O'REILLY®

Getting Started with OpenBTS

BUILD OPEN SOURCE MOBILE NETWORKS

Michael Iedema
Foreword by Harvind Samra

Getting Started with OpenBTS

Michael Iedema

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Getting Started with OpenBTS

by Michael Iedema

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian MacDonald
Production Editor: Melanie Yarbrough
Copyeditor: Lindsy Gamble
Proofreader: Charles Roumeliotis
Indexer: WordCo Indexing Services
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January 2015: First Edition

Revision History for the First Edition:

2015-01-12: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491910658* for release details.

ISBN: 978-1-491-91065-8

[LSI]

Table of Contents

Foreword

As one of the original developers of the OpenBTS project, I'm excited to see this book become a reality. It all started in our living rooms; we were not telecom people—we were digital signal processing (DSP) software guys who had been doing GSM stuff, started playing with some early software defined radios, got introduced to VoIP, and just wanted to see if we could build our own mobile network for less than the GDP of a South Pacific island.

Since the original source code release in 2008, it's been astounding to see the widespread adoption and interest in OpenBTS, which now helps provide cellular service on all seven continents (including several islands) and has found its way into innumerable labs and universities. It has really grown into a force for change in telecommunications.

Typically, when I'm asked about the value or importance of OpenBTS, the discussion turns to the technical aspects of software defined radios and replacing legacy telecom protocols with IP, etc. But why is OpenBTS—and this book, by extension—really important?

Nowadays, few would argue that mobile telephony is not an essential utility—for many people, as indispensable as water, electricity, and gas. But the world of mobile network infrastructure today is a monolithic "black box." It has de facto closed, complex systems that few suppliers provide, which are run on pre-Internet protocols and architectural concepts that have evolved in increments over 40 years to do a specific job—but not economically, and with little flexibility.

So OpenBTS is really about empowerment. You can build a cell network. You can operate a cell network. You can learn how cell networks work. The technologies needed to build a network are no longer prohibitively expensive, nor has a handful of big organizations locked them down.

Enjoy getting started on your journey into the world of cellular networks.

— Harvind Samra, Cofounder of the OpenBTS project, cofounder and CTO of Range Networks

Introduction

Telephones are cool. Yes, smartphones are cool, too, but I'm talking about plain old two-wire, curly corded telephones. The ability to transmit your voice between any two points on Earth is an amazing human accomplishment.

Perhaps even more amazing is how quickly this accomplishment has been taken for granted. Monumental efforts were undertaken over the past 100 years to build the public switched telephone network. Webs of copper were hung and buried. Long distance lines between cities and then towns were laid. Humans manually routed and connected calls, then analog machines and finally digital computers did that automatically. Along the way, mobile networks were invented and deployed. The same infrastructure story took place as technologies advanced: equipment was upgraded, mobile phones repeatedly replaced.

Now, with both wired and wireless telephony networks delivering voice service solidly for decades, the next upgrade cycle is under way for data bandwidth: fiber optic to your home and LTE to your smartphone.

OpenBTS bridges these two worlds. By converting between the wireless radio interface and open IP protocols, it allows anyone with IP connectivity to deploy a mobile network.

Many places on Earth still do not have home telephone lines or mobile network reception. But, more often than not, they do have an Internet connection via satellite or long-haul WiFi. Properly integrated, OpenBTS can convert and distribute this Internet connection as a mobile network across a large geographic region. Any GSM phone can connect and use voice services or SMS, even basic data. Connectivity can be brought to remote regions while skipping the entire cycle of infrastructure build-out and upgrades.

The combination of OpenBTS and software defined radios changes the way we should think about mobile networks. This new technology allows the construction of complex radio networks purely in software. OpenBTS is a C++ application that implements the GSM stack. As new features are implemented or protocol support added, an existing OpenBTS mobile network's capabilities can be enhanced via a simple software update!

Also, because OpenBTS is just software, you can make it do whatever you'd like. You no longer need a hardware vendor's permission to access its closed black box implementations. You can build a niche product or experimental feature; the mobile network is finally open for innovation.

Who Should Read This Book

Telecom engineers—wired or wireless, circuit-switched, or packet-switched—should be able to latch onto this introduction to the OpenBTS project. At the risk of spreading the material too thin, care has been taken to explain both the radio and IP sides of OpenBTS. If you are a radio frequency (RF) expert, you will learn something about Internet telephony. If you are comfortable with SIP and RTP, you will pick up a thing or two about radio systems and protocols.

Software engineers of baseband firmware, smartphone apps, or hosted services will learn about how the mobile network itself can now be controlled and inspected at a very low level. If you're interested in debugging an application on a mobile device, OpenBTS provides several raw interfaces to see exactly what's going on over the air. There are also new data APIs your software can consume to build applications for search-and-rescue, emergency response, power optimization, roadway traffic analytics, etc.

Why I Wrote This Book

My background is mainly in VoIP. When I began working with OpenBTS, I was blissfully unaware of how complex radio systems can be. Conversely, other people I was working with were radio experts but had never touched VoIP. Documentation for the OpenBTS project is plentiful but very broad to support this wide audience of interested parties; it needed simplification.

We wanted a new book that would be able to give a complete newcomer to the technology enough information to successfully set up their own network: get voice calls working, exchange some SMS messages, etc. This initial success should then build confidence and let that person set out on their own.

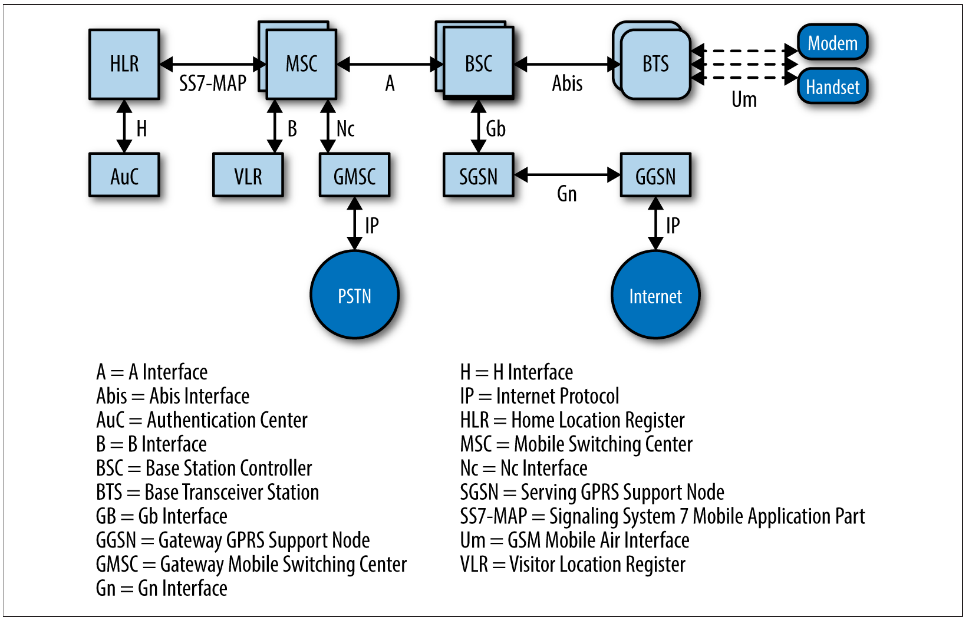
I've tried to mix a healthy amount of context into the step-by-step sections. Mobile networks are still quite complex on the GSM and RF side. Every hint helps. I'm hoping you will be able to avoid the large gotchas when setting yours up!

I also wanted the book to be interesting enough to read in the absence of hardware. This book should give you enough information to scope the required resources before diving into an OpenBTS-related project.

A Word on Mobile Networks Today

As Marc Andreessen famously stated, “Software is eating the world.” This is definitely true in the mobile industry. As processing power gets faster and cheaper, it is now viable to implement extremely complex signal processing algorithms in software. This software is also able to run on increasingly generic hardware. The days of vendor lock-in due to specialized hardware and protocols are numbered and a chance for some real competition and innovation in mobile infrastructure seems to be getting closer.

A few years ago, a “build-your-own-mobile-network” book would have been a tome for a very different audience. The words “compile” and “customize” may not even have made an appearance. The architecture for such a network is illustrated in Figure I-1.



A = A Interface
Abis = Abis Interface
AuC = Authentication Center
B = B Interface
BSC = Base Station Controller
BTS = Base Transceiver Station
GB = Gb Interface
GGSN = Gateway GPRS Support Node
GMSC = Gateway Mobile Switching Center
Gn = Gn Interface
H = H Interface
IP = Internet Protocol
HLR = Home Location Register
MSC = Mobile Switching Center
Nc = Nc Interface
SGSN = Serving GPRS Support Node
SS7-MAP = Signaling System 7 Mobile Application Part
Um = GSM Mobile Air Interface
VLR = Visitor Location Register

Figure I-1. Traditional architecture

This network architecture is an incredible piece of engineering that many people examined over many years. It is very robust and scalable, though unfortunately, quite inflexible and very expensive.

There is a growing focus on this problem at universities, most notably UC Berkeley’s Technology and Infrastructure for Emerging Regions (TIER) group and Carnegie Mellon University’s CyLab Mobility Research Center in Silicon Valley, which recently published a paper on the limitations of this traditional mobile architecture, including evolutionary baggage from the traditional wired network.

Multiple open source projects have sprung up to address this. To name a few: Osmocom, OpenLTE, and YateBTS. Each project has different goals and architectures, tackling the pain points of traditional networks in its own way.

This book, however, is about the veteran of the group, OpenBTS. The OpenBTS project is a collection of open source software components that can be used to build a more modern, lightweight network. OpenBTS allows the "Um" radio interface of the traditional mobile network to directly interconnect with Internet telephony protocols. This new "hybrid" architecture is illustrated in Figure I-2.

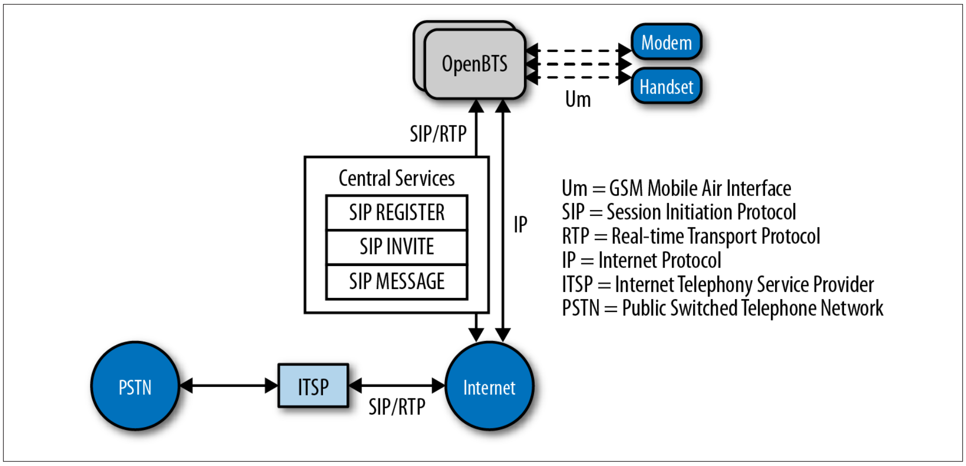

OpenBTS
Modem
Handset
Um
SIP/RTP
Central Services
SIP REGISTER
SIP INVITE
SIP MESSAGE
IP
Um = GSM Mobile Air Interface
SIP = Session Initiation Protocol
RTP = Real-time Transport Protocol
IP = Internet Protocol
ITSP = Internet Telephony Service Provider
PSTN = Public Switched Telephone Network
PSTN
ITSP
SIP/RTP
Internet

Figure I-2. Hybrid IP architecture

Software or configuration changes on the handset are unnecessary, as the radio interface to the mobile network is identical to a traditional network. The network core, however, is no longer comprised of an array of complex protocols and servers. It consists of open protocols and uses IP as its transport. Many software projects already exist that implement these open protocols. Some new components were also developed alongside OpenBTS to provide functionality, which was still unavailable, to bridge the GSM and Internet worlds.

With so much excitement around "the cloud" and so much excitement around "apps," it seems that finally the conduit between these two is beginning to bloom.

Navigating This Book

By the end of this book, you will have built a fully functional mobile network. It will appear as any other network does on your handset and route calls and SMS among the participants in the network, as well as provide data connectivity between handsets and the Internet.

- Chapter 1 guides you step-by-step through selecting radio and processing hardware, setting up the base operating system and development environment, and compiling and installing the software components.
- Chapter 2 covers initial component configuration and activation as well as network functionality tests.
- Chapter 3 dives into troubleshooting and performance-tuning techniques for production networks.
- Chapter 4 details how to expand your single-node network into a true multinode mobile network complete with Mobility and Handover.
- Chapter 5 builds on your configuration to offer general packet radio service (GPRS) data capability.
- Chapter 6 explores an OpenBTS-specific feature akin to WiFi captive portals, useful for emergency response and ad hoc networks.
- Chapter 7 provides reference material for interacting with and building applications on top of the OpenBTS NodeManager control and event APIs.
- Chapter 8 illustrates some next steps for your network: interconnecting with the public switched telephone network (PSTN), private branch eXchange (PBX) integrations, and the current state of spectrum regulation.
- Appendix A provides a quick reference for GSM terminology, RF measurements, component ports, paths, and files.
- Appendix B walks through the installation of the Ubuntu 12 operating system.
- Appendix C shows how to capture not only IP exchanges, but raw GSM radio frames as well.

Online Resources

- OpenBTS Community
- OpenBTS Documentation
- GSM Time Slot and Channel Visualizer
- OpenBTS Source Code Repository on GitHub

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley

Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472
> 800-998-9938 (in the United States or Canada)
> 707-829-0515 (international or local)
> 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/1yPbnN2*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Harvind Samra and David Burgess brought OpenBTS to life. This book would not have been possible without them. A special thanks to David for inviting me to join Range. I haven't stopped learning since!

Thanks to the community who never fails to surprise me with possible applications for OpenBTS, and to Rangers past and present who've humored my questions: Tom, Pat, John, Donald, Faith, and Doug.

My editor, Brian MacDonald, was very kind to this first-time author and provided great direction. Technical reviews from Kurtis Heimerl, Ralph Schmid, and Neel Pandeya kept the material honest. Thanks to everyone involved in that process, especially to John Callon for deflecting bureaucracy.

Manuel Kasper was the first one to introduce me to VoIP during the Free World Dialup days. Without that push, I would have never have dove in so deep to Internet telephony. Congrats to the entire Askozia team for building such a fine company from that project and to Diederich Wermser for believing in the concept from day one.

On a more personal note: I would never have been able to study abroad nor finish my CS degree without Tom Halverson of Dakota State University and Peter Riegler of the Fachhochschule Braunschweig Wolfenbüttel. Both went above and beyond their duties to make the exchange happen, which allowed me to start my IMS research. Thanks also to the Madison and Wolfenbüttel release celebration teams: Berger, Bobby, Dan, Jon, Jorj, Levi, Mitch, Phil, Seth, Steve, Dege, Markus, Milo, Raiko, Sid, and Löloff.

And finally to my family: Big G, Little M, Ry, and Norus. Let's never stop making each other laugh.

CHAPTER 1

Getting Set Up

In this chapter, we will guide you through the selection of hardware, installation of a base operating system, and development environment setup, as well as actually compiling and installing the components that compose the OpenBTS software suite. Several shortcuts can be taken along the way—for example, if you would prefer to use the official binary packages instead of compiling your own—but the entire process is detailed for those wishing to build from scratch.

Hardware Components

Although OpenBTS implements most of the complexity involved in building a mobile network in software, radio waves must still be transmitted and received somehow. This section details which hardware components you should procure to implement this capability in a development setting. The configuration of these components is covered in Chapter 2.

Linux Server

The first requirement is a standard commodity Linux server. Other architectures are beginning to be supported, but stick to an x86 processor running a 32-bit operating system for the best results for now. This computer can be a separate machine in your test environment or it can actually be a virtual machine on the laptop or desktop you use daily.

Minimum requirements for processing power and RAM are not clearly defined due to the many variables involved, such as the number of concurrent carrier signals, network load, network usage type, radio environment, etc. Each will affect the required resources.

A single carrier signal requires the OpenBTS software to generate downlink waveforms to transmit to the handset and demodulate the uplink waveforms received from the handset. OpenBTS supports the creation of multiple concurrent carrier signals on a single physical radio to linearly increase network capacity, but the processing demands are very high. For a stable lab setup with a single carrier signal (maximum of seven concurrent voice channels), an Intel i5 or something comparable with 2 GB of RAM is recommended. It must also have at least a USB2 interface, but USB3 is quickly becoming a requirement on newer software defined radios. It may be a good idea to start with USB3 now to avoid a future upgrade.

The need for increased throughput on the USB interface relates to the quantity and size of radio waveform samples being communicated through it. In a production environment, multiple simultaneous carrier signals can be utilized, which drastically increases the required sample bandwidth. Additionally, the algorithms used to demodulate signals can be configured to operate in a more robust manner (e.g., to rectify signal distortion due to the deployment environment). Thus, the processing power needed to generate and demodulate these signals could be an order of magnitude greater than in a lab setup.

Another thing to keep in mind is that as newer releases of OpenBTS are made available, new capabilities may require more processing power or memory. For example, OpenBTS in Global System for Mobile (GSM) mode can run smoothly on an Intel Atom processor but the new OpenBTS-UMTS (Universal Mobile Telephone System) requires at least an Intel i7. This sidenote is actually the main advantage of having a radio access network (RAN) defined in software. As new standards are released, they can be applied to a production RAN via a simple software update instead of swapping expensive hardware infrastructure.

Software Defined Radio

The software defined radio (SDR) is the key breakthrough that makes OpenBTS possible from a hardware perspective. SDRs have been used in military applications for about 20 years. Only recently have they become available to a wider audience due to the decreasing cost of the technology.

A modern SDR is a piece of hardware that connects to your computer via a USB cable or over Ethernet. They are typically as small as a deck of cards or as large as a stack of three DVDs and are powered either directly via the USB host connection or by a small external power supply. The SDR hardware implements a completely generic radio that can send and receive raw waveforms in a defined frequency range (i.e., 60 MHz to 4 GHz) with a host application. That host application could be an FM radio implementation that receives the raw waveforms, demodulates the signal, and plays back the audio. This means that radio hardware is no longer designed around a specific application; it is completely up to the host to define the implementation, allowing anyone to create radio applications purely in software.

OpenBTS supports SDRs from several vendors: Ettus Research, Fairwaves, Nuand, and Range Networks. These products range in price from approximately $500 to over $2500.

If you choose a product that connects via Ethernet, make sure your Linux server has a dedicated Ethernet port for the radio (preferably gigabit Ethernet).

Most SDRs are completely generic hardware suitable for any radio project while some are specifically optimized for implementing mobile networks. Check with the vendor before purchasing.

Antennas

Many SDRs have enough transmit and receive sensitivity to operate without antennas in a small environment. Typically, a coverage area with a radius of 1 m can be achieved. This is a desirable setup for a lab environment, especially if multiple developers are using multiple radios. The coverage areas will not overlap and interfere with each other. Additionally, your network will not interfere with any carriers in the area.

Even if your coverage area is very small, the national regulator will still most likely require you to obtain a test license before using GSM frequencies. More information is in "Spectrum Regulation" on page 78.

In lab environments where a single OpenBTS instance is to be shared among multiple developers, the coverage area must be expanded. Adding a pair of small 5 dBi antennas can increase it dramatically, up to a 25 m radius in an unobstructed environment. These are usually rubber duck–style antennas with a SubMiniature version A (SMA) connector, similar in appearance to a typical home WiFi router antenna. An example is shown in Figure 1-1.

Antennas are tuned for a specific frequency, so choose one that most closely matches the GSM band you will be using (850, 900, 1800, or 1900 MHz).

The frequency also plays a role in the coverage area size. Low-frequency bands (850 and 900 MHz) propagate larger distances than high-frequency bands—sometimes nearly twice as far.

Test Phones

For testing, at least two GSM handsets compatible with the band you will be using (850, 900, 1800, or 1900 MHz) are needed. Most modern GSM handsets are "quad-band," meaning that all bands are supported. There are also "tri-band" and "dual-band" handsets. The easiest way to determine if a handset will be band compatible with your network is to search for the model on GSM Arena. Complete technical specifications are listed there.

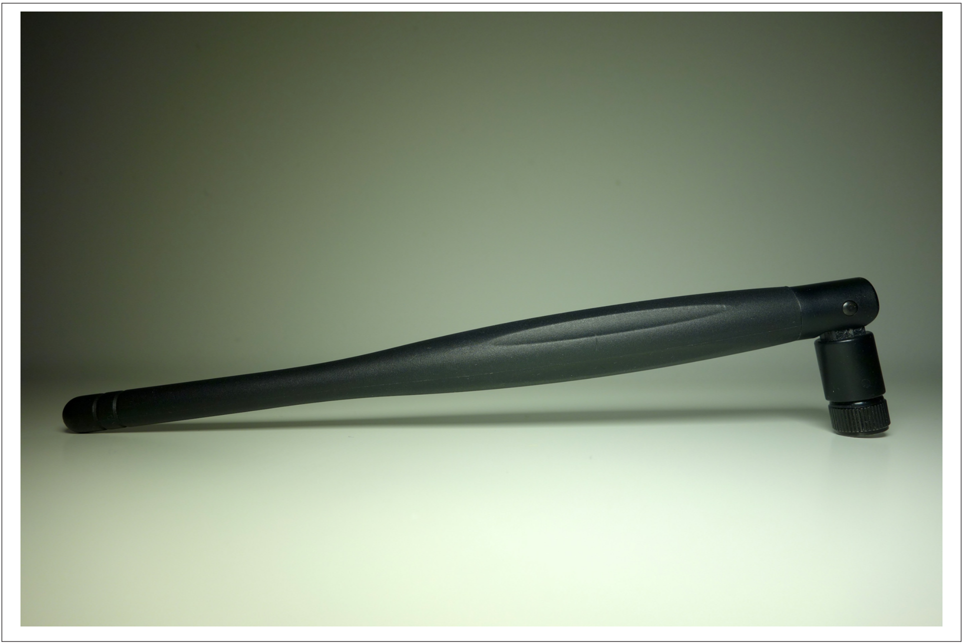


Figure 1-1. Rubber-duck antenna

You must also make sure that the handset you are using is unlocked. If the handset is "locked," it means that the manufacturer has programmed the hardware's baseband processor to only work with a specific carrier. This restriction can be removed, usually by entering a sequence of numbers on the dialpad, but that is beyond the scope of this book. The easiest choice is to use an unlocked handset that will accept any carrier's subscriber identity module (SIM) card.

