364283 )
space=A-32 pos=( 71.51304924242424 49.99484848484849 77.07008845356134 )
```

Applications can expect some delay in receiving location updates due to the sliding window filter used in `CricketDaemon`. We can reduce this delay by specifying a small value for the `FILTER_WINDOW_MS` variable (see Section 3.4) in the `launch-cricketdaemon.sh` script. In addition, the application can use the `setPositionAggressive` method described in Section 4.4.1 to reduce the position estimation delay.

# Bibliography

- [1] BALAKRISHNAN, H., ET AL. Lessons from Developing and Deploying the Cricket Indoor Location System. Available from <http://cricket.csail.mit.edu/>, November 2003.
- [2] <http://www.cygwin.org/>.
- [3] Familiar linux distribution. <http://familiar.handhelds.org>.
- [4] HARTER, A., HOPPER, A., STEGGLES, P., WARD, A., AND WEBSTER, P. The Anatomy of a Context-Aware Application. In *Proc. 5th ACM MOBICOM Conf.* (Seattle, WA, Aug. 1999).
- [5] PRIYANTHA, N., CHAKRABORTY, A., AND BALAKRISHNAN, H. The Cricket Location-Support System. In *Proc. 6th ACM MOBICOM Conf.* (Boston, MA, Aug. 2000).
- [6] <http://webs.cs.berkeley.edu/tos/>.
- [7] WANT, R., HOPPER, A., FALCAO, V., AND GIBBONS, J. The Active Badge Location System. *ACM Transactions on Information Systems* 10, 1 (January 1992), 91–102.