Test SIMs

The SIM used in GSM handsets is nothing more than a trimmed-down smart card. Smart cards are also known as chip cards or integrated circuit cards (ICCs) and can be found in many authentication and identification applications. A full-size smart card has the same dimensions as a credit card; newer credit cards actually use smart card technology to increase security. SIM cards are programmed using standard smart card writers and, once written, are popped out of the full-size card frame so they fit into a GSM handset. Figure 1-2 shows the full-size smart card carrier, popped out full-size SIM card, and trimmed-down micro SIM card.

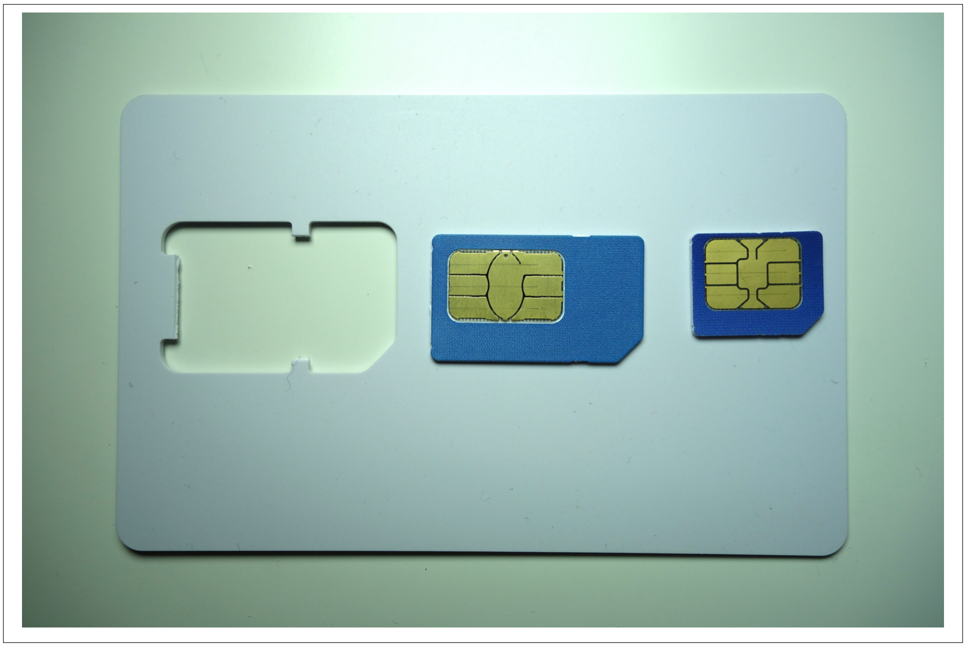


Figure 1-2. Smart card frame, full-size SIM, and micro SIM

There are a couple of ways to procure SIMs for use in your test environment. Depending on your needs and future deployment plans, you can decide to utilize spare SIMS, create some new ones yourself, or contract a company to program them in bulk.

Using spare SIMs

For testing purposes, any SIM card that physically fits into the handset will probably work. However, using your existing SIM, an expired SIM, or a foreign country's SIM has trade-offs, mainly concerning security features. There is a secret key, named "Ki," stored in the SIM, which only the SIM issuer knows. It is stored in an unreadable portion of memory in the SIM and is not retrievable after the manufacturer places it there. This shared secret between operator and handset is what allows the network to determine cryptographically if the subscriber is valid.

OpenBTS has an alternative authentication method for this situation known as "Cache Based Auth." It performs an initial authentication exchange with the handset and records the results. It uses this same request and expects the same answer in the future. The method is not as secure as unique exchanges for each request but is still better than completely disabling authentication.

Another key, named "Kc," is used to support call encryption. When using another carrier's SIM in your test environment you will lose these two features, but if you do not have access to a smart card writer and blank cards, this is your best option.

Some handsets are tricky when using another carrier's SIM to connect to your own OpenBTS network. If the handset can detect its "home" carrier signal, the option to manually select another carrier may be hidden from the settings menu. This is commonly observed when using AT&T SIMs in unlocked iPhones.

DIY SIMs

To create a small batch of SIM cards for your own test environment, you will need blank, rewritable "magic" SIMs and a smart card writer. Many products are available from several vendors on Alibaba's website. An example writer connected via USB is pictured in Figure 1-3, but there are many different-looking writers on the market.

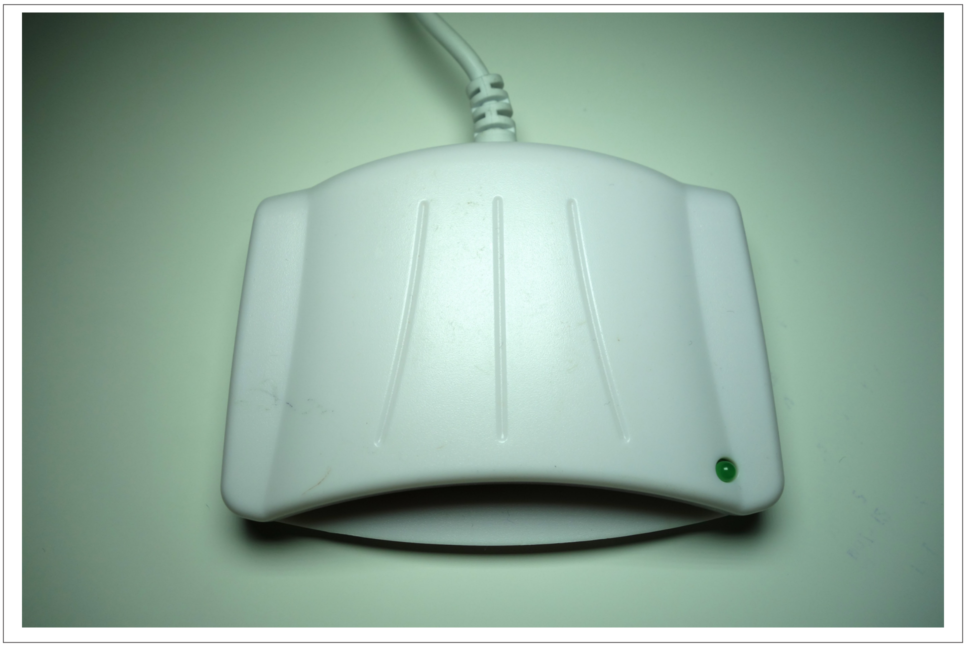


Figure 1-3. Smart card writer

The software installation and write commands depend on which writer and cards have been selected. The most up-to-date information about writing SIM cards can be found on the OpenBTS wiki page.

Batch SIMs

SIM-writing companies are starting to offer lower quantity runs of cards for reasonable prices. You will typically submit some parameters about your order, such as card artwork, authentication algorithm needed, and card quantity, to receive a quotation. Once ordered, the finished product will be a box of your SIMs along with a separate document or file containing a list of IDs and encryption keys for each card.

This list can then be batch-imported into your subscriber database by using a script. Now all of your subscribers are provisioned into the system without manually burning 10,000 individual SIMs and you're left with a nicely packaged SIM card for each user.

Full-size versus micro versus nano

Once you have a technically compatible SIM card (2G versus 3G versus 4G, etc.), it may still have an incompatible form factor. This is easily remedied by using a SIM cutter, something that carrier shops love to charge decent service fees to use. It's a device that looks like a stapler but functions more like a cookie cutter, as illustrated in Figure 1-4. They can be purchased for less than $20 and will convert SIMs from standard to micro and from micro to nano. The only difference between these sizes is the amount of plastic surrounding the actual chip.

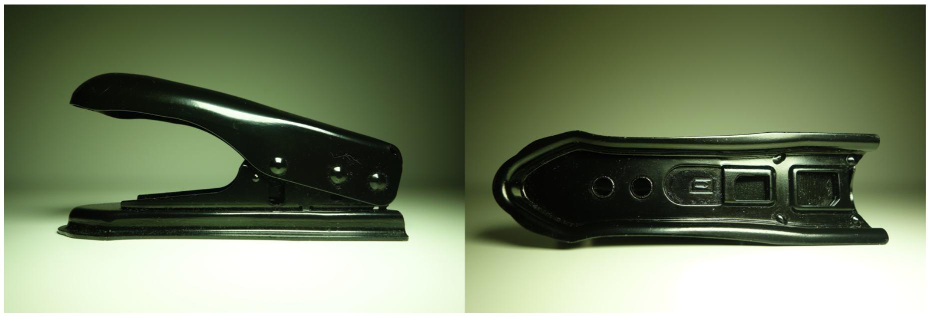


Figure 1-4. SIM cutter

Operating System and Development Environment Setup

Now that the hardware has been gathered, you are ready to proceed with setting up a development environment. OpenBTS has traditionally been developed and tested on Ubuntu Long-Term Support (LTS) distributions. It has also been tested on Debian and CentOS distributions.

For this book, the most tested distribution and architecture will be used: Ubuntu Server 12.04 LTS 32-bit. Starting with OpenBTS 5.0, 64-bit systems are also supported but have not been as widely deployed. In addition to Ubuntu 12, the Ubuntu 13 and 14 systems have also been used successfully. Preliminary packaging for RPM-based systems (CentOS, Fedora, and Red Hat Enterprise Linux) is also available but will not be covered in this book. Please visit OpenBTS.org for more information.

If you do not yet have a compatible operating system installed, Appendix B has a complete step-by-step guide.

If you plan on using the official OpenBTS release packages, feel free to skip ahead to "Installation" on page 10.

Git Compatibility

Git is a version-control system that manages software source code changes. The OpenBTS project utilizes several new features in Git, such as submodule branch tracking. To make sure your client is compatible (e.g., newer than 1.8.2), it needs to be updated.

First, execute this command to add support for Personal Package Archives, an alternate way to distribute binary release packages:

```
$ sudo apt-get install software-properties-common python-software-properties
```

Then, execute the following command to add a repository for the latest Git builds to your system:

```
$ sudo add-apt-repository ppa:git-core/ppa
```

Now, you must simply refresh the list of packages and install Git again to update your system's client:

```
$ sudo apt-get update
$ sudo apt-get install git
```

To confirm that the new Git client is installed properly, run the following command:

```
$ git --version
git version 1.9.1
```

Now that you have Git installed, you can proceed to downloading the development scripts.

Downloading the Code

The OpenBTS project consists of multiple software components hosted in separate development repositories on GitHub. Understanding the intricacies of Git should not be a barrier to using OpenBTS, so several development scripts have been written to make it easier to download the code, switch branches, and compile components. To download these development scripts into your new environment, run the following command:

```
$ git clone https://github.com/RangeNetworks/dev.git
```

The development scripts assume that you have Secure Shell (SSH) keys set up for GitHub. If you do not, please follow these instructions to set them up before proceeding.

Now, to download all of the components, simply run the `clone.sh` script:

```
$ cd dev
$ ./clone.sh
```

Each component's repository will be cloned from GitHub into your development environment. The `clone.sh` script also automatically initializes any submodules needed.

Now that the OpenBTS project sources are in your development environment, you can select a specific branch or release to compile. The `switchto.sh` script is used to toggle between build version targets. For example, if you wanted to build the v4.0.0 release, run the following command:

```
$ ./switchto.sh v4.0.0
```

The current version target can be listed for each component by using the `state.sh` script. This script also lists any outstanding local changes for each component.

```
$ ./state.sh
```

This book focuses on the 5.0 series branch. To target the latest, greatest code in 5.0, run the following command:

```
$ ./switchto.sh 5.0
```

Building the Code

Your development environment is now prepared to build the newest bits in the 5.0 series branch. To compile binary packages, you will use the `build.sh` script. It automatically installs the compiler and autoconfiguration tools as well as any required dependencies. It also controls which radio transceiver application will be built. As there are several different drivers available for the various radio types, `build.sh` requires an argument so it knows which hardware is being targeted (valid radio types are SDR1, USRP1, B100, B110, B200, B210, N200, and N210).

OpenBTS also supports the UmTRX hardware from Fairwaves. However, it uses a custom version of the UHD radio driver and the `build.sh` script has not yet automated its installation.

This book targets the Ettus Research N200. Run the build command now:

```
$ ./build.sh N200
```

This process can take a while (30–60 minutes) the first time it is run, depending on the hardware it is being executed on. Going forward, you will only need to recompile updated components. The following command, for example, recompiles just the OpenBTS package for an Ettus Research B100 radio:

```
$ ./build.sh B100 openbts
```

When the build script finishes, you will have a new directory named "BUILDS" containing a subdirectory with the build's timestamp. An example listing of this directory follows:

```
$ ls dev/BUILDS/2014-07-29--20-44-51/*.deb
liba53_0.1_i386.deb                   range-asterisk-config_5.0_all.deb
libcoredumper1_1.2.1-1_i386.deb       range-configs_5.0_all.deb
libcoredumper-dev_1.2.1-1_i386.deb    sipauthserve_5.0_i386.deb
openbts_5.0_i386.deb                  smqueue_5.0_i386.deb
range-asterisk_11.7.0.4_i386.deb
```

Congratulations! You can now move on to installing and starting each component, as well as learning what purpose each serves.

Installation

Now that you've downloaded a set of official release packages or compiled your own, they need to be installed and started. A bit of background for each component will be provided, followed by the installation procedure and any initializing configuration needed. As some components depend on others, they will be presented in the order needed to satisfy these interdependencies.

Change into your new build directory before continuing:

```
$ cd dev/BUILDS/2014-07-29--20-44-51/
```

Installing Dependencies

If you compiled your own set of packages in the previous section using the `build.sh` script, these dependencies have already been installed and this section can be skipped over. If you're using a set of official release packages, you'll need to install some additional system libraries and define an additional repository source so all dependencies can be found and installed.

Execute the following commands to define an additional repository source for ZeroMQ, a library that all the components use:

```
$ sudo apt-get install software-properties-common python-software-properties
$ sudo add-apt-repository ppa:chris-lea/zeromq
$ sudo apt-get update
```

Coredumper library

OpenBTS uses the coredumper shared library to produce meaningful debugging information if OpenBTS crashes. Google originally wrote it and there are actually two libcoredumper packages: `libcoredumper-dev` contains development files needed to compile programs that utilize the coredumper library, and `libcoredumper` contains the shared library that applications load at runtime:

```
$ sudo dpkg -i libcoredumper1_1.2.1-1_i386.deb
```

The exact version numbers found in the package names may have changed since the publishing of this book.

A5/3 library

OpenBTS uses the A5/3 shared library to support call encryption. It contains cryptographic routines that must be distributed separately from OpenBTS:

```
$ sudo dpkg -i liba53_0.1_i386.deb
```

Installing Components

By installing all of the following components on a fresh system, you are guaranteed a functional GSM network-in-a-box. Everything needed for voice, SMS, and data will be running in a single system.

The overall architecture of what you will be installing is visible in Figure 1-5. The Session Initiation Protocol (SIP) and Real-time Transport Protocol (RTP) are the two protocols that OpenBTS uses to convert GSM traffic into VoIP.

System configs

This package contains a set of default configurations that will allow a fresh Ubuntu system to work out of the box when installed. It includes settings for the network interface, firewall rules, domain name system (DNS) configuration, logging, etc.

You may not want to install this package if you are already comfortable configuring a Linux distribution, but its contents can serve as a guide for the required changes. During installation, you will be prompted several times to confirm the overwriting of certain configuration files. If you are unsure what the file does, a safe answer is always "Y" when dealing with a fresh system:

```
$ sudo dpkg -i range-configs_5.0_all.deb
```

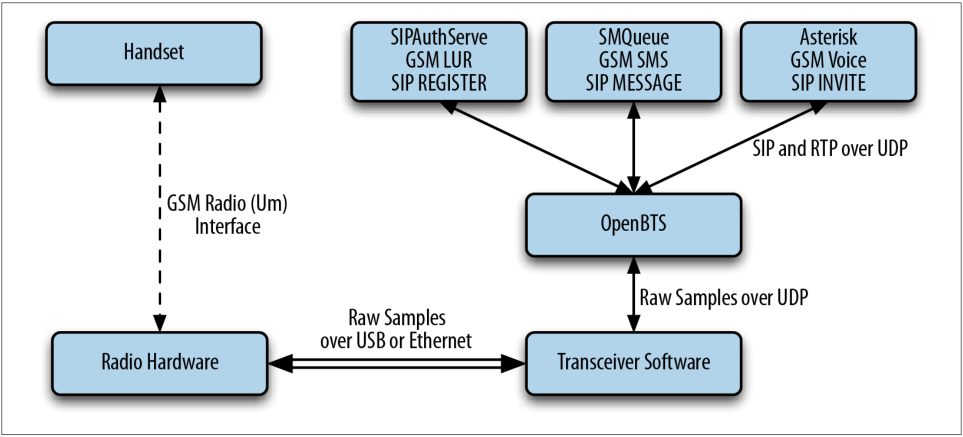


Figure 1-5. Component architecture

This package will overwrite your network interface configuration if you answer "Y" when asked about the */etc/network/interfaces* change. You can answer "N" to this prompt and your configuration will be retained: however, GPRS will not function correctly until you have manually added the text detailed in "Central Services" on page 62 to your interface definition. It is responsible for calling `iptables-restore` to automatically load firewall rules.

Asterisk

Asterisk is a VoIP switch responsible for handling SIP INVITE requests, establishing the individual legs of the call, and connecting them together. There are two packages responsible for setting up an Asterisk installation that works without any additional configuration: range-asterisk and range-asterisk-configs.

The range-asterisk package contains a confirmed-working version of the Asterisk SIP switch software and ensures that the appropriate modules needed for OpenBTS are already included. No other patches to Asterisk are included; it is simply intended to represent the latest confirmed-working version of Asterisk.

The range-asterisk-configs package contains a set of configuration files so Asterisk knows about and can communicate with the subscriber registry database. This database is where the various components store and update subscribers' phone numbers, identities, authentications, caller IDs, and registration states. Also, by using this database, it is no longer necessary to manually edit Asterisk configuration files when adding new handsets to the network:

```
$ sudo dpkg -i range-asterisk*.deb
$ sudo apt-get install -f
```

SIPAuthServe

SIP Authorization Server (SIPAuthServe) is an application that processes SIP REGISTER requests that OpenBTS generates when a handset attempts to join the mobile network.

When a handset authenticates successfully, SIPAuthServe is responsible for updating the subscriber registry database with the IP address of the OpenBTS instance that initiated it, allowing other subscribers to call the handset:

```
$ sudo dpkg -i sipauthserve_5.0_i386.deb
$ sudo apt-get install -f
```

SMQueue

SIP MESSAGE Queue (SMQueue) is an application that processes SIP MESSAGE requests that OpenBTS generates when a handset sends an SMS. It stores the messages, schedules them for delivery in the network, and reschedules them if the target handset is unavailable:

```
$ sudo dpkg -i smqueue_5.0_i386.deb
$ sudo apt-get install -f
```

OpenBTS

Finally, we reach the star of the show. OpenBTS is responsible for implementing the GSM air interface in software and communicating directly with GSM handsets over it. This communication is converted into SIP and RTP on the IP network side and interacts with the components above to form the core network.

The GSM handsets see a fully compatible GSM radio access network and the core network sees standard SIP endpoints. Neither side must know that there is a layer between allowing the handsets to connect seamlessly to the IP world:

```
$ sudo dpkg -i openbts_5.0_i386.deb
$ sudo apt-get install -f
```

Starting/Stopping Components

Now that each component has been installed, you need to start them. Components are controlled on Ubuntu with a system named Upstart. Future releases of the OpenBTS suite will support other mechanisms such as systemd, but for now, Upstart is used.

To start all components, execute the following:

```
$ sudo start asterisk
$ sudo start sipauthserve
$ sudo start smqueue
$ sudo start openbts
```

Conversely, to stop all components, use:

```
$ sudo stop openbts
$ sudo stop asterisk
$ sudo stop sipauthserve
$ sudo stop smqueue
```

The order of startup and shutdown is not critical, but if OpenBTS is running without the other components, a GSM network will be visible but unusable.

Each component runs in the background and will automatically restart if a fault arises. To monitor the console output for the component once it is running in the background, the following log files can be used:

```
/var/log/upstart/asterisk.log
/var/log/upstart/sipauthserve.log
/var/log/upstart/smqueue.log
/var/log/upstart/openbts.log
```

The system components have been installed and are running. The next step will be to start testing and configuring them.

CHAPTER 2

Initial Testing and Configuration

The software and hardware should now be in place. This chapter will guide you through some initial sanity checks, functional testing, and basic configuration customization. By the end of this chapter, you will have successfully exchanged the first SMS messages and voice calls among phones over your private mobile network!

Initial State

Some of the manual steps that follow will conflict and fail if other instances of the services are already running. To make sure that nothing else is running on this system, execute the following:

```
$ sudo stop openbts
$ sudo stop asterisk
$ sudo stop sipauthserve
$ sudo stop smqueue
```

Now you can proceed to confirm connectivity at each step in the chain before running the first basic tests.

Confirm Radio Connectivity

The first thing you should verify is that the transceiver application can communicate with the radio hardware. Different vendors have different methods for accomplishing this.

Ettus Research Radios

All Ettus hardware uses the Transceiver52M binary, which was installed in */OpenBTS* in the last chapter. Run it as follows to see if the hardware device is detected:

```
$ cd /OpenBTS
$ sudo ./transceiver
[sudo] password for openbts:
linux; GNU C++ version 4.6.3; Boost_104601; UHD_003.006.002-release

Using internal clock reference
-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes
```

The example above shows a successful attempt. The transceiver can be stopped by pressing Ctrl-C. If you instead see something like the following output, there is a problem:

```
$ cd /OpenBTS
$ sudo ./transceiver
[sudo] password for openbts:
linux; GNU C++ version 4.6.3; Boost_104601; UHD_003.007.002-release

Using internal clock reference
ALERT 1745:1745 2014-09-05T22:37:00.4 UHDDevice.cpp:528:open: No UHD devices
found with address ''
ALERT 1745:1745 2014-09-05T22:37:00.4 runTransceiver.cpp:160:main: Transceiver
exiting...
```

Ettus provides a couple of helper applications to automatically detect and inspect attached radios. Run the following command to list all attached devices. If yours does not show up, skip to "Troubleshooting USB" on page 18 or "Troubleshooting Ethernet" on page 19 to remedy the connectivity issue. This example shows an N200 attached via Ethernet:

```
$ uhd_find_devices
linux; GNU C++ version 4.6.3; Boost_104601; UHD_003.007.002-release

UHD Device 0
Device Address:
    type: usrp2
    addr: 192.168.10.2
    name:
    serial: XXXXXX
```

Another helpful application is uhd_usrp_probe, which will inspect a device and return its technical information and configuration. An example run of this application follows:

```
$ uhd_usrp_probe
linux; GNU C++ version 4.6.3; Boost_104601; UHD_003.007.002-release

-- Opening a USRP2/N-Series device...
-- Current recv frame size: 1472 bytes
-- Current send frame size: 1472 bytes

  _____________________________________________________
 /
```

```
|       Device: USRP2 / N-Series Device
|     _____________________________________________________
|    /
|   |       Mboard: N200r4
|   |   hardware: 2576
|   |   mac-addr: XX:XX:XX:XX:XX:XX
|   |   ip-addr: 192.168.10.2
|   |   subnet: 255.255.255.255
|   |   gateway: 255.255.255.255
|   |   gpsdo: none
|   |   serial: XXXXXX
|   |   FW Version: 12.3
|   |   FPGA Version: 10.0
|   |
|   |   Time sources: none, external, _external_, mimo
|   |   Clock sources: internal, external, mimo
|   |   Sensors: mimo_locked, ref_locked
|   |     _____________________________________________________
|   |    /
|   |   |       RX DSP: 0
|   |   |   Freq range: -50.000 to 50.000 Mhz
|   |     _____________________________________________________
|   |    /
|   |   |       RX DSP: 1
|   |   |   Freq range: -50.000 to 50.000 Mhz
|   |     _____________________________________________________
|   |    /
|   |   |       RX Dboard: A
|   |   |   ID: SBX (0x0054)
|   |   |   Serial: XXXXXX
|   |   |     _____________________________________________________
|   |   |    /
|   |   |   |       RX Frontend: 0
|   |   |   |   Name: SBXv3 RX
|   |   |   |   Antennas: TX/RX, RX2, CAL
|   |   |   |   Sensors: lo_locked
|   |   |   |   Freq range: 400.000 to 4400.000 Mhz
|   |   |   |   Gain range PGA0: 0.0 to 31.5 step 0.5 dB
|   |   |   |   Connection Type: IQ
|   |   |   |   Uses LO offset: No
|   |   |     _____________________________________________________
|   |   |    /
|   |   |   |       RX Codec: A
|   |   |   |   Name: ads62p44
|   |   |   |   Gain range digital: 0.0 to 6.0 step 0.5 dB
|   |   |   |   Gain range fine: 0.0 to 0.5 step 0.1 dB
|   |     _____________________________________________________
|   |    /
|   |   |       TX DSP: 0
|   |   |   Freq range: -250.000 to 250.000 Mhz
|   |     _____________________________________________________
|   |    /
|   |   |       TX Dboard: A
|   |   |   ID: SBX (0x0055)
|   |   |   Serial: XXXXXX
|   |   |     _____________________________________________________
|   |   |    /
|   |   |   |       TX Frontend: 0
|   |   |   |   Name: SBXv3 TX
|   |   |   |   Antennas: TX/RX, CAL
|   |   |   |   Sensors: lo_locked
|   |   |   |   Freq range: 400.000 to 4400.000 Mhz
|   |   |   |   Gain range PGA0: 0.0 to 31.5 step 0.5 dB
|   |   |   |   Connection Type: QI
|   |   |   |   Uses LO offset: No
|   |   |     _____________________________________________________
|   |   |    /
|   |   |   |       TX Codec: A
|   |   |   |   Name: ad9777
|   |   |   |   Gain Elements: None
```

Range Networks Radios

Range Networks SDR1 radio uses the TransceiverRAD1 binary, which was installed in */OpenBTS* in the last chapter. Run it as follows to see if the hardware device is detected:

```
$ cd /OpenBTS
$ sudo ./transceiver 1
```

If the program does not stop immediately and continues running, it has successfully detected the radio hardware. The transceiver can be stopped by pressing Ctrl-C as shown here:

```
$ cd /OpenBTS
$ sudo ./transceiver 1
^CReceived shutdown signal
```

If the program stopped immediately as shown in the following output, continue on to the Troubleshooting USB section that immediately follows to remedy the problem:

```
$ cd /OpenBTS
$ sudo ./transceiver 1
$
```

Troubleshooting USB

When using a virtual machine setup, the most common problem with USB connectivity is that the desired USB device is not associated with the correct virtual machine. Use your virtual machine's settings to find and assign the appropriate USB device to the virtual machine running your OpenBTS development environment.

Troubleshooting Ethernet

Whether using a virtual machine or real server, an extra Ethernet interface must be available for your Ethernet connected radio. While it is possible to change the IP address of the radio to match your local network, this is undesirable because the samples being exchanged over Ethernet between the transceiver application and radio hardware are very sensitive to delay and loss. They should be on a dedicated connection. Install an extra physical Ethernet interface in your server or create an additional virtual Ethernet interface in the virtual machine.

Make sure this interface is on the same subnet as the radio hardware. The default IP address for all Ettus hardware is 192.168.10.2. Assign an appropriate address to your extra Ethernet interface using the following example:

```
$ sudo ifconfig eth1 192.168.10.1/24
```

Now test the connection with a simple ping. Press Ctrl-C to stop the ping test:

```
$ ping 192.168.10.2
PING 192.168.10.2 (192.168.10.2) 56(84) bytes of data.
64 bytes from 192.168.10.2: icmp_req=1 ttl=64 time=1.037 ms
64 bytes from 192.168.10.2: icmp_req=2 ttl=64 time=1.113 ms
^C
```

Starting Up the Network

Now that you have confirmed the transceiver software can communicate with the radio hardware, you can start running the OpenBTS service in the background. Do that with the following command:

```
$ sudo start openbts
```

The OpenBTS service will automatically start an instance of the transceiver software and connect to the radio hardware. Radio samples are then exchanged between the transceiver software and OpenBTS software over a local User Datagram Protocol (UDP) socket.

The Configuration System and CLI

All configuration of OpenBTS is accomplished by manipulating keys stored in an SQLite3 database. By default, this database is stored at */etc/OpenBTS/OpenBTS.db*. Each key is defined in a schema that is compiled in OpenBTS and used to validate the values being used.

One advantage afforded by this style of configuration system is that most key values can be changed and are applied to the running system within a few seconds without interrupting service. These are dynamic keys. There are also a few static keys in the configuration system that require a restart of OpenBTS to apply the change.

The easiest way to manipulate the configuration keys is via the OpenBTS command-line interface (CLI). Run the following shell command to open it:

```
$ sudo /OpenBTS/OpenBTSCLI
```

You are now presented with an OpenBTS prompt. This is where commands, including configuration changes, can be executed for processing by OpenBTS.

From now on, commands prefixed with `$` are to be executed on the Linux command line. Commands prefixed with `OpenBTS>` are for the OpenBTS command line. It may be convenient to have two terminal windows open so there is no need to constantly enter and exit the OpenBTS command line.

Changing the Band and ARFCN

The first things you must check are the radio band and Absolute Radio Frequency Channel Number (ARFCN) being used. The radio band is one of four values: 850, 900, 1800, or 1900 MHz, corresponding to the four GSM bands available around the world. An ARFCN is simply a pair of frequencies within the selected band that will be used for the transmission and reception of radio signals. Each radio band has over 100 different ARFCNs that can be used. ARFCN may also be referred to as the carrier (e.g., systems using multiple ARFCNs are multiple carrier systems). Choosing the correct band and ARFCN is important for regulatory reasons and to avoid interference with or from local carriers. You use the OpenBTS `config` command to inspect the current band and ARFCN settings.

These configuration keys are in the `GSM.Radio` category. To view all configuration keys with the word `GSM.Radio` in their name, enter the following command:

```
OpenBTS> config GSM.Radio
GSM.Radio.ARFCNs 1     [default]
GSM.Radio.Band 900     [default]
GSM.Radio.C0 51     [default]
GSM.Radio.MaxExpectedDelaySpread 4     [default]
GSM.Radio.PowerManager.MaxAttenDB 10     [default]
GSM.Radio.PowerManager.MinAttenDB 0     [default]
GSM.Radio.RSSITarget -50     [default]
GSM.Radio.SNRTarget 10     [default]
```

The `GSM.Radio.Band` key shows that the 900 MHz band is being used and the `GSM.Radio.C0` key indicates that ARFCN #51 in that band is currently selected.

If your radio hardware does not have limitations on or optimizations for a particular frequency, you can proceed with these settings. An easy optimization for eliminating interference is to choose a band that is not used by other carriers in your country. In general, the Americas use 850 and 1900 MHz systems while the rest of the world uses 900 and 1800 MHz. A country-by-country list can be found on Wikipedia. Also, choose a lower frequency if possible to improve coverage with lower power. If your local regulator has assigned you a specific band and ARFCN, then you must use it.

To change your GSM band, you must again use the OpenBTS `config` command. This time, add the desired band to the end of the command. The following example changes the band to 850 MHz:

```
OpenBTS> config GSM.Radio.Band 850
GSM.Radio.Band changed from "900" to "850"
WARNING: GSM.Radio.C0 (51) falls outside the valid range of ARFCNs 128-251 for
GSM.Radio.Band (850)
GSM.Radio.Band is static; change takes effect on restart
```

The command confirms that the band has been changed but delivers two additional pieces of information. First, there is a warning about the ARFCN not being valid anymore for the 850 MHz band. The valid range is 128–251 for 850 MHz. Second, you are informed that the `GSM.Radio.Band` parameter is static and cannot be applied at runtime; OpenBTS must be restarted.

To fix the first warning, use the `config` command again to set a valid ARFCN for the 850 MHz band:

```
OpenBTS> config GSM.Radio.C0 166
GSM.Radio.C0 changed from "51" to "166"
GSM.Radio.C0 is static; change takes effect on restart
```

The command confirms that the ARFCN has been changed and warns again about this parameter being static. You can now restart OpenBTS to apply the change:

```
$ sudo stop openbts
openbts stop/waiting
$ sudo start openbts
openbts start/running, process 6075
```

The service will take a few seconds to start back up and you are again free to use the OpenBTS CLI.

Range Networks Radio Calibration

The Range Networks SDR1 hardware can ship with filters in place that optimize the radio for a specific GSM band. Each unit is also calibrated to a specific band and may not perform well if the OpenBTS settings do not match this calibration. To view the radio's factory calibration, use the `trxfactory` command:

```
OpenBTS> trxfactory
Factory Information
  SDR Serial Number = XXXX
  RF Serial Number = XXX
  GSM.Radio.Band = 850
  GSM.Radio.RxGain = 52
```

```
    TRX.TxAttenOffset = 1
    TRX.RadioFrequencyOffset = 116
```

To determine if anything needs adjusting, use the `audit` command. The `audit` command is your constant companion when troubleshooting an installation. It reports nondefault values, invalid values, conflicting values, and if you are using Range Networks hardware, there will be a section listing any mismatches between the factory calibration and the current running configuration:

```
OpenBTS> audit
+-------------------------------------------------------------------+
| WARNING : Factory Radio Calibration [key current-value (factory)]   |
|   To use the factory value again, execute: rmconfig key             |
+-------------------------------------------------------------------+
TRX.RadioFrequencyOffset "132" ("116")
```

In this example, the `TRX.RadioFrequencyOffset` parameter does not match the factory calibration. As the message indicates, you can use the factory value again by using the `rmconfig` command:

```
OpenBTS> rmconfig TRX.RadioFrequencyOffset
TRX.RadioFrequencyOffset set back to its default value
TRX.RadioFrequencyOffset is static; change takes effect on restart
```

This is another static parameter that requires a restart of OpenBTS as detailed above. Most configuration parameters are not static, so restarting OpenBTS will be the exception rather than the rule. Repeat this procedure for any other calibration parameters that are reported as mismatches.

Ettus Research Radio Calibration

One main difference between the Range Networks and Ettus Research radios is in the proper value for `GSM.Radio.RxGain`. Range Networks uses a much higher value for this parameter and if it is not adjusted, the Ettus Research equipment will not work correctly. The signal being received will overdrive the demodulator.

For starters, set `GSM.Radio.RxGain` to 10:

```
OpenBTS> devconfig GSM.Radio.RxGain 10
GSM.Radio.RxGain changed from "52" to "10"
GSM.Radio.RxGain is static; change takes effect on restart
```

There is also a dedicated command that allows you to set this parameter without restarting OpenBTS. Use `rxgain` if you need to make fine adjustments to avoid restarting each time.

Searching for the Network

Now that the radio is calibrated and the settings are confirmed, you will use a handset to search for the newly created network. Each handset's menu is different but the item is usually similar to "Carrier Selection" or "Network Selection." The process for manually selecting a different carrier on Android is detailed in Figure 2-1.

1. Launch the "Settings" application from the Android menu system.
2. Select "More."
3. Select "Mobile networks."
4. Select "Network operators." This may or may not start a search. If it does not, select "Search networks."
5. Once the search has finished, a list of available carrier networks is presented.

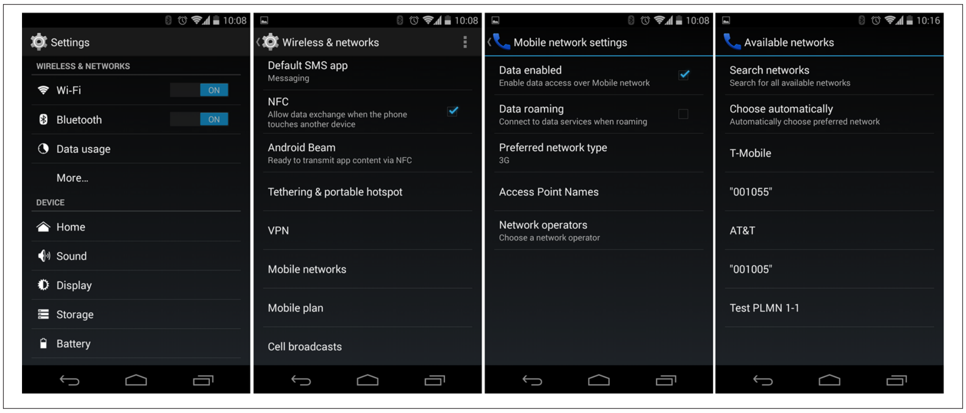
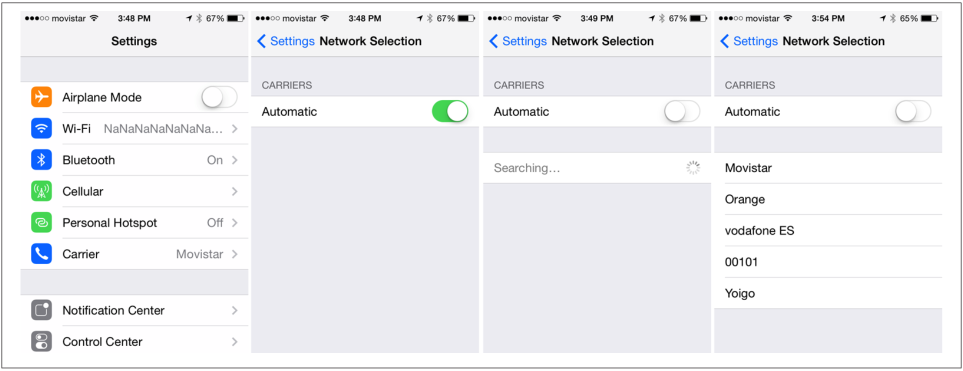


Figure 2-1. Android manual carrier selection

The process for manually selecting a different carrier on iOS 7 is detailed in Figure 2-2.

1. From the home screen, open the "Settings" app.
2. Select "Carrier."
3. On the "Network Selection" screen, disable the automatic carrier selection.
4. The handset will now search for available carrier networks.
5. Once the search has finished, a list of available carrier networks is presented.


●●●○○ movistar 3:48 PM 67%
Settings
Airplane Mode
Wi-Fi NaNaNaNaNaNaNa…
Bluetooth On
Cellular
Personal Hotspot Off
Carrier Movistar
Notification Center
Control Center

●●●○○ movistar 3:48 PM 67%
Settings Network Selection
CARRIERS
Automatic

●●●○○ movistar 3:49 PM 67%
Settings Network Selection
CARRIERS
Automatic
Searching…

●●●○○ movistar 3:54 PM 65%
Settings Network Selection
CARRIERS
Automatic
Movistar
Orange
vodafone ES
00101
Yoigo

Figure 2-2. iOS 7 manual carrier selection

Here we see the test network in the list of selectable carriers. Depending on the handset model, firmware, and SIM used, the network ID will be displayed as "00101," "001-01," "Test PLMN 1-1," or the GSM shortname of "OpenBTS." If your test network is not detected, force the search again by either reselecting the menu item, toggling airplane mode between on and off, or power cycling the handset. If that still does not work, confirm again that the handset supports the GSM band you have configured above and that the baseband is unlocked (i.e., not restricted by contract to only using a specific carrier).

This section had you verify that the downlink is functional; the following section will guide you through verifying the uplink. Then you will move on to entering the identity parameters for your handset and connecting to the network in "First Connection" on page 27.

Testing Radio Frequency Environment Factors

If you've never had a project that involves RF or analog signals in general, you may be surprised by the number of things that can go wrong with them. You may actually be surprised, in the end, that RF communication can work at all! RF experts in the OpenBTS community are sometimes regarded as practitioners of black magic…but I digress.

In a GSM network, two separate radio frequencies are used so the base station and handsets can communicate simultaneously in both directions. Put another way, GSM uses frequency division multiple access to establish full duplex communication. The ARFCN selected (see "Changing the Band and ARFCN" on page 20) is what determines which pair of frequencies will be used. The path from the base station to the handset is known as the downlink and the path from the handset back to the base station is known as the uplink.

In the previous section, you've been able to successfully search for and find the new network beacon. This shows that the network's downlink is reaching your handset and the signal is clean enough to demodulate and interpret the information as evidenced by the network shortname and/or identity numbers being displayed.

The next thing to look out for when setting up a new network is excess radio interference or "noise" from other sources on the uplink. If the uplink is too noisy, the signals from handsets cannot reliably be demodulated into usable information. OpenBTS will show the current level by using the `noise` command:

```
OpenBTS> noise
noise RSSI is -68 dB wrt full scale
MS RSSI target is -50 dB wrt full scale
INFO: the current noise level is acceptable.
```

In this example, the detected environmental noise Received Signal Strength Indication (RSSI) is –68 dB (lower numbers are better and mean less noise is present) and the configured target RSSI level for handsets is –50 dB. This means that the base station can, at best, receive 18 dB more energy from the handsets than the environmental noise —a very good margin meaning uplink reception issues due to noise should not be a problem.

Smaller margins between these two numbers will produce different informational messages. For example, having a margin of 10 dB or less will report:

```
WARNING: the current noise level is approaching the MS RSSI target, uplink
connectivity will be extremely limited.
```

And a margin of zero or less will report:

```
WARNING: the current noise level exceeds the MS RSSI target, uplink connectivity
will be impossible.
```

If either of these WARNING messages are reported, you will need to take action to reduce uplink noise and/or increase the handset transmit power.

In the future, if your handset can see the base station but can no longer connect, noise should be the first thing to check. Your configuration could still be 100% correct and functional but the radio environment may have changed, preventing communication.

Reducing Noise

If your base station radio setup does not include a frequency duplexer, the number one source of noise on the uplink can actually be the downlink signal. Without proper duplexing to filter it out, the downlink signal is usually the closest energy source to the uplink both physically and by frequency. More information about duplexers can be found in "Stronger, Cleaner Signals" on page 46. Even without a duplexer, there are ways to reduce noise on the uplink.

Antenna alignment

A quick duplexer of sorts is simply aligning the antennas so they do not so readily feed into each other. If you are using rubber duck–style antennas, tilt them so they form a 90 degree angle. The radiation pattern for these antennas will then be perpendicular as shown in Figure 2-3.

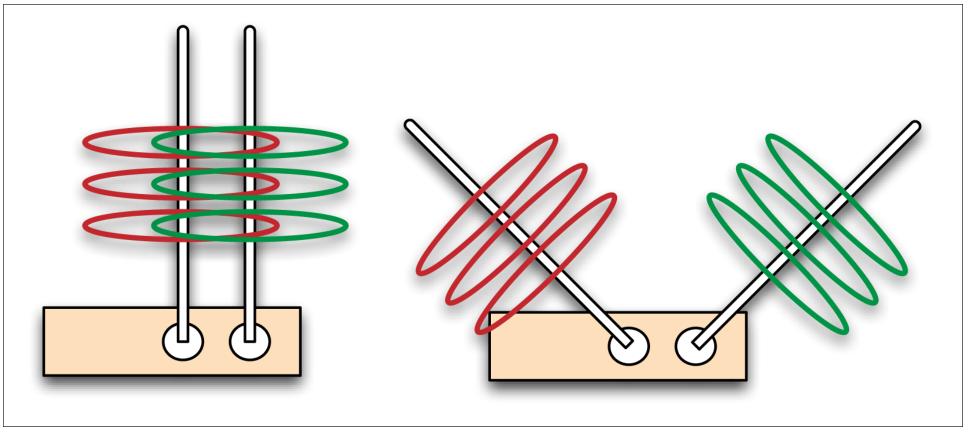


Figure 2-3. Antenna alignment

If the antennas are parallel to each other, signal can efficiently flow from the transmit to the receive antenna, but when the antennas form a 90 degree angle, the signal is being transmitted on a different plane than it is being received on. Observe the change by running the `noise` command before and after your adjustment. This simple adjustment can reduce noise by as much as 10 dB.

Downlink transmission power

The alignment step above reduced the flow of energy from the transmit antenna to the receive antenna. This received energy may still be too high for the uplink to be usable. Decreasing the downlink transmission power will further clean up the uplink. The coverage area lost by decreasing the downlink power is not significant in a lab environment. Cleaner signals are preferable to strong ones. Run the `power` command with no arguments to see the current level. The power is reported in decibels of attenuation:

```
OpenBTS> power
current downlink power 0 dB wrt full scale
```

To decrease the downlink transmission power, for example by 20 dB, enter the following:

```
OpenBTS> power 20
current downlink power -20 dB wrt full scale
```

The downlink is now transmitting with 20 dB less power. Use the `noise` command to observe the improvement.

Boosting Handset Power

Handsets can also be told to use more power by adjusting the `GSM.Radio.RSSITarget` and `GSM.Radio.SNRTarget` keys. The default values for these keys should be sufficient in most situations. However, if you encounter large fluctuations in received power (e.g., walking around corners in an underground mine), it may be necessary to increase these values to provide a larger buffer in allowable power differences. Boosting the power for all handsets will drain their batteries more rapidly but uplink signals will be more reliable. There are always trade-offs.

If your noise level is still too high, go back to "Starting Up the Network" on page 19 to change your ARFCN to a less noisy one.

First Connection

You have now verified both the downlink and uplink. To make sure all settings have been applied, issue the following command to restart OpenBTS:

```
$ sudo stop openbts
$ sudo start openbts
```

The only step left before actually connecting to your test network is to find and enter your handset's identity parameters so it will be accepted onto the network.

Finding the IMSI

The main identity parameter you will be searching for is the International Mobile Subscriber Identity (IMSI). This is a 14–15 digit number stored in the SIM card and is analogous to the handset's username on the network.

Handsets will not usually divulge the IMSI of their SIM card. It can sometimes be located in a menu or through a field test mode, but this method of determining a SIM's IMSI is very cumbersome to explain. Luckily, there are other methods; OpenBTS also knows the IMSIs it has interacted with and, because you are in control of the network side, you also have access to this information.

To force an interaction between a handset and your test network, you will perform a location update request (LUR) operation on the network, analogous to a registration. This is nothing more complicated than selecting the network from the carrier selection list.

Before attempting any LURs, you need to start the SIPAuthServe daemon responsible for processing these requests:

```
$ sudo start sipauthserve
sipauthserve start/running, process 7017
```

Now, again following the steps in "Searching for the Network" on page 23, bring up the carrier selection list and choose your test network. After a short time, the handset should report a registration failure.

It may also receive an SMS from your test network indicating that registration has failed. This message automatically includes the IMSI, thus you can skip to "Adding a Subscriber" on page 29. However, this feature does not work on all hardware so continue on if you did not receive an SMS.

OpenBTS remembers these LUR interactions in order to perform something called IMSI/Temporary Mobile Subscriber Identity (TMSI) exchanges. IMSI/TMSI exchanges swap the user-identifiable IMSI for a TMSI and are used to increase user privacy on the network. The exchanges are disabled by default (modify `Control.LUR.SendTMSIs` to enable); however, the information is still there to inspect using the `tmsis` command. Use it now to view all recent LUR interactions with handsets:

```
OpenBTS> tmsis
IMSI            TMSI IMEI            AUTH CREATED ACCESSED TMSI_ASSIGNED
214057715229963 -    012546629231850 0    78s     78s      0
001010000000002 -    351771054186520 1    80h     95s      0
001010000000003 -    351771053005400 1    80h     108s     0
```

Entries are sorted by time, with the top entries corresponding to the most recent interactions. Your handset should be the top entry on this list—the most recent interaction with AUTH set to 0 because the LUR failed due to the handset not being a known subscriber. The other entries in this example are additional test handsets that have successfully performed an LUR as indicated by the AUTH column being set to 1.

Finding the IMEI

In a busy environment, it can be difficult to ascertain which handset hardware corresponds to which entry on this list. To match an IMSI to a specific piece of hardware, you can use the International Mobile Equipment Identifier (IMEI). It is the unique identifier given to the handset's physical radio hardware, analogous to a MAC address on an Ethernet interface.

A handset's IMSI is usually printed under its battery cover or somewhere very near the SIM itself. On many handsets, the IMEI can also be accessed by dialing the following on the keypad:

```
*#06#
```

The IMEI value is typically only used for reporting and detecting stolen hardware in production environments. Here it serves as a convenient way to determine which SIM is in which handset. The final digit of your IMEI may not match what OpenBTS displays. It is a check digit and is shown as a zero in OpenBTS.

Adding a Subscriber

You should now have all the necessary pieces of information to create a new subscriber account on your test network.

A couple of fields are still needed but are freely selectable: Name and Mobile Station International Subscriber Directory Number (MSISDN). The Name field is merely a friendly name for this subscriber so you can remember which handset or which person it is associated with. The MSISDN field is nothing more complicated than the subscriber's phone number. Because you are not connected to the public telephone network, this can be any number you choose.

The program you need to add subscribers is `nmcli.py`. It is a simple client for the NodeManager APIs (more on those in Chapter 7) and allows you to change configuration parameters, add subscribers, monitor activity, etc., all via JSON formatted commands.

`nmcli.py` is already present in your development directory—move there now to access it:

```
$ cd dev/NodeManager
```

There are two ways to add a subscriber using `nmcli.py`. The first creates a subscriber that will use cached authentication:

```
$ ./nmcli.py sipauthserve subscribers create name imsi msisdn
```

The second creates a subscriber that will use full authentication:

```
$ ./nmcli.py sipauthserve subscribers create name imsi msisdn ki
```

If the IMSI is the "username," the "password" for a handset is the Ki field discussed in "Using spare SIMs" on page 5. If you are using a spare SIM from another provider, you do not have access to Ki and should use the first invocation style. If you are using a SIM that you have burned yourself, Ki will be known to you. Use the second invocation style of `nmcli.py` to include it for your new subscriber.

In this example, a spare SIM is being used in an iPhone 4 and it will be assigned a fake number in South Dakota (area code 605):

The IMSI field consists of the numeric IMSI prefixed with the literal string "IMSI."

```
$ ./nmcli.py sipauthserve subscribers create "iPhone 4" IMSI214057715229963 \
6055551234
raw request: {"command":"subscribers","action":"create","fields":
{"name":"iPhone 4","imsi":"IMSI214057715229963","msisdn":"6055551234","ki":""}}
raw response: {
        "code" : 200,
        "data" : "both ok"
}
```

Perform the same command substituting your own information to add the first subscriber to your test network.

Connecting

Now when you select your test network in the connection menu, the LUR should succeed. This can be confirmed with the `tmsis` command in OpenBTS. The "AUTH" column will now have a "1" in the entry corresponding to your IMSI:

```
OpenBTS> tmsis
IMSI            TMSI IMEI            AUTH CREATED ACCESSED TMSI_ASSIGNED
214057715229963 -    012546629231850 1    11m     56s      0
001010000000002 -    351771054186520 1    80h     8m       0
001010000000003 -    351771053005400 1    80h     9m       0
```

Congratulations, you've successfully registered to your own private mobile network! Feel free to register any additional handsets you wish to use before proceeding.

Test SMS

Now that a handset has access to your network, you can perform some more interesting tests. The first is a quick test of your network's SMS capabilities.

The component responsible for receiving, routing, and scheduling the delivery of SMS messages is SMQueue. It must be started before testing out these features; execute the following command to do so:

```
$ sudo start smqueue
smqueue start/running, process 21101
```

Echo SMS (411)

On your handset, compose an SMS to the number 411. This is a "shortcode" handler in SMQueue that will simply echo back whatever it receives along with some additional information about the network and subscriber account that was used. The body of the message to 411 can be whatever you'd like, although it can be useful to use unique content for each message or sequential numbers or letters. This helps you pinpoint which message is being responded to in case an error occurs.

Once you have your message composed to 411, hit send. After a few seconds, a reply should appear (an example follows):

"1 queued, cell 0.1, IMSI214057715229963, phonenum 6055551234, at Sep 8 02:30:59, *Ping pong*"

This indicates the following:

- There is one message queued for delivery.
- The base station has a load factor of 0.1.
- The message was received from IMSI 214057715229963, MSISDN 6055551234.
- The message was sent on September 8 at 02:30:59.
- The message body was "Ping pong."

Direct SMS

SMS messages can also be tested directly from OpenBTS by using the `sendsms` command. From the OpenBTS CLI, let's see how it is invoked by using the `help` command:

```
OpenBTS> help sendsms
sendsms IMSI src# message... -- send direct SMS to IMSI on this BTS, addressed
from source number src#.
```

Messages are sent by specifying a target IMSI, the source number the message should appear to have originated from, and the message body itself. Substitute the information for your subscriber account to compose a message and press Enter:

```
OpenBTS> sendsms 214057715229963 8675309 direct SMS test
message submitted for delivery
```

After a few seconds, your handset should display a new incoming message from the imaginary number 8675309 with a body of "direct SMS test."

SMS messages created in this way do not route through SMQueue at all; they are sent directly out through the GSM air interface to the handset and, as such, cannot be rescheduled. If the handset is offline or unreachable, these messages are simply lost. This is why SMQueue is needed—to attempt and reschedule deliveries in the inherently unpredictable wireless environment.

Two-Party SMS

If you have configured more than one handset for use in your network, feel free to send a few messages back and forth between them. Verify that the source numbers are correct when receiving messages and that replies to these messages are routed back to the original sender.

Test Calls

The other service to test is voice. As with SMS, OpenBTS does not directly handle voice and requires an additional service to be run—in this case, Asterisk.

Start Asterisk now:

```
$ sudo start asterisk
asterisk start/running, process 1809
```

Using the same handset you used in the SMS tests, you will now verify a few aspects of the voice service. This is accomplished by utilizing a few test extensions that the range-asterisk-configs package defines. An extension is an internal phone number, unreachable from the outside.

Test Tone Call (2602)

The first test extension you will use plays back a constant tone. This might not sound too exciting but does confirm many things about the network:

- Asterisk is running and reachable.
- Call routing is working as expected.
- Downlink audio is functional.

Call 2602 with your handset now.

As you listen to the tone, listen for changes in pitch. These changes in pitch are due to missing information in the downlink voice stream path, similar to packet loss. In the field, this is the primary use for the test tone extension: testing downlink quality. A downlink loss of 3% is normal in production networks, with losses of 5%–7% still providing an understandable conversation.

Echo Call (2600)

The next test extension creates an "echo call." Basically, all audio that Asterisk receives will be immediately echoed back to the sender. In addition to confirming the items listed for the test tone call, the echo call will reveal any delay or uplink quality issues present in your network.

Call 2600 with your handset now.

As you speak into the microphone, you should hear yourself very shortly afterward in the earpiece. A little delay is normal, but longer delays lead to an experience more like using a walkie-talkie. The human brain can deal with delays up to about 200 ms without trouble. Beyond that, the conversation starts to break down and both sides stop speaking because it becomes uncomfortable.

Two-Party Call

If you have configured more than one handset for use in your network, feel free to place some calls between them. Verify that the source numbers are correct when receiving a call.

Measuring Link Quality

At this point, you should learn about a handy tool available on the OpenBTS CLI, the `chans` command. This tool can be used to objectively quantify link quality instead of basing it on user perception. To see anything useful with this command, there must be an active call.

In this example, a call was placed to the 2602 test tone extension and after 10 seconds the `chans` command was executed (note the "Time" column). The handset was moved a meter farther away from the radio being used and another sample with the `chans` command was taken:

```
OpenBTS> chans
CN TN chan   transaction Signal SNR  FER   TA  TXPWR RXLEV_DL BER_DL Time IMSI
      type   id          dB          pct   sym dBm   dBm      pct
0  1  TCH/F  T101        28     28.7 0.15  1.2 7     -61      0.00   0:10 2140...
OpenBTS> chans
CN TN chan   transaction Signal SNR  FER   TA  TXPWR RXLEV_DL BER_DL Time IMSI
      type   id          dB          pct   sym dBm   dBm      pct
0  1  TCH/F  T101        14     31.4 0.50  1.2 19    -73      0.00   0:26 2140...
```

There are a lot of fields here and they're all very useful but for now let's focus on signal-to-noise ratio (SNR), TXPWR, and RXLEV_DL.

In both readings, there is a single active channel. The SNR column represents the uplink's SNR as measured by the base station: higher is better. As the handset is moved away, this number actually improves! How can that be? The answer lies in the TXPWR column. This column represents the uplink transmit power that the handset reported. In the second reading, this number has jumped from 7 to 19 dBm, meaning the handset used more power to transmit its signal to the base station. This would explain why there was a better SNR measured at the base station.

The network independently instructs the handsets to transmit with different power levels depending on how well the base station receives their uplink signal. This is so all signals are received at the base station with about the same strength, making it easier to demodulate.

Base stations, however, use the same transmission power on the downlink for all handsets. This can be observed in the RXLEV_DL column. This column represents the downlink signal level that the handset reported. In the second reading, this number has gone down from –61 dBm to –73 dBm as the handset moves farther away from the base station. It is receiving the downlink signal with less strength because it is now farther away.

A complete listing of these fields can be retrieved by running:

```
OpenBTS> help chans
```

Configuration System, Continued

In this chapter you've become familiar with several OpenBTS commands, the most important one likely being `config`. Before moving on, some final details on `config` will be provided and you will be introduced to a few of its relatives.

config

You have so far used `config` to search for configuration keys and change their values. It can also be used to provide additional information about a specific configuration key if the complete name is provided:

```
OpenBTS> config SIP.Proxy.SMS
SIP.Proxy.SMS 127.0.0.1:5063 (default)

 - description:      The hostname or IP address and port of the proxy
to be used for text messaging. This is smqueue, for example.
 - type:             hostname or IP address and port
 - default value:    127.0.0.1:5063
 - visibility level: customer warn - a warning will be presented and
confirmation required before changing this sensitive setting
 - static:           0
 - scope:            value must be the same across all nodes
```

devconfig

The `devconfig` command functions identically to the `config` command. However, it allows you to manipulate more types of keys. Each key has a visibility level that signals whom the key was intended for: user, developer, or factory. Using `devconfig` gives you access to these more sensitive keys, such as protocol timers:

```
OpenBTS> config GSM.Timer
GSM.Timer.Handover.Holdoff 10     [default]
GSM.Timer.T3109 30000     [default]
GSM.Timer.T3212 0     [default]

OpenBTS> devconfig GSM.Timer
GSM.Timer.Handover.Holdoff 10     [default]
GSM.Timer.T3103 12000     [default]
GSM.Timer.T3105 50     [default]
GSM.Timer.T3109 30000     [default]
GSM.Timer.T3113 10000     [default]
GSM.Timer.T3212 0     [default]
```

rawconfig

The `rawconfig` command takes things a bit further than `devconfig` and removes all input validation. If you have a key that you would like changed to an experimental value outside of the valid range, you need to use `rawconfig`.

Another use of `rawconfig` is to define completely new custom key/value pairs in the database. This is convenient when authoring a new feature:

```
OpenBTS> rawconfig My.New.Setting zebra
defined new config My.New.Setting as "zebra"
```

unconfig

Some keys that control optional features can be disabled. Keys that can be disabled will have "string (optional)" set in their type field:

```
OpenBTS> config Control.LUR.FailedRegistration.Message
Control.LUR.FailedRegistration.Message Your handset is not provisioned
for this network.      [default]

 - description:      Send this text message, followed by the IMSI, to
unprovisioned handsets that are denied registration.
 - type:             string (optional)
```

The `unconfig` command will attempt to disable a key and report back if it was successful:

```
OpenBTS> unconfig Control.LUR.FailedRegistration.Message
Control.LUR.FailedRegistration.Message disabled
```

rmconfig

To set a key back to its default value, use `rmconfig`:

```
OpenBTS> rmconfig SIP.Proxy.SMS
SIP.Proxy.SMS set back to its default value
```

This command can also be used to remove custom key/value pairs that were defined with `rawconfig`:

```
OpenBTS> rmconfig My.New.Setting
My.New.Setting removed from the configuration table
```

Personalizing Your Network

Your network is up and running and you know how to control it, not a bad start. This is, however, your network, so it should be customized to your tastes. This section will walk through the process of personalizing a few things so that it's definitely your network.

Shortname

The shortname is displayed on some handsets when browsing. It's the first thing someone will notice when searching for the network, so go ahead and change it from the default of OpenBTS:

```
OpenBTS> config GSM.Identity.ShortName GroundControl
GSM.Identity.ShortName changed from "OpenBTS" to "GroundControl"
```

Spaces and special characters are not allowed in the name—only alphanumeric characters.

Registration Messages

If you remember, in "First Connection" on page 27 there was a "registration failed" SMS sent to the handset. Actually, there are several SMS messages like this for different events. To see a list of them, search for configuration keys with `Registration.Message` in their name:

```
OpenBTS> config Registration.Message
Control.LUR.FailedRegistration.Message Your handset is not provisioned for
     this network.      [default]
Control.LUR.NormalRegistration.Message (disabled)     [default]
Control.LUR.OpenRegistration.Message Welcome to the test network.  Your IMSI
     is      [default]
```

The registration failed message is nice but kind of boring. You can change it with the `config` command:

```
OpenBTS> config Control.LUR.FailedRegistration.Message Nuh-uh-uh, you didn't say
the magic word.
```

```
Control.LUR.FailedRegistration.Message changed from "Your handset is not
provisioned for this network." to "Nuh-uh-uh, you didn't say the magic word."
```

Or, to completely disable the failed registration message, you can use `unconfig`:

```
OpenBTS> unconfig Control.LUR.FailedRegistration.Message
Control.LUR.FailedRegistration.Message disabled
```

There is another message, `Control.LUR.NormalRegistration.Message`, which is disabled by default. This message is sent to the handset every time it successfully registers. While annoying in a production environment, it can be a useful tool for labs, especially if they are operating multiple base stations. The message serves as a heads-up if the registration updates or switches:

```
OpenBTS> config Control.LUR.NormalRegistration.Message Welcome to BTS 1
Control.LUR.NormalRegistration.Message changed from "" to "Welcome to BTS 1"
```

To test it out, try power cycling your handset or toggling airplane mode between on and off. When the handset reacquires your base station signal and registers, you should receive a message.

The `Control.LUR.OpenRegistration.Message` parameter will be covered in Chapter 6.

CHAPTER 3

Troubleshooting and Performance Tuning

As your network becomes ever more production-ready, you will need some additional techniques to help debug problems as they arise. While some errors are directly related to misconfiguration, others are less obvious. Depending on the deployment conditions and network usage patterns, problems with the network may actually be related to poor performance. Optimizing the network's performance for your particular scenario can be key. Because of this, troubleshooting and tuning are presented together.

The stats Command

A quick and easy way to observe when events occur in OpenBTS is by using the `stats` command. There are several dozen event types that are tracked. To get a full list of them, run the `stats` command with no arguments:

```
OpenBTS> stats
```

Each event type is simply a key name in a small SQLite3 database and the value for each key corresponds to the number of times this event has happened since the last time the stats database was cleared. Clearing the database is handy to give yourself a known starting count for the events you are interested in. To clear the database, execute the following:

```
OpenBTS> stats clear
stats table (gReporting) cleared
```

Now, if you send an SMS from one handset to the other on your network and search for SMS-related events, you will see something like the following:

```
OpenBTS> stats SMS
OpenBTS.GSM.MM.CMServiceRequest.MOSMS: 1 events over 4 minutes
OpenBTS.GSM.SMS.MOSMS.Start: 1 events over 4 minutes
OpenBTS.GSM.SMS.MOSMS.Complete: 1 events over 4 minutes
OpenBTS.GSM.SMS.MTSMS.Start: 1 events over 4 minutes
OpenBTS.GSM.SMS.MTSMS.Complete: 1 events over 4 minutes
```

`OpenBTS.GSM.MM.CMServiceRequest.MOSMS` shows that a handset signaled to the base station that it wished to perform a mobile originated SMS (MOSMS). The `OpenBTS.GSM.SMS.MOSMS.Start` and `Complete` keys show that an MOSMS was both started and completed. These first three keys are related to the initial transmission from source handset to base station. The last two keys show that the mobile terminated SMS (MTSMS) from the base station to the destination handset was both started and completed.

Runtime Logs

The logs for all OpenBTS components are stored in */var/log/OpenBTS.log*. To monitor them live as they are produced, execute:

```
$ tail -f /var/log/OpenBTS.log
```

If you'd like to only monitor log entries that contain a certain bit of text, you can chain the `tail` and `grep` commands. For example, to only monitor new entries that contain the text "sipauthserve", use the following:

```
$ tail -f /var/log/OpenBTS.log | grep sipauthserve
```

You can also search through existing logs for a specific text. In this example, we're not monitoring for new entries, but rather searching the entire logfile for the text "sipauthserve":

```
$ grep sipauthserve /var/log/OpenBTS.log
```

Log entries contain many different pieces of information. Here is an example entry that OpenBTS produced:

```
2014-09-04T10:50:09.258961+02:00 ubuntu openbts: NOTICE 27238:27238 2014-09-04T
10:50:09.2 GSMConfig.cpp:132:regenerateBeacon: regenerating system information
messages, changemark 5
```

The fields from left to right are:

- Data write timestamp: "2014-09-04T10:50:09.258961+02:00"
- System hostname: "ubuntu"
- System application: "openbts"
- Event level: "NOTICE"
- Event user and group: "27238:27238"
- Data create timestamp: "2014-09-04T10:50:09.2"
- Event source file, line, and function name: "GSMConfig.cpp:132:regenerateBeacon"

- Event text: “regenerating system information messages, changemark 5”

Log Levels

By default, the components are only set to log events at the NOTICE level and above. This is usually enough information to debug typical service errors such as dropped calls or high interference rates. The logging system that all OpenBTS components use has eight levels for reporting events:

EMERG
: Report serious faults associated with service failure or hardware damage

ALERT
: Report likely service disruption caused by misconfiguration or poor connectivity

CRIT
: Report anomalous events that are likely to degrade service

ERR
: Report internal errors of the software that may result in degradation of service in unusual circumstances

WARNING
: Report anomalous events that may indicate a degradation of normal service

NOTICE
: Report anomalous events that probably do not affect service but may be of interest to network operators

INFO
: Report normal events

DEBUG
: Will degrade system performance; only for use by developers

To get different information out of the system, you adjust the logging level. The INFO level, for example, also reports normal events (in addition to NOTICE and above). These normal events can provide context for you to deduce what error may have occurred. To change the logging level component-wide, execute the following:

```
OpenBTS> config Log.Level INFO
Log.Level changed from "NOTICE" to "INFO"
```

Once you’ve localized some suspicious activity from the logs, you can use the source file field to expose more information from that part of the application. The event example above was emitted from *GSMConfig.cpp*. To enable DEBUG level logging on just that source file, execute the following:

```
OpenBTS> rawconfig Log.Level.GSMConfig.cpp DEBUG
defined new config Log.Level.GSMConfig.cpp as "DEBUG"
```

You need to use `rawconfig` to define this key/value pair because it is not in the configuration schema. To remove this custom key/value pair from the configuration database, use `rmconfig`:

```
OpenBTS> rmconfig Log.Level.GSMConfig.cpp
Log.Level.GSMConfig.cpp removed from the configuration table
```

Starting in OpenBTS 5.0, it is now possible to define a log level for the individual subsystem groups in OpenBTS. Each of the following groups can be individually adjusted:

- `Log.Level.Control`
- `Log.Level.SIP`
- `Log.Level.GSM`
- `Log.Level.GPRS`
- `Log.Level.Layer2`
- `Log.Level.SMS`

Again, because this is a custom key/value pair, you must use `rawconfig`:

```
OpenBTS> rawconfig Log.Level.SIP DEBUG
defined new config Log.Level.SIP as "DEBUG"
```

And again, to remove a custom key/value pair from the configuration database, use `rmconfig`:

```
OpenBTS> rmconfig Log.Level.SIP
Log.Level.SIP removed from the configuration table
```

Using the DEBUG level component-wide produces so much information that it can destabilize OpenBTS. DEBUG should only be used on an individual source file or log group.

Environmental Tuning

Adapting OpenBTS to perform well in the environment where it is deployed is a critical step. The surrounding buildings and foliage, climate, tower height, antenna selection, cable length, and amplifier power all play a role. Entire businesses are built on providing expertise in this area and the hardware topic is much too broad for it to be covered in a meaningful way here. Any robust deployment will need both hardware and software tuning. This section sticks to the controls available in the OpenBTS software.

Nonsubscriber Phones

If your deployment is using an amplified RF chain as described in "Stronger, Cleaner Signals" on page 46 and other carrier signals in the area are weak or nonexistent, you will have to deal with nonsubscriber handsets. All handsets that can see your network will attempt to join because they have no local service.

In rural community deployments, this is a very common problem. The majority of the residents in these communities have handsets for when they travel to nearby towns with service. When they arrive back, the handset may not be powered down. If a new tower goes up in these situations, it may have to deal with thousands of nonsubscriber handsets.

These handsets generate LURs that will be rejected. There are over 20 different ways to reject these requests; the key is to use the right one for your deployment. By default, OpenBTS uses a very friendly reject cause (0×04), allowing the handset to retry within a few minutes. You need to use a reject cause that tells the handset to go away for a long period of time so LUR traffic does not overwhelm your system to the point that subscribers cannot use it.

There are two parameters where you can define the reject cause you'd like to use:

```
OpenBTS> config RejectCause
Control.LUR.404RejectCause 0x04     [default]
Control.LUR.UnprovisionedRejectCause 0x04     [default]
```

The key `Control.LUR.404RejectCause` defines which cause to use when an unknown subscriber attempts to join, whereas `Control.LUR.UnprovisionedRejectCause` defines which to use when a known subscriber attempts to join but fails to authenticate. There may be reasons to adjust these independently but for now they will be set identically. The allowed reject codes are described in Table 3-1.

Table 3-1. Reject causes, as defined in GSM 04.08 section 10.5.3.6

Hex value	Description
0x02	IMSI unknown in HLR
0x04	IMSI unknown in VLR
0x05	IMEI not accepted
0x0B	PLMN not allowed
0x0C	**Location area not allowed**
0x0D	**Roaming not allowed in this location area**
0x11	Network failure
0x16	Congestion
0x20	Service option not supported
0x21	Requested service option not subscribed

Hex value	Description
0x22	Service option temporarily out of order
0x26	Call cannot be identified
0x30	Retry upon entry into a new cell
0x5F	Semantically incorrect message
0x60	Invalid mandatory information
0x61	Message type nonexistent or not implemented
0x62	Message type not compatible with the protocol state
0x63	Information element nonexistent or not implemented
0x64	Conditional IE error
0x65	Message not compatible with the protocol state
0x6F	Unspecified protocol error

Each of these may have a different effect depending on the handset. Through testing, a few favorites have emerged and are highlighted in Table 3-1. These are values that will instruct nonsubscriber handsets to go away for a long time, but at the same time, not cause that handset to give up on joining other networks.

Change these values now to cut down on nonsubscriber LUR traffic:

```
OpenBTS> config Control.LUR.404RejectCause 0x0C
Control.LUR.404RejectCause changed from "0x04" to "0x0C"
OpenBTS> config Control.LUR.UnprovisionedRejectCause 0x0C
Control.LUR.UnprovisionedRejectCause changed from "0x04" to "0x0C"
```

Coverage Area

Another way to make handsets ignore your network is to actually prevent them from seeing it. This is done by shrinking the usable coverage area both physically and by policy.

The coverage area for SMS and registration services can be approximately 4× larger (2× the radius) than that of voice services because lost frames can be retransmitted without breaking the service.

Shrinking physically

OpenBTS has a power control mechanism that adjusts attenuation of the transmit power to expand or contract the coverage area. To check the base station's current attenuation level, use the `power` command:

```
OpenBTS> power
current downlink power -10 dB wrt full scale
```

To adjust the attenuation, simply provide an argument to the power command. To increase the base station to maximum power with the largest coverage area, specify that there should be 0 dB of attenuation:

```
OpenBTS> power 0
current downlink power 0 dB wrt full scale
```

Conversely, increase the attenuation to decrease power and shrink the coverage area:

```
OpenBTS> power 20
current downlink power -20 dB wrt full scale
```

There is another physical power control available. Handsets also transmit to the base station using different power levels so the received power from all handsets is approximately equal in strength when it reaches the base station. The base station controls the power levels used by and available to the handsets so we can limit this range. The handset has no control over what power level it will use so we can limit the range of values it is told to use. The `MS.Power` keys allow this:

```
OpenBTS> config MS.Power
GSM.MS.Power.Damping 75     [default]
GSM.MS.Power.Max 33     [default]
GSM.MS.Power.Min 5     [default]
```

Although this method is less commonly used, it is included here for completeness.

Shrinking by policy

Physically shrinking the coverage area may not be enough to stabilize your network under a heavy usage load. There is still one further parameter that can come in handy, `GSM.MS.TA.MAX`:

```
OpenBTS> config GSM.MS.TA
GSM.MS.TA.Damping 50     [default]
GSM.MS.TA.Max 62     [default]
```

TA stands for Timing Advance. Timing Advance is a method that GSM uses to compensate for handsets that are very far away from the base station. The farther away the handset is, the larger the TA value will be. This value tells the handset to transmit its radio bursts earlier so they arrive at the base station squarely within their allotted time slot. The speed of light is fast but still not instantaneous.

Back to the parameter: setting `GSM.MS.TA.MAX` to 10, for example, causes OpenBTS to silently ignore any radio bursts with a TA greater than 10. TA is measured in something called a symbol period and corresponds to a distance of about 550 m. With `GSM.MS.TA.MAX` set to 10, OpenBTS will ignore by policy any bursts from handsets located over 5.5 km away.

Because coverage areas are never perfect circles, this is where a limitation by policy is convenient. If your tower is in a valley and is guarded by elevation in three directions, the remaining direction will receive a signal from the tower much farther out. You don't want to turn down power because there are users on the hillsides, so instead, you can limit the coverage area by policy.

Signal Distortion

Signal distortion is highly dependent on the terrain surrounding your installation. By default, OpenBTS is configured to be very thorough in mitigating multipath distortions. It does this by looking at a larger window of time when attempting to cancel out distortions. This is very computationally expensive.

If your installation is in open terrain without any buildings or trees, you may be able to reduce the CPU load considerably by adjusting `GSM.Radio.MaxExpectedDelaySpread`. Smaller coverage areas are also candidates for using a small value here:

```
OpenBTS> config Delay
GSM.Radio.MaxExpectedDelaySpread 4     [default]
```

This key dictates how many symbol periods will be examined. Adjust back to the default of 4 if poor performance occurs after adjusting down.

Stronger, Cleaner Signals

To effectively extend the coverage of the network for production use you need an amplifier and cavity duplexer. Because GSM handsets usually have a transmit power capability of 2 W and your SDR has a transmit power of 100 mW, the base station will be the factor limiting your coverage area. The uplink signal can reach from handset to base station, but the downlink signal back from base station to handset is too weak for the handset to receive. The relative power asymmetry between the base station and handset is illustrated in Figure 3-1.

To illustrate, adding an amplifier of 2 W into the transmit chain on the base station will increase your coverage area, but now an extra precaution must be taken regarding the receive chain. The receive antenna is now receiving such an increased amount of energy from the transmit antenna that it can actually damage the circuitry of the SDR. At the very least, it will make demodulating a clean signal impossible because the additional 2 W of local transmit energy from the amplifier counts as noise and will completely drown out any remote handsets. To solve this, you need a cavity duplexer.

A cavity duplexer attaches to both the transmit and receive antenna ports. A single antenna is then attached to the duplexer as shown in Figure 3-2. GSM transmits and receives on different frequencies, so the duplexer is able to cleanly split transmit and receive signals.

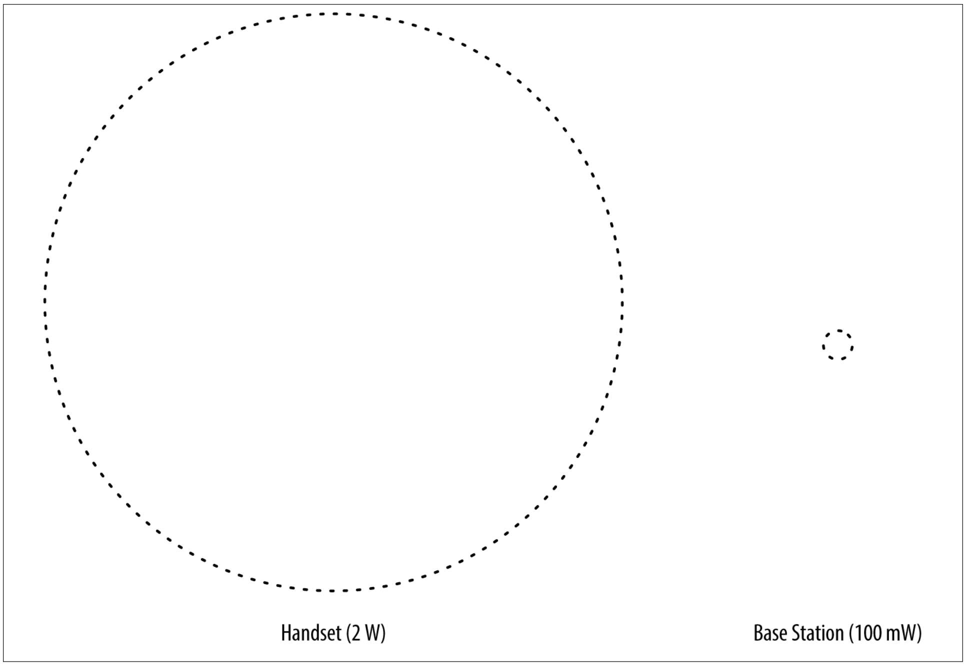


Figure 3-1. Power asymmetry

This prevents unwanted transmit energy from circling back into the receive chain. Because this split is based on frequency, a duplexer must be selected that matches the GSM band you will be using (850, 900, 1800, or 1900 MHz).

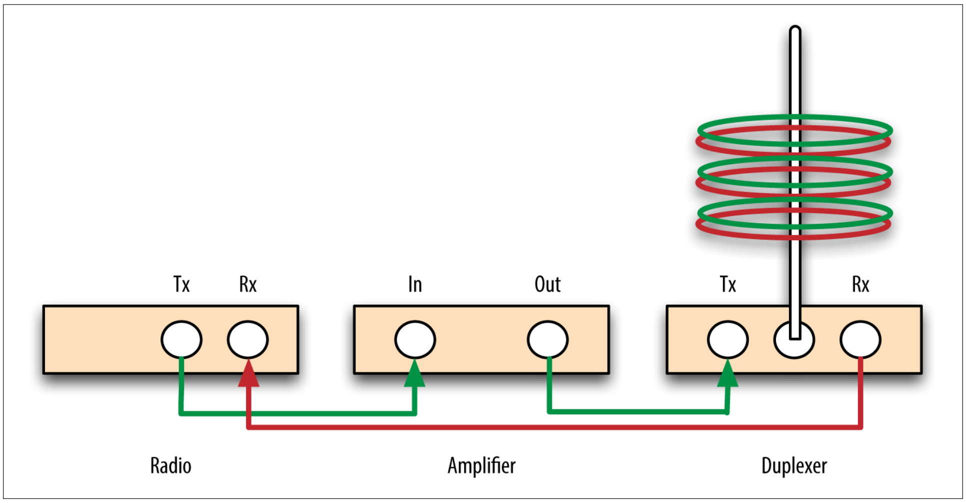


Figure 3-2. Duplexer hookup

CHAPTER 4

From Single to Multinode

A mobile network with a single tower is infinitely more useful than one with zero towers. However, there comes a point where no matter how well that single tower is tuned, it cannot effectively cover a given area.

Because OpenBTS uses the same software stack to implement large and small coverage areas and has even been made to run on a Raspberry Pi, try to free your mind of the classic "cell tower" image. A tower could easily fit in a shoebox or blend in with your WiFi access point.

This chapter describes how to expand your network to multiple physical sites but still maintain a single logical network. This logical network will support Mobility and Handover as any commercial network would.

Mobility, Handover, and Roaming

There is some confusion among these terms. While they do mean very specific things, even experts in the field will throw them around in discussion and expect the other party to understand the correct meaning from the context. To clear up this ambiguity each term will be outlined briefly.

Mobility

Mobility is the ability of a handset to receive service on different physical base stations in an operator's network. As the handset moves in three-dimensional space, the signal quality it receives from neighboring base stations will fluctuate. When the handset detects a significantly better signal from a neighboring base station, it sends an LUR to register or "camp" to the new base station. To differentiate, a periodic LUR at the same base station is refreshing an existing registration but an LUR at a new base station is recamping, switching the registration to a new tower.

An LUR is not performed unless the two base stations have different location area codes (LACs). In a traditional GSM network, all base stations in a given geographic region have the same LAC. However, OpenBTS currently requires all base stations to have a unique LAC so LURs will be performed when moving in between base stations. This triggers an updated SIP REGISTER message and, thus, an updated subscriber registry entry containing the IP of the new base station.

Mobility is a function of the network but the handset makes the decision to recamp. Also, it is also only possible when the handset is not in an active transaction such as a voice call or exchanging SMS. Mobility may also be referred to as "idle mode recamping."

Handover

Handover is the ability of an active voice call to survive the transition between two base stations. Mobility is a prerequisite for Handover, but unlike Mobility, the network determines and executes Handover. The handset has no choice but to obey Handover commands that the network has sent it.

The Base Station Controller (BSC) controls Handover in a traditional GSM network. OpenBTS eliminates the need for a BSC through the use of a new peer-to-peer protocol. Information about neighboring frequencies, identities, and active transactions is exchanged over this protocol, simplifying the deployment architecture.

Several factors determine the decision to execute a Handover. First, the downlink signal from the base station currently serving the handset must be sufficiently weak. Also, the signal from the strongest neighboring base station must also exceed the serving base station level by a set threshold. Finally, the strongest neighboring base station must not have recently rejected any Handovers due to congestion. If these conditions are met, the serving base station will initiate a Handover to the strongest neighboring base station.

Roaming

Roaming is Mobility across carriers. It requires an administrative agreement between those carriers and common interface technology, usually GSM-Mobile Application Part (MAP) signaling across the Signaling System No. 7 (SS7) network. It is impossible to roam within your own network.

Topology

Up until now, you have been building a "network-in-a-box" topology with every single component located on a single server, a single logical entity. In a multinode network there are now two logical entities: Central Services and the "tower." The components you are familiar with so far must be rearranged a bit to support this. Central Services will need SIPAuthServe, SMQueue, and Asterisk installed. For the sake of simplicity, SIPAuthServe, SMQueue, and Asterisk will be referred to as Central Services for the remainder of this chapter. The tower will only run OpenBTS.

There will be a single Central Services install and multiple tower installs as shown in Figure 4-1.

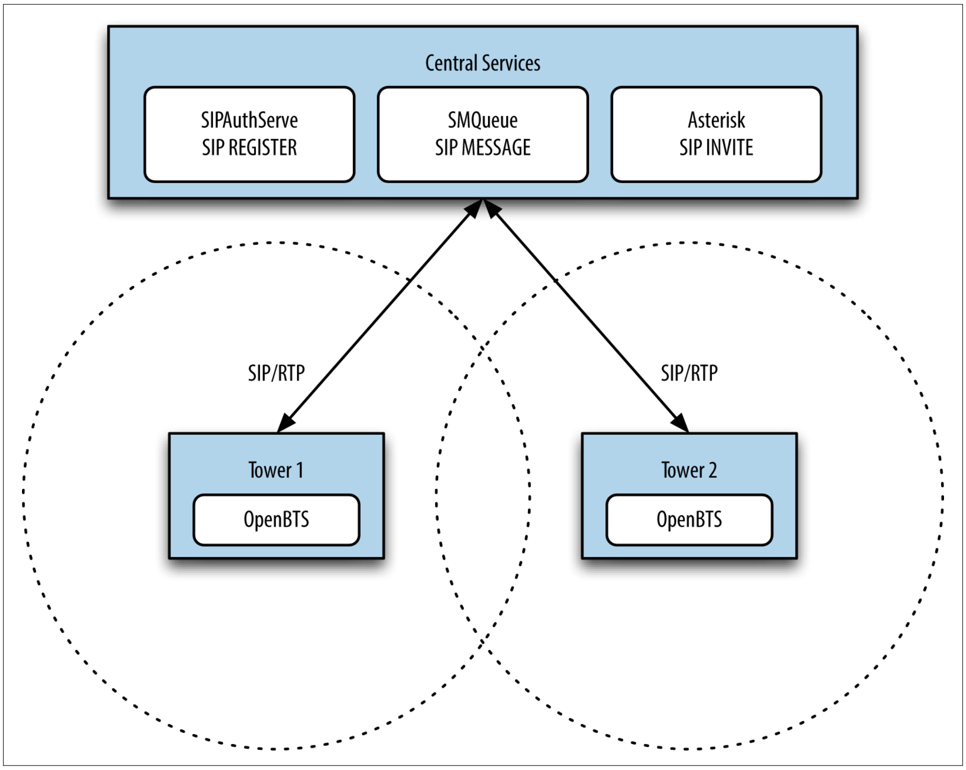


Figure 4-1. Multinode topology

Towers are backhauled over IP to Central Services, allowing multiple coverage areas to share the same subscriber database and configuration.

In the following instructions, Central Services will be located at 192.168.158.100 and towers will start at 192.168.158.201.

Central Services Setup

Instead of extracting Central Services from your existing network-in-a-box configuration, it's easier to just move the OpenBTS instance and radio hardware to a new home. This avoids having to transplant the subscriber data and configurations for SMQueue, SIPAuthServe, and Asterisk.

Remove OpenBTS

To make sure OpenBTS doesn't start on boot and continually fail because it's missing a radio, uninstall it now:

```
$ sudo apt-get remove openbts
```

Configure Logging

Each tower independently, as well as Central Services, will generate logging information. You need to set up Central Services to accept logging information from all towers so all activity across the network can be captured in a single file.

This step is not necessary to establish a functional multinode network but is extremely useful when debugging. Tracking down errors across multiple towers with separate logs is not very much fun.

Edit */etc/rsyslog.conf* with your favorite editor and uncomment the two lines directly after "provides UDP syslog reception":

```
# provides UDP syslog reception
$ModLoad imudp
$UDPServerRun 514
```

These two lines mean that rsyslog will accept UDP log traffic on port 514. To apply these settings, a restart of the service is needed:

```
$ sudo service rsyslog restart
```

Asterisk, SMQueue, and SIPAuthServe

There is nothing to reconfigure in these components. They don't care if the OpenBTS instance is communicating locally or across an IP network.

Tower Setup

Jump back to "Operating System and Development Environment Setup" on page 7 and follow along on a new virtual machine or fresh server to get your new tower's OS and components up and running. Make sure to pick a unique hostname for the tower so it's distinguishable in the centralized logs. Also, save some time at the end of those instructions by only installing OpenBTS.

These instructions assume that both your tower and Central Services machines are located on the same network subnet. Once you log in to the new tower installation, make sure you can ping Central Services (press Ctrl-C to stop the `ping` command):

```
$ ping 192.168.158.100
PING 192.168.158.100 (192.168.158.100) 56(84) bytes of data.
64 bytes from 192.168.158.100: icmp_req=1 ttl=128 time=4.28 ms
64 bytes from 192.168.158.100: icmp_req=2 ttl=128 time=4.34 ms
64 bytes from 192.168.158.100: icmp_req=3 ttl=128 time=4.48 ms
64 bytes from 192.168.158.100: icmp_req=4 ttl=128 time=4.48 ms
^C
--- 192.168.158.100 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 4.285/4.399/4.484/0.086 ms
```

Configure SIP Proxies

OpenBTS connects to all other components using SIP. Now that these components are on another host, the `SIP.Proxy` key needs to be adjusted accordingly. Also, now that OpenBTS is speaking to services on IP addresses other than localhost, it needs to know what address to send its own requests to. The `SIP.Local.IP` key allows you to set the IP address that other services should use when contacting OpenBTS or replying to its requests. Set it now using your tower's IP address:

```
OpenBTS> config SIP.Local.IP 192.168.158.201
SIP.Local.IP changed from "127.0.0.1" to "192.168.158.201"
SIP.Local.IP is static; change takes effect on restart
```

Restart OpenBTS to apply this static key:

```
$ sudo stop openbts
$ sudo start openbts
```

Now, each `SIP.Proxy` key can be updated to point to Central Services, so registration, voice, and SMS traffic are sent there instead of to localhost:

```
OpenBTS> config SIP.Proxy.Registration 192.168.158.100:5064
SIP.Proxy.Registration changed from "127.0.0.1:5064" to "192.168.158.100:5064"
OpenBTS> config SIP.Proxy.SMS 192.168.158.100:5063
SIP.Proxy.SMS changed from "127.0.0.1:5063" to "192.168.158.100:5063"
OpenBTS> config SIP.Proxy.Speech 192.168.158.100:5060
SIP.Proxy.Speech changed from "127.0.0.1:5060" to "192.168.158.100:5060"
```

Make sure to include the port information. Each component runs on a different port and routing requests to the incorrect port would break functionality.

Force a handset to register via this new topology to test things out. Voice and SMS should also function. Verify this before moving on.

Configure Logging

Logging information that each tower generates can be sent to Central Services. The logging daemon on Central Services has already been changed to accept this traffic; now it's time to tell our tower nodes to send it.

Edit */etc/rsyslog.d/OpenBTS.conf* with your favorite editor so it contains a new line directly below the existing `local7` entry:

```
local7.*              /var/log/OpenBTS.log
local7.*              @the-central-services-ip
```

This line means that in addition to logging information to the *OpenBTS.log* file on disk, the same messages will also be sent to Central Services for logging there. To apply these settings, a restart of the service is needed:

```
$ sudo service rsyslog restart
```

Topology Reworked

So far your network is functionally the same. There is still one coverage area created with one logical tower and Central Services—they've just been split onto two separate physical machines. The next step involves adding additional neighboring towers to the network and configuring them to appear as a single RAN.

Adding Neighboring Towers

The tower setup section can now be repeated for as many physical towers as you'd like to add. Once they have been configured to at least function as an independent tower, they must be configured to behave as a proper neighboring tower within the network. Additionally, all existing towers must be informed about this new tower.

So far, assume that there is only one tower in your network—the one you just ported from the network-in-a-box layout. Take a look at its identity parameters now:

```
OpenBTS> config Identity
GSM.Identity.BSIC.BCC 2     [default]
GSM.Identity.BSIC.NCC 0     [default]
GSM.Identity.CI 10     [default]
GSM.Identity.LAC 1000     [default]
GSM.Identity.MCC 001     [default]
GSM.Identity.MNC 01     [default]
GSM.Identity.ShortName GroundControl
```

Of these parameters, some must be unique and others must be identical across all neighbors. Table 4-1 is a convenient summary. The configuration schema also embeds this information in the `scope` field. The `scope` field will tell you if the key must be unique or identical among neighboring towers or across all towers in your network.

Table 4-1. Configuration scope

Identical	Unique
`GSM.Identity.MCC`	`GSM.Identity.LAC`
`GSM.Identity.MNC`	`GSM.Identity.CI`
`GSM.Identity.BSIC.NCC`	`GSM.Identity.BSIC.BCC`
`GSM.CellSelection.NCCsPermitted`	`GSM.Radio.C0`
`GSM.Identity.ShortName`	

Some background information will be presented, so each parameter is understood, along with a step-by-step guide to setting up your next tower.

Must Be Identical

The `GSM.Identity.MCC` and `GSM.Identity.MNC` keys are the highest-level identification for any network, indicating what country this network is in and which carrier is running it. Regulatory bodies assign both of these IDs. Test networks always use `001` and `01`, respectively. If this isn't a test network, you will probably have paid money for a license and be very aware of the appropriate value.

The `GSM.Identity.BSIC.NCC` key is your Network Color Code (NCC), a piece of information that handsets use to very quickly determine if they can gain access to the network. All carriers in a given area must have unique color codes.

Related to the last key is `GSM.CellSelection.NCCsPermitted`. This key is used when working with partner networks to express the following: "not only is my NCC OK but these other NCCs are permitted." The `NCCsPermitted` key is a bitwise flag, not an integer value. Regardless, all towers should have the same NCCsPermitted setting.

The `GSM.Identity.ShortName` key should be identical across all towers but it is not required to be. A convenient debugging technique when deploying a multinode network is to set unique shortnames for each tower. If your handset reliably shows the shortname, you now instantly know what tower your handset is camped to while moving around.

Step-by-step

The step-by-step instructions for this section are pretty basic: make sure all of these keys are identical to the existing towers in your network.

Must Be Unique

The first key, `GSM.Radio.C0`, must be unique because it is the ARFCN and thus RF being used on this tower. If physically adjacent towers are using the same frequency, it will create interference and deny service to any participants in the overlapping signal region.

Not only do the C0 ARFCNs need to be unique, but they also must not be numerically adjacent. Because the signal spacing is only 200 KHz and the signal bandwidth is 270 KHz, adjacent ARFCNs overlap. Use at least every other ARFCN when deploying physically adjacent towers (e.g., tower 1 uses ARFCN 151 and tower 2 uses ARFCN 153).

The `GSM.Identity.BSIC.BCC` must also be unique for physically adjacent towers. It signals the Base Station Color Code (BCC), a 3-bit field allowing seven unique values.

Your licensed C0 ARFCNs and the BCC are very limited in number and a color map might be helpful in assigning values to towers if you plan on having a lot of them. A color map shows the geographic location of all towers and assigns each one a color. Each color corresponds to an ARFCN number and BCC pair as shown in Figure 4-2. If your planning shows that no two adjacent towers have the same "color," then Mobility and Handovers will not be hindered by clashing ARFCNs or BCCs.

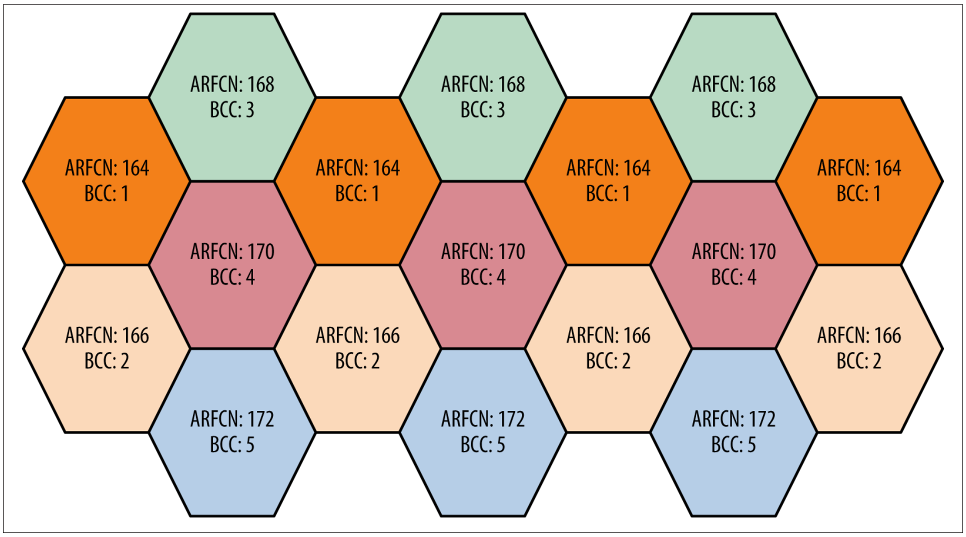


Figure 4-2. Color map

GSM.Identity.LAC must be unique across all towers in your network. As a handset moves across tower boundaries it realizes that the Location Area Code is changing and should perform another LUR. This LUR is translated into a SIP REGISTER, and SIPAuthServe can update the IP address for the tower where this handset is reachable. The requirement that the LAC be unique is specific to OpenBTS. Traditional networks have LACs that correspond to much larger geographic areas containing multiple towers.

GSM.Identity.CI is the Cell ID (CI) and must be unique across all towers in your network.

Step-by-step

Assign a color to the new tower and use the corresponding C0 value and BCC:

```
OpenBTS> config GSM.Radio.C0 168
GSM.Radio.C0 changed from "151" to "168"
GSM.Radio.C0 is static; change takes effect on restart
OpenBTS> config GSM.Identity.BSIC.BCC 3
GSM.Identity.BSIC.BCC changed from "2" to "3"
```

As there are plenty of LAC and CI values, simply use the next one for each tower:

```
OpenBTS> config GSM.Identity.LAC 1001
GSM.Identity.LAC changed from "1000" to "1001"
OpenBTS> config GSM.Identity.CI 11
GSM.Identity.CI changed from "10" to "11"
```

Restart OpenBTS to apply the changes:

```
$ sudo stop openbts
$ sudo start openbts
```

Neighbor List and Command

Your towers are now configured so as not to stomp on each other either physically through their transmission frequencies or logically in their ID numbering schemes. They must still be told about each other in the configuration. This is accomplished using the GSM.Neighbors key. Each tower must have its GSM.Neighbors key set to a space-separated list of IP addresses corresponding to all physically adjacent towers but excluding itself.

On your new tower (192.168.158.202), enter the following to tell it about your original tower (192.168.158.201):

```
OpenBTS> config GSM.Neighbors 192.168.158.201
GSM.Neighbors changed from "" to "192.168.158.201"
```

Do the same for your original tower so it knows about the new tower:

```
OpenBTS> config GSM.Neighbors 192.168.158.202
GSM.Neighbors changed from "" to "192.168.158.202"
```

Still on your original tower, 192.168.158.201, check out the `neighbors` command:

```
OpenBTS> neighbors
host                  C0  BSIC FreqIndex Noise ARFCNs TCH-Avail Updated Holdoff
------------------    --- ---- --------- ----- ------ --------- ------- -------
192.168.158.202:16001 168 2    0         68    1      7         16      0
```

The OpenBTS peering protocol automatically queries for information about the neighboring base stations defined in `GSM.Neighbors`. The `neighbors` command provides access to this information. Here you see the C0 ARFCN, current noise level, total ARFCN count, and available traffic channels (TCHs).

Neighbor-Enabled Commands

Now that you have a functional pair of neighbors, the `chans` command will also provide additional information during an active call. This information is related to the current reception level of the strongest neighboring tower that the handset reported. To see it, use the `chans` command with a –l flag:

```
OpenBTS> chans -l
CN TN chan  transaction LAPDm           recyc Signal RSSI RSSP SNR  FER  BER TA
      type  id          state                 dB     dB   dB        pct  pct sym
0  1  TCH/F T100        LinkEstablished false 26     -50  -24  38.7 0.00 0   0.9

                  TXPWR TA_DL RXLEV_DL BER_DL Time IMSI     Neighbor    Handover
                  dBm   sym   dBm      pct    M:S           ARFCN(dBm)  C0:BSIC
                  7     0     -93      2.26   0:25 2140...  168         (-76)
```

Coverage Overlap Tuning

With each tower tuned individually, the trickiest part is to tune them in relation to each other. Take a look again at Figure 4-1. The area of overlap between Tower 1 and Tower 2 is crucial. This area must be wide enough so that a handset traveling through it can contact both towers reliably for as long as it takes to perform a Handover. The terrain in this area must be taken into account as well as the potential handset velocity. The overlap must not be too excessive as to waste resources though. Finding this balance between reliable overlap and optimized overall coverage is key. Again, entire businesses can be built on this expertise and a deep dive falls outside the scope of a "getting started" book.

Handset Mobility only has a single key to configure: `GSM.CellSelection.CELL-RESELECT-HYSTERESIS`. This key indicates to the handset how much better a potential neighboring tower's signal should be before it recamps.

OpenBTS can more directly control Handover. There are several keys available for tuning Handover behavior:

```
OpenBTS> config Handover
GSM.Handover.FailureHoldoff 20     [default]
GSM.Handover.Margin 15     [default]
GSM.Handover.Ny1 50     [default]
GSM.Timer.Handover.Holdoff 10     [default]
```

The `GSM.Handover.Margin` key is similar to `GSM.CellSelection.CELL-RESELECT-HYSTERESIS` except that it's used for Handover operations. With both of these settings, it is important to remember that if they are set too high (e.g., the tower must be much stronger before switching), there is a risk that the handset will not jump over in time. If they are set too low (e.g., the tower must only be slightly stronger before switching), the network can be hit with a spike of traffic from handsets located in an area where both signals are approximately equal as the handsets rapidly jump back and forth.

Even more parameters are available with `devconfig` if you would like to experiment with the weights inside the OpenBTS Handover algorithm:

```
OpenBTS> devconfig Handover
GSM.Handover.FailureHoldoff 20     [default]
GSM.Handover.History.Max 32     [default]
GSM.Handover.Margin 15     [default]
GSM.Handover.Ny1 50     [default]
GSM.Handover.RXLEV_DL.History 6     [default]
GSM.Handover.RXLEV_DL.Margin 10     [default]
GSM.Handover.RXLEV_DL.PenaltyTime 20     [default]
GSM.Handover.RXLEV_DL.Target 60     [default]
GSM.Timer.Handover.Holdoff 10     [default]
```

CHAPTER 5

GPRS

Mobile networks have, in many areas of the world, been reduced to being data networks. Over-the-top (OTT) services like WhatsApp and Skype only need a data pipe and participants can connect regardless of carrier. Fees between the participants are also not dependent upon geographic location, unlike local versus long-distance charges.

GPRS is much too slow to support bidirectional streaming video but can suffice for a low-quality voice call. Its speeds are ideal for email and OTT text messaging.

The world of sensors and infrastructure such as heat and flow sensors or electrical and parking meters also needs data connectivity. These low-bandwidth machine-to-machine (M2M) devices, now referred to as Internet of Things (IoT) devices, are a very common use for GPRS.

GPRS is actually not a part of GSM. It was developed after GSM had been standardized and is usually referred to as 2.5G, whereas plain GSM is 2G. OpenBTS abstracts these differences and presents a unified configuration where possible.

Enabling/Disabling

By default, the GPRS service is disabled in OpenBTS. Turn it on now by toggling the `GPRS.Enable` key:

```
OpenBTS> config GPRS.Enable 1
GPRS.Enable changed from "0" to "1"
GPRS.Enable is static; change takes effect on restart
```

Restart OpenBTS to apply this static key:

```
$ sudo stop openbts
$ sudo start openbts
```

Once OpenBTS has restarted, log back in to its command line and use the `gprs list` command to confirm that OpenBTS has set up a few channels for GPRS:

```
OpenBTS> gprs list
 PDCH ARFCN=166 TN=1 FER=0%
 PDCH ARFCN=166 TN=2 FER=0%
```

Central Services

GPRS does not rely on any additional components but some configuration must be in place on your Linux host for things to work correctly. This should be taken care of already from the range-configs package during setup but this is how to double-check that things are in order.

The handsets' IP traffic is piped through OpenBTS and into a virtual network interface named `sgsntun`. You can confirm now that OpenBTS has created it by using `ifconfig`:

```
$ ifconfig sgsntun
sgsntun  Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
         UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:500
         RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

The virtual network interface also needs routes and rules applied to it for the iptables Linux firewall. Example rules are located in */etc/OpenBTS/iptables.rules* and can be modified if needed to change the gateway interface name. By default, they are written for `eth0`. Apply the rules now manually:

```
$ sudo iptables-restore < /etc/OpenBTS/iptables.rules
```

To have the system apply these rules every time your `eth0` interface comes up, modify */etc/network/interfaces* to add the final line below, which contains "pre-up":

```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet dhcp
        pre-up iptables-restore < /etc/OpenBTS/iptables.rules
```

With the tunnel device present and the rules applied, your Linux host should be in order.

Connecting

While your handset should perform an LUR and join the network once it's back up, there may be additional steps to make sure the GPRS service is recognized and usable.

In GPRS this is not an LUR; it is called GPRS Attach. The handset may not perform this Attach for several reasons.

If you are using another carrier's SIM card, the handset GPRS subsystem will probably consider itself to be roaming when joining your OpenBTS network. The GPRS subsystem will not attempt to Attach unless "use data roaming" has been switched on in your handset.

Also double-check your handset's Access Point Name (APN) settings. OpenBTS does not care what information is put in the name, username, and password fields but the handset may not make an Attach request until something has been entered.

Android has a very peculiar bug when it comes to APN entries. In some versions, you are allowed to add a new APN but it will not save unless the Mobile Country Code (MCC) and Mobile Network Code (MNC) you enter match those defined on your SIM card. This failure is silent and no indication is given that the new APN settings will not be used. They also do not show up in the list of APNs on the device. However, if you replace the SIM card with one that matches the MCC and MNC you entered, your APN information magically reappears.

All that said, some phones may still take a few minutes to Attach. They may be presenting old network access information to OpenBTS before giving up and starting a fresh Attach. Once they have successfully attached, the handset's IP address is visible via the `sgsn list` command:

```
OpenBTS> sgsn list
 GMM Context: imsi=001010000000009 ptmsi=0x47001 tlli=0xc0047001
              state=GmmRegisteredNormal age=32 idle=0 MS#1,TLLI=c0047001,7d4373ae
              IPs=192.168.99.1
```

Troubleshooting

If you are able to connect but have not received an IP address, the firewall setting may be getting in the way. Normally this is not a problem, but depending on the Linux installation and IP network configuration it is possible. Disable the firewall now and restart OpenBTS to apply the change:

```
OpenBTS> config GGSN.Firewall.Enable 0
GGSN.Firewall.Enable changed from "1" to "0"
GGSN.Firewall.Enable is static; change takes effect on restart
```

OpenBTS also tries to detect your DNS server settings and pass them on to the handsets. If you find the handset unable to resolve domain names, try setting a DNS server manually and restart OpenBTS to apply the change:

```
OpenBTS> config GGSN.DNS 8.8.8.8
GGSN.DNS changed from "" to "8.8.8.8"
GGSN.DNS is static; change takes effect on restart
```

The Gateway GPRS Support Node (GGSN) and Serving GPRS Support Node (SGSN) are typically separate entities in a GSM network but are embedded in OpenBTS directly. To gain some more insight into what is happening in those components, you can set up a separate logfile and observe its contents as handsets interact with the GPRS service on your network. Set the `GGSN.Logfile.Name` key to a file path where you would like the logs sent and restart OpenBTS to apply the change:

```
OpenBTS> devconfig GGSN.Logfile.Name /tmp/GGSN.log
GGSN.Logfile.Name changed from "" to "/tmp/GGSN.log"
GGSN.Logfile.Name is static; change takes effect on restart
```

Now, as activity on the GPRS network passes through the GGSN, new entries are added to this log file. Use the `tail` command with an `-f` flag to monitor the contents:

```
$ tail -f /tmp/GGSN.log
 19:44:04.5:  GGSN.MS.IP.Base=192.168.99.1
 19:44:04.5:  GGSN.MS.IP.MaxCount=254
 19:44:04.5:  GGSN.MS.IP.Route=192.168.99.0/24
 19:44:04.5:  GGSN.IP.MaxPacketSize=1520
 19:44:04.5:  GGSN.IP.ReuseTimeout=180
 19:44:04.5:  GGSN.Firewall.Enable=0
 19:44:04.5:  GGSN.IP.TossDuplicatePackets=0
 19:44:04.5:GGSN: DNS servers: 8.8.8.8 0.0.0.0
 19:44:04.5:ip link set sgsntun up
 19:44:04.7:ip route add to 192.168.99.0/24 dev sgsntun
```

Performance Tuning

OpenBTS attempts to intelligently divide resources between standalone dedicated control channels (SDCCHs) for signaling and TCH for media. However, it does not yet have a mechanism to balance different types of TCH usage.

Voice versus GPRS

Both voice and GPRS traffic use time slots that carry TCH logical channels. If your network will be deployed to primarily serve GPRS instead of voice, you should adjust the `GPRS.Channels.Min.C0` key that specifies the minimum number of available TCH time slots that should be used for GPRS. The default value is two. To view the current number of available channels, use the `load` command:

```
OpenBTS> load
== GSM ==
SDCCH load/available: 0/4
TCH/F load/available: 0/5
PCH load: active, total: 0, 0
```

```
AGCH load: active, pending: 0, 0
== GPRS ==
current PDCHs: 2
utilization: 0%
```

Here we see that there are seven total TCH channels available: five for GSM voice (GSM:TCH/F) and two available for GPRS data (GPRS:PDCHs). To maximize the network for GPRS, set `GPRS.Channels.Min.C0` to seven, the total number of TCH channels available, and restart OpenBTS to apply the change:

```
OpenBTS> config GPRS.Channels.Min.C0 7
GPRS.Channels.Min.C0 changed from "2" to "7"
GPRS.Channels.Min.C0 is static; change takes effect on restart
```

Rerunning `load` will show that all TCH channels have been assigned to GPRS:

```
OpenBTS> load
== GSM ==
SDCCH load/available: 0/4
TCH/F load/available: 0/0
PCH load: active: 0 total: 0
AGCH load: active: 0 () pending: 0
== GPRS ==
current PDCHs: 7
utilization: 0%
```

OpenBTS allows you to change all time slot assignments if you'd like to control them manually. You can adjust how many Combination 1 (TCH) versus Combination 7 (SDCCH) time slots are assigned. You can also adjust the ordering of those time slots, and where the GPRS time slots should appear in multi-ARFCN systems. Take a look at all the keys by searching for "Channels":

```
OpenBTS> config Channels
GPRS.Channels.Min.C0 2     [default]
GPRS.Channels.Min.CN 0     [default]
GSM.Channels.C1sFirst 0     [default]
GSM.Channels.NumC1s auto     [default]
GSM.Channels.NumC7s auto     [default]
GSM.Channels.SDCCHReserve 0     [default]
```

Individual Handset Throughput

Handsets can use more than one time slot concurrently to access GPRS. The number of concurrent time slots that are supported for uplink and downlink is known as a multislot class. By default, OpenBTS is set to support a 3+2 multislot class: three concurrent time slots for downlink and two for uplink. This does not mean that one handset will dominate all five time slots concurrently, but you may now see why adding as many time slots as possible to a GPRS-focused network is important. A higher multislot class will deliver higher speed data to a single handset, but the network will become more quickly congested as multiple handsets attempt to use it.

You can adjust the supported multislot class using the following two keys:

```
OpenBTS> config Multislot
GPRS.Multislot.Max.Downlink 3     [default]
GPRS.Multislot.Max.Uplink 2     [default]
```

Coverage Area versus Throughput

GPRS defines four different coding schemes (CSs) for the data being delivered. They are appropriately named CS1, CS2, CS3, and CS4. Coding Scheme 1 has the lowest throughput but the highest reliability. It assigns more bits to be used for backing out errors introduced during the radio transmission. Coding Scheme 4, on the other hand, has the highest throughput but is susceptible to errors during transmission. These transmission errors limit the usable coverage area for each coding scheme.

OpenBTS supports CS1 and CS4. You can choose between low throughput but a reliable and, thusly, larger coverage area, or higher throughput with less robust error correction, resulting in a smaller coverage area. By default, OpenBTS has both enabled, but they can be adjusted individually on the uplink and downlink:

```
OpenBTS> devconfig Codecs
GPRS.Codecs.Downlink 1,4     [default]
GPRS.Codecs.Uplink 1,4     [default]
```

Expectations

GPRS is not fast, but data is data. Where WiFi is unavailable, relatively large areas can have modem-speed Internet access whether that be for M2M/IoT devices or people contacting their friends and relatives. Some expected throughputs per handset are illustrated in Table 5-1.

Table 5-1. Multislot class and coding scheme throughputs

Slots down+up	1+1	2+1	2+2	3+2
CS1	9.05/9.05 kbps	18.1/9.05 kbps	18.1/18.1 kbps	27.15/18.1 kbps
CS4	21.4/21.4 kbps	42.8/21.4 kbps	42.8/42.8 kbps	64.2/42.8 kbps

These are optimal throughputs and are very dependent on network congestion, both locally on the radio as well as on the Internet uplink.

Admittedly, the OpenBTS GPRS support is not perfect yet. GSM voice and SMS are nearly "finished" pieces of work inside of OpenBTS, but GPRS still has improvements and optimizations that can be made. Most of these improvements have to do with algorithms that divvy up access to the radio medium and how packets are rescheduled. Optimizing these will greatly improve the GPRS experience in OpenBTS.

CHAPTER 6

OpenRegistration

OpenRegistration is an OpenBTS-specific feature that provides a WiFi captive portal-like implementation for mobile networks. Captive portals are familiar to anyone who has used an airport or hotel's public WiFi connection. Your device can connect to the WiFi network but is denied access to certain features until you answer a question, watch an advertisement, enter a pin, etc. The device is used to provision itself.

Similarly, OpenRegistration allows a handset to join a mobile network with initially restricted access. It may be able to dial out but the handset does not have an assigned number and as such cannot be called by other participants in the network. However, it can be used to provision its own number via SMS.

This type of network is very useful in any ad hoc installation where the users are temporary and fluid or the network itself is only temporarily needed: emergency response, remote work areas, tourist destinations, large festivals, etc. Because an administrator is not needed to create accounts and assign numbers, OpenRegistration networks are easier to deploy and still very useful for the users.

Enabling

To get started with an OpenRegistration network, the feature itself must be enabled. First, take a look at the keys:

```
OpenBTS> config OpenRegistration
Control.LUR.OpenRegistration (disabled)     [default]
Control.LUR.OpenRegistration.Message Welcome to the test network.  Your
IMSI is      [default]
Control.LUR.OpenRegistration.Reject (disabled)     [default]
Control.LUR.OpenRegistration.ShortCode 101     [default]
```

To enable OpenRegistration, the `Control.LUR.OpenRegistration` key must be set to a regular expression. A regular expression (sometimes written "regex") is a way of defining a pattern to be matched. There are a few standards for them and more information can be found on Wikipedia. IMSIs matching this regular expression will be given access to the network. A few examples of patterns and their effect are in Table 6-1.

Table 6-1. OpenRegistration regular expressions

Regular Expression	OpenRegistration Effect
.*	Match all IMSIs
^460	Match any IMSI starting with "460", the MCC for China
^46002	Match any IMSI from China Mobile (MCC=460, MNC=02)
460027217080245	Match only IMSI "460027217080245"
0	Match any IMSI containing a "0"
1	Match any IMSI containing a "1"
1234$	Match any IMSI ending in "1234"

For this book, OpenRegistration will be enabled to accept any IMSI it encounters:

```
OpenBTS> config Control.LUR.OpenRegistration .*
Control.LUR.OpenRegistration changed from "" to ".*"
```

There is an additional key that allows the explicit rejection of IMSIs it matches. This is convenient if you would like to allow every IMSI *except* a specific group. Drawing again from the table above, set the network to explicitly reject any IMSI from China Mobile:

```
OpenBTS> config Control.LUR.OpenRegistration.Reject ^46002
Control.LUR.OpenRegistration.Reject changed from "" to "^46002"
```

Make sure you are confident in understanding how these patterns work before deploying them in the wild. The nightmare scenario is that someone inadvertently joins your open network, has a heart attack, and cannot call emergency services because your network does not support it. Double-check that your network's `GSM.RACH.AC` is still set to the default of `0x0400`. This advertises "Emergency Calls Not Supported" in your beacon.

Personalizing

Users joining your network will be greeted with the contents of the `Control.LUR.OpenRegistration.Reject` key. Change it now to be more relevant for your installation:

```
OpenBTS> config Control.LUR.OpenRegistration.Message Welcome to IslandNet! Call
or text 101 for assistance. Your IMSI is:
Control.LUR.OpenRegistration.Message changed from "Welcome to the test network.
Your IMSI is " to "Welcome to IslandNet! Call or text 101 for assistance. Your
IMSI is: "
```

This instructs newly joined members to seek assistance by dialing 101. Your network should have someone assigned to the 101 number or implement an automated voice menu there. Users may also send an SMS to that number as instructed. By default, SMQueue has a shortcode handler for the 101 number and will begin a dialogue with the users to get their desired phone number assigned. The `SC.Register.*` parameters in SMQueue control which shortcode is use and how messages are worded, as well as what user-selectable number ranges are allowed. SMQueue does not have a CLI interface yet but `nmcli.py` can be used to read and modify these parameters:

```
$ ./nmcli.py smqueue config read
$ ./nmcli.py smqueue config update Configuration.Key.Name new-value
```

If you would like to use a number other than 101, you should make sure the OpenBTS `Control.LUR.OpenRegistration.ShortCode` is set to your new number. This key sets the source address for the welcome SMS. If it's set to your new help number, users can simply reply to the initial message.

Disabling

To disable OpenRegistration again, the accept and reject patterns must be reset:

```
OpenBTS> unconfig Control.LUR.OpenRegistration
Control.LUR.OpenRegistration disabled
OpenBTS> unconfig Control.LUR.OpenRegistration.Reject
Control.LUR.OpenRegistration.Reject disabled
```

OpenBTS has now been reverted to using the subscriber database to grant or deny access instead of IMSI patterns.

CHAPTER 7

NodeManager APIs

NodeManager is a common control interface across OpenBTS, SMQueue, and SIPAuthServe used for system configuration and monitoring. All components support some common functionality (such as configuration manipulation) with specialized APIs available for component-specific tasks or data sources. The NodeManager APIs are divided into two main categories: request/response and streaming. Both types use JSON formatted messages and communicate over ZeroMQ, a library that creates simple and robust socket connections between processes. NodeManager APIs were implemented to simplify the remote management of multiple OpenBTS instances, as well as Central Services like SMQueue and SIPAuthServe.

APIs are such a normal part of the Internet that this might not seem too exciting. However, keep Figure I-1 in mind when reading through "PhysicalStatus API" on page 74. No need for a dozen protocols and entities—the core radio measurement data is now available directly as a stream of JSON messages! Additional APIs will be added to the NodeManager interface in future releases.

nmcli.py

A small Python utility is included in the NodeManager repository named `nmcli.py`. It requires a few dependencies to be installed, but the `build.sh` script (in "Building the Code" on page 9) should have already done this. To make sure you have them, here are the manual commands:

```
$ sudo add-apt-repository -y ppa:chris-lea/zeromq
$ sudo apt-get update
$ sudo apt-get install libzmq3-dev libzmq3 python-zmq
```

The capabilities of `nmcli.py` are always being expanded as new NodeManager APIs are implemented. To see the current usage, run `nmcli.py` without any arguments:

```
$ ./nmcli.py
```

The utility of `nmcli.py` is not as a professional interface to NodeManager APIs, but rather a development tool to quickly format messages and test new API endpoints as they are implemented. It will be used in this chapter to demonstrate the different API endpoints. First the `nmcli.py` usage will be listed, followed by the formatted JSON messages that are exchanged.

Version API

All components implement `version`, a request/response API. It is a trivial API but exposes a very important piece of information about a component running on a remote node: its current version. Use `nmcli.py` to ask SMQueue for its version now:

```
$ ./nmcli.py smqueue version
```

The JSON request it sends to SMQueue is quite simple:

```
{
        "command" : "version"
}
```

…and the response back is equally unambiguous:

```
{
        "code" : 200,
        "data" : "release 5.0.0-prealpha+667f928701 CommonLibs:3ad343b97b
built 2014-09-17T21:26:56 "
}
```

Configuration API

All components implement `config`, a request/response API. This API is used to modify configuration values in each component. Each component's configuration schema validates requests before they are accepted, which makes the API preferable to simply updating values in the appropriate SQLite3 databases.

Read All Keys

Use `nmcli.py` to ask SMQueue for all configuration parameters now:

```
$ ./nmcli.py smqueue config read
```

It generates a request with a command of "config" and an action of "read":

```
{
        "command" : "config",
        "action" : "read"
}
```

SMQueue responds with a gigantic list of every configuration key it supports (clipped for space):

```
{
        "code" : 200,
        "data" : [
                ...
        ]
}
```

Read One Key

The API also supports reading a particular key. Use `nmcli.py` to ask SMQueue for the `SC.Register.Code` parameter now:

```
$ ./nmcli.py smqueue config read SC.Register.Code
```

It generates a request with a `command` of "config," an `action` of "read," and a new `key` field of `SC.Register.Code`:

```
{
        "command" : "config",
        "action" : "read",
        "key" : "SC.Register.Code"
}
```

SMQueue responds with this specific configuration key's schema and value information:

```
{
        "code" : 200,
        "data" : {
                        "defaultValue" : "101",
                        "description" : "Short code to the application which↵
 registers the sender to the system.",
                        "key" : "SC.Register.Code",
                        "scope" : 0,
                        "static" : false,
                        "type" : "string",
                        "units" : "",
                        "validValues" : "^[0-9]{3,6}$",
                        "value" : "101",
                        "visibility" : "customer - can be freely changed by
the customer without any detriment to their system"
        }
}
```

Requesting a key that does not exist will result in a 404 response like this from the component:

```
{
        "code" : 404
}
```

Update

A configuration API would not be too useful if it did not allow new values to be set. The message's `action` field is changed to "update" and a new `value` field must be specified that contains the desired change. Use `nmcli.py` to update the last key we queried—`SC.Register.Code`—to 555 instead of 101:

```
$ ./nmcli.py smqueue config update SC.Register.Code 555
```

Notice the new `value` field and updated `action` field:

```
{
        "command" : "config",
        "action" : "update",
        "key" : "SC.Register.Code",
        "value" : "555"
}
```

The response includes two pieces of information:

```
{
        "code" : 204,
        "dirty" : 0
}
```

The 204 code indicates that the action was successful but there is no data that needs reporting. The `dirty` field lets you know that there aren't any static keys that need to be applied, a.k.a., your configuration is live. If `dirty` is nonzero, the component will need to be restarted eventually to fully apply the changes.

Here are the other response possibilities:

- 304: nothing to change, old and new values match
- 404: unknown key (just like the read command)
- 406: new value is invalid
- 409: new value conflicts with another configuration key
- 500: database error, storing the new value failed

PhysicalStatus API

OpenBTS is the only component that implements `PhysicalStatus`, a streaming API. It is not activated by default; use the `NodeManager.API.PhysicalStatus` key to enable it. To maintain backward compatibility, the `PhysicalStatus` API supports different schemas as they are released. Currently, only schema 0.1 has been implemented. Setting the key to the value 0.1 will activate the PhysicalStatus API and stream event data using schema 0.1. Since you already have `nmcli.py` handy, use it now to activate the API:

```
    $ ./nmcli.py openbts config update NodeManager.API.PhysicalStatus 0.1
```

With this stream activated, you need to attach a client to view the data that is being produced. An example client named *JSONEventsClient.cpp* is provided in the *OpenBTS/apps* repository directory. It is compiled by default when building OpenBTS, but not installed onto the system when using the package. This client connects to the ZeroMQ port defined by `NodeManager.Events.Port` and waits for the API to publish events.

Go into your clone of the OpenBTS source code now and execute the client in place:

```
    $ cd dev/openbts/apps
    $ ./JSONEventsClient
```

Nothing interesting will happen until there is activity on your OpenBTS installation. Power cycle a handset, send an SMS, or place a call, and `PhysicalStatus` readings will start showing up on the console. An example of these readings follows:

```
    {
            "name" : "PhysicalStatus",
            "timestamp" : "18446744072283447705",
            "version" : "0.1",
            "data" : {
                    "burst" : {
                            "RSSI" : -49.4808,
                            "RSSP" : -27.4808,
                            "actualMSPower" : 11,
                            "actualMSTimingAdvance" : 0,
                            "timingError" : 1.59709
                    },
                    "channel" : {
                            "ARFCN" : 153,
                            "IMSI" : "001010000000001",
                            "carrierNumber" : 0,
                            "timeslotNumber" : 0,
                            "typeAndOffset" : "SDCCH/4-1",
                            "uplinkFrameErrorRate" : 0
                    },
                    "reports" : {
                            "neighboringCells" : [],
                            "servingCell" : {
                                    "RXLEVEL_FULL_dBm" : -67,
                                    "RXLEVEL_SUB_dBm" : -67,
                                    "RXQUALITY_FULL_BER" : 0,
                                    "RXQUALITY_SUB_BER" : 0
                            }
                    }
            }
    }
```

These readings show the physical radio burst information as traffic flows between the handset and base station. These readings are also known as Measurement Reports and happen in the background as part of the GSM standard. They provide the base station with information used to appropriately adjust transmit power on the handset, trigger Handover messages, and calculate Timing Advance.

So what is the value in accessing this data? Many applications can use this raw meta information about the radio conditions of a network to make smart decisions on power consumption, share the load more intelligently, and so on.

It is currently being used in search-and-rescue operations. Helicopters in Iceland have been outfitted with portable base stations running OpenBTS and are being flown into remote areas where lost hikers are known to be. OpenRegistration is configured so all handsets in the area are automatically allowed to join the network. The radio reception levels and transmit power from both base station and handset are recorded constantly as the network sends out test SMS messages. Alongside this radio information, the current latitude, longitude, and altitude of the helicopter is recorded. This information can then be analyzed algorithmically to determine the location of any handsets in the area within 65 m of accuracy at distances of up to 35 km. More information about this project can be found here.

CHAPTER 8

Onward and Upward

Hopefully you've completed the book having succeeded at implementing the areas you were interested in. Here are a few jumping-off points for further research and experimentation.

Connecting to the Outside World

To connect your handsets with "real" phones in the outside world, you will need an Internet Telephony Service Provider (ITSP). ITSPs accept the SIP signaling and RTP media coming from your network and bridge them to the PSTN.

Voice

Finding ITSPs for voice service is straightforward. There are already online directories for these companies and a variety of plans are offered from prepaid minutes to flat monthly rates. Inbound phone numbers, or Direct Inward Dials (DIDs), are also offered in the majority of countries around the world. Many of these companies also supply instructions for how to connect to them when using Asterisk and can provide technical assistance for your installation.

SMS

SMS is more difficult. There are not very many ITSPs offering SIP MESSAGE to SMS service. Providers that do offer this service do not usually pair it with a voice DID so a single handset will have a different caller ID when making voice calls than when it sends SMS messages. This situation is improving but for now alternatives are needed.

Other providers offer Short Message Peer-to-Peer (SMPP) gateways or HTTP REST API gateways. Again, availability across all DIDs is unpredictable and some extra glue is needed to connect from SIP MESSAGE to either of these solutions.

An example of a successful integration using such APIs is available from nexmo.

Spectrum Regulation

The number one problem that this book alone cannot solve is the legal availability of radio frequencies that are heavily regulated. While access to spectrum has traditionally only been for the biggest players, this is beginning to change. Pressure is mounting on regulators to divide access into smaller usable chunks and distribute them to entities willing to do something useful in their regions.

The most forward-thinking spectrum policies seem to be developing in Europe. The United Kingdom, Netherlands, and Sweden have already allocated several ARFCNs' worth of spectrum for public use, including in standard GSM bands. Anyone can use these frequencies at a limited power level without a license. The United States FCC appears to be evolving and is planning a large spectrum auction in 2015 that would release as much as 28 MHz of guard-band spectrum for unlicensed use in a given area.

Groups such as the Dynamic Spectrum Alliance and initiatives like GSM whitespace research are, respectively, addressing the problem of spectrum access politically and technologically.

Switch Integrations

Work is also taking place to standardize the SIP signaling that OpenBTS needs—primarily, the use of GSM and UMTS authentication algorithms instead of MD5.

So far integrations have been on a one-off basis but the effort is gathering pace. Patches to have Asterisk and FreeSWITCH directly support registration and voice calls from OpenBTS have been written and are in various stages of adoption by those projects. The Kazoo platform already has the most complete support, with registration, voice, and SMS all working.

3G Data

OpenBTS-UMTS 1.0 was released in October 2014. It is the first open source UMTS stack that we know of and adds 3G data speeds to the list of capabilities for these new "hybrid networks." More information is available here.

Open Source Hardware

If you are a hardware designer and curious about how SDRs are built, you are in luck. Not only are mobile networks being constructed with open source software, but they are being built with open source hardware. The schematics for several SDRs are completely open and available on GitHub.

- Fairwaves UmTRX
- HackRF
- Range Networks RAD1/SDR1

The Community

The OpenBTS community can provide expertise on many subjects. Sign up for the email list. There are participants with many different backgrounds: makers, hackers, researchers, and integrators welcome!

The Revolution

What would a modern mobile network look like if it were to be designed using standards from the Internet world? We're starting to see the answer to that question: massively simplified, extremely flexible, open to all kinds of new ideas, and a heck of a lot less expensive.

For the first time ever, mere mortals can download some software, pick up a radio, and build a functional mobile network. Welcome to the revolution!

APPENDIX A

Quick Reference

Here's a quick reference for many facts scattered throughout the book. I find myself looking these things up constantly and wanted to present them in a unified section.

GSM Hierarchy

This book cannot attempt to teach you GSM in a few short chapters. However, a very helpful thing to have a grasp on is a hierarchy of GSM jargon from macro to micro.

A *tower* is the mast with antennas that we're all familiar with, sometimes disguised as a cactus or tree. Each tower can be divided into multiple *sectors*. These sectors cover a portion of the possible 360º (e.g., four 90º sectors on a tower). A single sector tower covers all 360º. Each sector will be assigned one or more *ARFCNs*, sometimes also called carriers, because they are the actual pair of frequencies where data is physically exchanged.

Each ARFCN has eight *time slots*. Each time slot has a *combination* assigned to it. Each combination is composed of multiple *logical channels*. These logical channels are divided into two main categories of signaling and media. *SDCCHs* (standalone dedicated control channels) carry signaling such as handset registration traffic or SMS traffic. *TCHs* (traffic channels) carry media such as GPRS data or voice traffic. Logical channels are made up of multiple *frames* and frames are made up of *bursts*.

To see this system in action, open up the GSM Timeslot and Channel Visualizer.

Decibels and Decibel Milliwatts

Decibels can be used to express ratios between two values or, when paired with a base unit, an absolute value. When written as dB, only a ratio is being expressed. For example, an SNR of 10 dB means that the signal is 10× stronger than the noise. When written as dBm, an absolute value in milliwatts is being expressed. For example, GSM handsets can transmit with a maximum of 33 dBm or 2 W.

A convenient factor to keep in mind is that every time a change of 3 dB occurs, the ratio doubles or halves.

Table A-1. dB: Decibels and ratios

dB	Ratio
30	1000
20	100
10	10
6	~4
3	**~2**
1	1.259
0	1
-1	0.794
-3	**~0.5**
-6	~0.25
-10	0.1
-20	0.01
-30	0.001

Table A-2. dBm: Decibel milliwatts

dBm	Absolute value
30	1000 mW or 1 W
20	100 mW
10	10 mW
6	4 mW
3	2 mW
1	1.3 mW
0	1.0 mW or 1000 µW
-1	794 µW
-3	501 µW
-6	251 µW
-10	100 µW
-20	10 µW
-30	1 µW or 1000 nW

Network Ports

Table A-3. Network ports

Port	Type	Setting	Description
5062	UDP	`SIP.Local.Port`	OpenBTS SIP signaling
5063	UDP	`SIP.Local.Port`	SMQueue SIP signaling
5064	UDP	`SIP.Local.Port`	SIPAuthServe VoIP SIP signaling
5700	UDP	`TRX.Port`	Raw radio samples between OpenBTS and transceiver application
16001	UDP	`Peering.Port`	OpenBTS multinode info and event string exchange
16484	UDP	`RTP.Start` and `RTP.Range`	OpenBTS VoIP RTP media
45060	ZeroMQ RESP	`NodeManager.Commands.Port`	OpenBTS NodeManager command port
45063	ZeroMQ RESP	`NodeManager.Commands.Port`	SMQueue NodeManager command port
45064	ZeroMQ RESP	`NodeManager.Commands.Port`	SIPAuthServe NodeManager command port
45160	ZeroMQ PUB	`NodeManager.Events.Port`	OpenBTS NodeManager events port
49300	TCP	`CLI.Port`	OpenBTS command-line interface string exchange

File Paths

Table A-4. Important files

File path	Description
/etc/OpenBTS/OpenBTS.db	OpenBTS configuration database
/etc/OpenBTS/SIPAuthServe.db	SIPAuthServe configuration database
/etc/OpenBTS/SMQueue.db	SMQueue configuration database
/var/log/OpenBTS.log	Logging destination for events from all components
/etc/asterisk/sip.conf	Extra SIP account configuration for Asterisk
/etc/asterisk/extensions.conf	Extra dialplan configuration for Asterisk
/var/lib/asterisk/sqlite3dir/sqlite3.db	Subscriber registry database used by SIPAuthServe, SMQueue, and Asterisk

APPENDIX B

Operating System Installation

On either a fresh computer or in a virtual machine, follow these steps to produce a compatible minimal OS installation. If you already have a compatible operating system installed, feel free to skip to "Git Compatibility" on page 8.

When using a virtual machine, at least two cores must be assigned to the instance.

Downloading and Preparing the Boot Media

The Ubuntu 12.04 LTS installation image (an *.iso* file) is available from this page. In the "Server install CD" section, download the file linked from "PC (Intel ×86) server install CD." This is the 32-bit server installer.

Ubuntu Desktop will also work equally as well as Ubuntu Server, but the Server edition has a smaller install footprint.

Once the image has been downloaded, it can be burnt onto a physical disk for installation on standalone server hardware. If you are using a virtual machine, simply select this *.iso* image file when prompted for boot media.

If you are using VMware software, uncheck the “Use Easy Install” option.

Once the physical server has this fresh disk in its CD tray, power it on. Similarly, the new virtual machine can now be started.

Starting the Installation

The first thing you will see is a language selection menu. This is the language that will be used for the installation process. English will be used for this book but feel free to use your native language. Move the selector to highlight the desired language with the arrow keys, then press Enter.

The main menu will now appear as shown in Figure B-1.

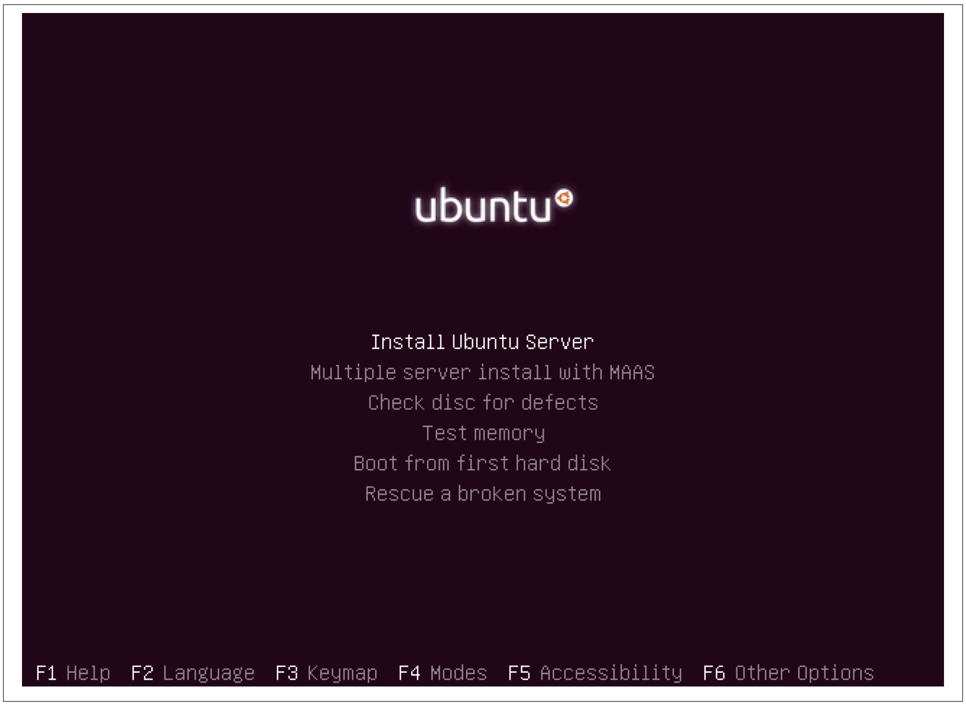

ubuntu
Install Ubuntu Server
Multiple server install with MAAS
Check disc for defects
Test memory
Boot from first hard disk
Rescue a broken system
F1 Help F2 Language F3 Keymap F4 Modes F5 Accessibility F6 Other Options

Figure B-1. Main menu

Select “Install Ubuntu Server” from this menu.

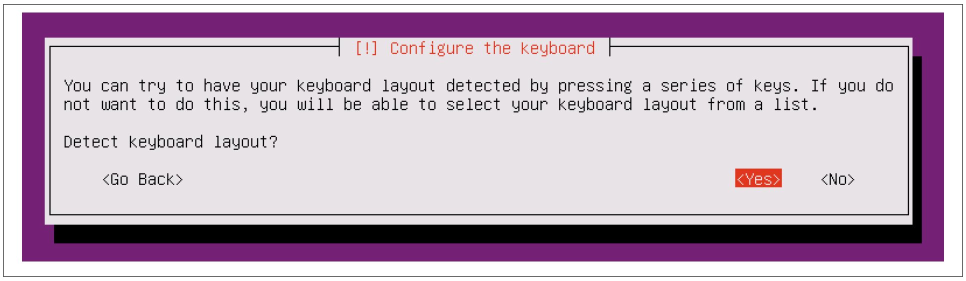
Configuring the Installation

The next screen that appears is another language selection menu. This is the language that will be used in the installed system, not the install process. Again, English will be used in this book but you can select a more comfortable option if desired. Move the selector to highlight the desired language by using the arrow keys, then press Enter.

The subsequent screen is titled "Select your location" and is used to configure the time zone and regional settings such as the formatting of dates and times. The US will be used in this book. Move the selector to highlight your country by using the arrow keys, then press Enter.

Now your keyboard layout must be selected.

You can either use keyboard layout autodetection or select the country and layout. Autodetection is a relatively simple series of requested key presses and questions that will determine the keyboard type being used. The steps in this book assume you have used the autodetection method. After the question "Detect keyboard layout?" appears, use the arrow keys to select "Yes" and press Enter to begin the autodetection process as shown in Figure B-2. When completed, confirm the detected layout by pressing Enter to choose "Continue."



```
[!] Configure the keyboard

You can try to have your keyboard layout detected by pressing a series of keys. If you do
not want to do this, you will be able to select your keyboard layout from a list.

Detect keyboard layout?

    <Go Back>                                                          <Yes>     <No>
```

Figure B-2. Keyboard selection

Finally, a hostname must be entered for your system. You may choose a hostname that fits into your network naming scheme. The default name "Ubuntu" is used in this book. Select it by pressing Tab to highlight "Continue," then press Enter to confirm.

Adding the OpenBTS User

The user "openbts" must be present on the system. The easiest way to get this into place is to configure it during installation. On the "Set up users and passwords" screen, in the "Full name of the new user" field, type "openbts," press Tab to highlight "Continue," then press Enter to confirm.

The next field, “Username for your account,” will appear. Enter “openbts” in this field as well, press Tab to highlight “Continue,” then press Enter to confirm.

The password field for the new user will now appear. Enter a password, press Tab to highlight “Continue,” then press Enter to confirm. A password confirmation field will then appear. Enter the same password, press Tab to highlight “Continue,” then press Enter to confirm.

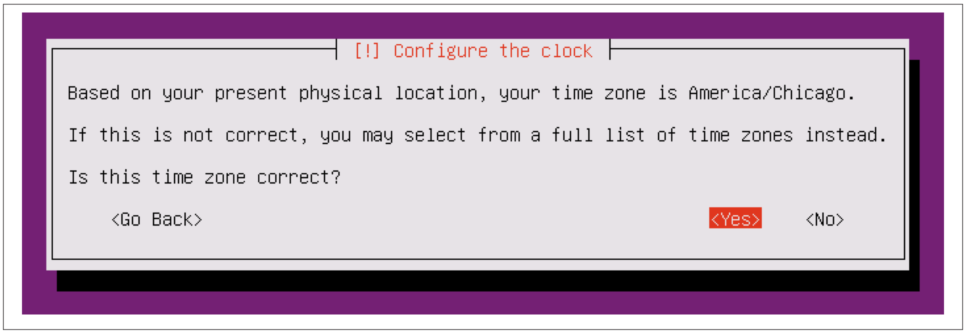
If the password is not complex enough, a screen may appear asking if you’d like to use this weak password. It’s up to you to either make a more complex password or accept it as is. Once the question “Use weak password?” is asked, use the arrow keys to select “Yes” or “No” and press Enter to confirm.

Finally, a screen will appear that asks if you’d like to encrypt the home directory for this user. The OpenBTS suite has not been tested with encrypted user directories so the safe choice here is “No.” Use the arrow keys to select “No” and press Enter to confirm.

Network Configuration and Autodetection

The installer mostly automates the next section, as it automatically configures the network connection by using DHCP. If a DHCP service is not available, you will need to manually enter an IP address for your server as well as your network’s netmask, gateway, and DNS information. This falls outside the scope of this book. Please consult the Ubuntu server documentation online.

After the network hardware and configuration has completed automatically, a prompt will appear with the autodetected time zone as shown in Figure B-3.


```
[!] Configure the clock

Based on your present physical location, your time zone is America/Chicago.

If this is not correct, you may select from a full list of time zones instead.

Is this time zone correct?

    <Go Back>                                          <Yes>     <No>
```

Figure B-3. Time zone autodetection

If the time zone corresponds to your location, use the arrow keys to select “Yes” and press Enter to confirm. If it does not match, select “No” and press Enter to bring up a manual selection menu.

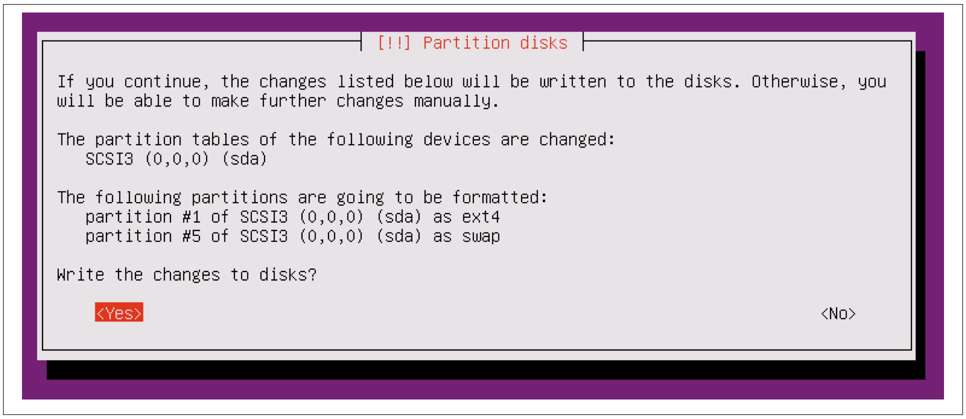
Configuring the Disk

The installer is now nearly ready to start writing files to the disk but it must first be partitioned. The easiest way is to have Ubuntu use the entire disk. If you already have another operating system installed on the disk or would like to customize the partitioning scheme used, you will need to again consult the Ubuntu documentation, as this falls outside the scope of this book.

On the "Partitioning disks" screen, in the "Partitioning method" question, use the arrow keys to highlight "Guided - Use Entire Disk" and press Enter to confirm.

A new screen will appear that lists the available disks. Use the arrow keys to highlight the desired target disk and press Enter to confirm.

Finally, a confirmation screen will appear as shown in Figure B-4.



```
[!!] Partition disks

If you continue, the changes listed below will be written to the disks. Otherwise, you
will be able to make further changes manually.

The partition tables of the following devices are changed:
   SCSI3 (0,0,0) (sda)

The following partitions are going to be formatted:
   partition #1 of SCSI3 (0,0,0) (sda) as ext4
   partition #5 of SCSI3 (0,0,0) (sda) as swap

Write the changes to disks?

     <Yes>                                                                  <No>
```

Figure B-4. Commit changes to disk prompt

Be absolutely sure it is the correct target disk. After this step completes, the disk contents may be irreversibly lost.

For the "Write the changes to disks?" question, use the arrow keys to select "Yes" and press Enter to confirm.

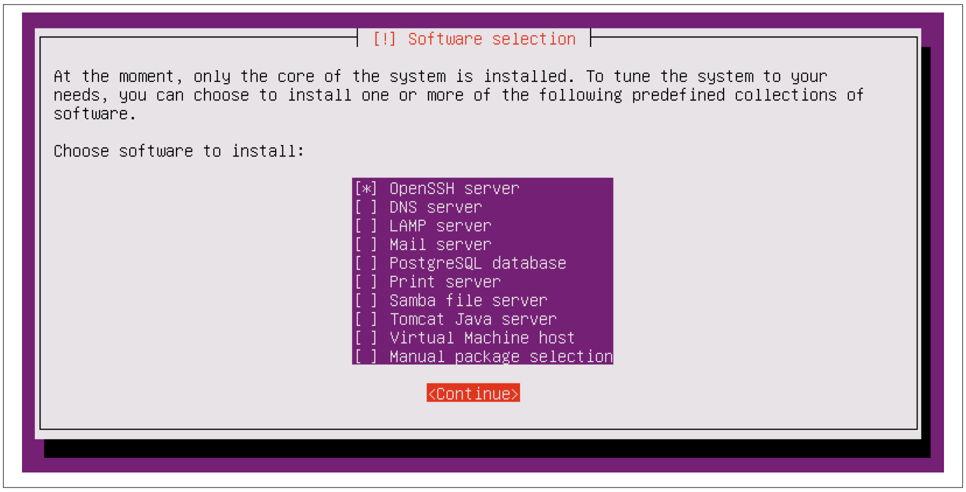
Base Software and Updates

The package manager now needs to be configured. The first screen that appears allows you to specify the HTTP proxy on your network. If you have one, enter it in the "HTTP proxy information" field; otherwise leave it blank. Press Tab to highlight "Continue" and press Enter to confirm.

The next screen titled "Configuring tasksel" allows automatic updates to be activated on the system. To avoid the possibility of a security update breaking your system without your permission, this should be left deactivated. Confirm the default of "No automatic updates" by pressing Enter.

Finally, a screen will appear that allows you to add some base software to the default installation. The only interesting item is "OpenSSH," which installs the Secure Shell service, allowing remote console access to the server.

Use the space bar to activate "OpenSSH," then press Tab to highlight "Continue" and press Enter to confirm, as shown in Figure B-5.



```
[!] Software selection

At the moment, only the core of the system is installed. To tune the system to your
needs, you can choose to install one or more of the following predefined collections of
software.

Choose software to install:

                              [*] OpenSSH server
                              [ ] DNS server
                              [ ] LAMP server
                              [ ] Mail server
                              [ ] PostgreSQL database
                              [ ] Print server
                              [ ] Samba file server
                              [ ] Tomcat Java server
                              [ ] Virtual Machine host
                              [ ] Manual package selection

                                  <Continue>
```

Figure B-5. Adding SSH to the default installation

Finishing the Installation

A small piece of software called GRUB is still needed so that the operating system can be loaded on system boot. As with the partitioning instructions, if you are installing Ubuntu alongside another existing operating system, you will need to consult the Ubuntu documentation. Once the question, "Install the GRUB boot loader to the master boot record?" appears, use the arrow keys to highlight "Yes" and press Enter to confirm.

The final screen will appear confirming that the installation is complete. If this is a physical server, remove the installation CD at this point. Press Enter to confirm and the system will now reboot.

First Login

Once the system has rebooted, log in with the username "openbts" and your password.

SSH

If you do not want to use the Linux Server hardware or virtual machine directly, use SSH access instead. This is the "OpenSSH" service that was installed during the installation steps. It allows you to log in to the machine across a network connection. If you are using OS X or a Linux desktop, the SSH client is built in. Open the Terminal application and execute the following:

```
$ ssh openbts@your-linux-server-ip
```

If you are using Windows, you will need a third-party client. The most popular one is PuTTY. Use the following login details to connect:

- Username: openbts
- Password: *your-chosen-password*
- Host: *your-linux-server-ip*
- Port: 22

You're logged in and now ready to continue configuring the development environment.

APPENDIX C

Capturing Traffic

Two sources can be used to capture activity on an OpenBTS installation. The IP side of the network can be recorded for debugging or analysis, as can the raw GSM and GPRS radio frames. Unified into a single stream, it provides a very powerful research tool for baseband developers, application authors, and network engineers.

IP Traffic

The first source is the raw IP packets being sent between the individual components of OpenBTS. In addition to VoIP signaling and media streams, there are a multitude of other ports exchanging data. A list of these ports, their settings, and their types can be found in Table A-3.

To capture data from these ports, a small utility named `tcpdump` is needed. It can listen to any network interface on your system and display and/or record the traffic to a file. Install it now:

```
$ sudo apt-get install tcpdump
```

The exact usage of tcpdump is outside of the scope of this book so only a few examples will be presented. To learn more about it, use the `man` (manual) command:

```
$ man tcpdump
```

To display (`-s0 -A`) all SIP signaling (`portrange 5060-5069`) on the console from the local loopback interface (`-i lo`), execute the following:

```
$ sudo tcpdump -i lo -n -s0 -A portrange 5060-5069
```

Instead of displaying the traced traffic, you can record it in a pcap (packet capture) formatted file. The Wireshark GUI tool, which is outlined below, uses the pcap format. To record all RTP media (`portrange 16484-16584`) to a file (`-w rtp.pcap`), execute the following:

```
$ sudo tcpdump -i lo -n -s0 -w rtp.pcap portrange 16484-16584
```

When running a multinode system, the traffic will not be present on the loopback interface. It will be physically transmitted and received over a hardware network interface. To specify this interface, use the "-i" flag as above but replace "lo" with the interface name. For example, to display (`-s0 -A`) all peering protocol (`port 16001`) exchanges on the console from the `eth0` interface, execute the following:

```
$ sudo tcpdump -i eth0 -n -s0 -A port 16001
```

GSM Traffic

IP traffic is easy to capture but how about the radio interface? There's a convenient feature for doing this named GSMTAP. It re-encapsulates the GSM and GPRS radio bursts being sent and received over the air into IP packets. These packets then become part of a normal IP trace, captured by tcpdump, as detailed above.

There are three parameters involved in configuring GSMTAP:

`Control.GSMTAP.GSM`
: Enable or disable GSM traces

`Control.GSMTAP.GPRS`
: Enable or disable GPRS traces

`Control.GSMTAP.TargetIP`
: IP destination for GSMTAP data

Assuming all of your components are running from the same machine, `Control.GSMTAP.TargetIP`'s default value of 127.0.0.1 will be fine. It means that the GSMTAP frames will be sent to the local loopback interface for capturing by tcpdump. If your components are running on different machines, you will want to set `Control.GSMTAP.TargetIP` to the IP address of your Central Services machine.

To begin sending GSM radio traffic as IP, the `Control.GSMTAP.GSM` key must be enabled:

```
OpenBTS> config Control.GSMTAP.GSM 1
Control.GSMTAP.GSM changed from "0" to "1"
```

Now, use tcpdump to capture all IP traffic on the loopback interface and store it as a pcap file:

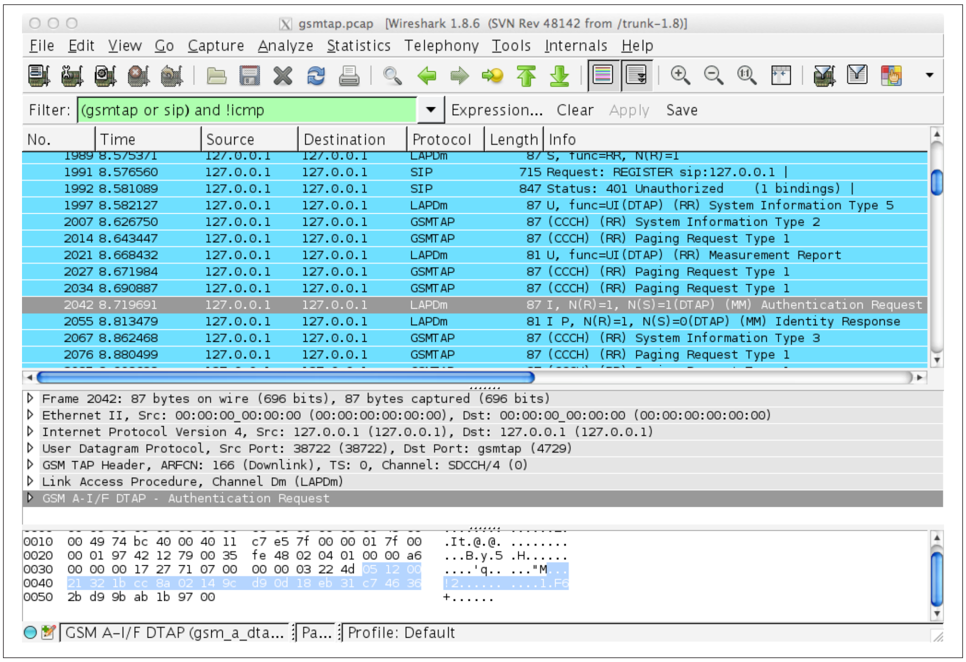
```
$ sudo tcpdump -i lo -n -s0 -w gsmtap.pcap
```

When you're done tracing, tcpdump can be stopped by pressing Ctrl-C.

The resulting file, *gsmtap.pcap*, can be opened in Wireshark. You can either install Wireshark on your test system if it already has a usable desktop, or copy the pcap file to your daily desktop machine using SFTP.

Once open in Wireshark, GSMTAP data can be displayed as any other network traffic. It even has its own display filter: `gsmtap`. In the screenshot below, a pretty common filter combination is used: `(gsmtap or sip) and !icmp`. With this filter combination, GSM and SIP signaling are visible in chronological order, making traces easy to understand.

Figure C-1 is from a GSMTAP trace of a handset successfully registering with a base station. Packets 1991 and 1992, a SIP exchange between OpenBTS and SIPAuthServe, initiate this GSM request for authentication information. The highlighted packet, 2042, is an authentication request sent from the base station to the handset. The following packet, 2055, is the handset's response back to the base station.


gsmtap.pcap [Wireshark 1.8.6 (SVN Rev 48142 from /trunk-1.8)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Expression... Clear Apply Save

Filter: (gsmtap or sip) and !icmp

```
No.   Time      Source     Destination  Protocol  Length Info
 1989 8.575371  127.0.0.1  127.0.0.1    LAPDm      87 S, func=RR, N(R)=1
 1991 8.576560  127.0.0.1  127.0.0.1    SIP       715 Request: REGISTER sip:127.0.0.1 |
 1992 8.581089  127.0.0.1  127.0.0.1    SIP       847 Status: 401 Unauthorized     (1 bindings) |
 1997 8.582127  127.0.0.1  127.0.0.1    LAPDm      87 U, func=UI(DTAP) (RR) System Information Type 5
 2007 8.626750  127.0.0.1  127.0.0.1    GSMTAP     87 (CCCH) (RR) System Information Type 2
 2014 8.643447  127.0.0.1  127.0.0.1    GSMTAP     87 (CCCH) (RR) Paging Request Type 1
 2021 8.668432  127.0.0.1  127.0.0.1    LAPDm      81 U, func=UI(DTAP) (RR) Measurement Report
 2027 8.671984  127.0.0.1  127.0.0.1    GSMTAP     87 (CCCH) (RR) Paging Request Type 1
 2034 8.690887  127.0.0.1  127.0.0.1    GSMTAP     87 (CCCH) (RR) Paging Request Type 1
 2042 8.719691  127.0.0.1  127.0.0.1    LAPDm      87 I, N(R)=1, N(S)=1(DTAP) (MM) Authentication Request
 2055 8.813479  127.0.0.1  127.0.0.1    LAPDm      81 I P, N(R)=1, N(S)=0(DTAP) (MM) Identity Response
 2067 8.862468  127.0.0.1  127.0.0.1    GSMTAP     87 (CCCH) (RR) System Information Type 3
 2076 8.880499  127.0.0.1  127.0.0.1    GSMTAP     87 (CCCH) (RR) Paging Request Type 1
```

```
▷ Frame 2042: 87 bytes on wire (696 bits), 87 bytes captured (696 bits)
▷ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▷ Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
▷ User Datagram Protocol, Src Port: 38722 (38722), Dst Port: gsmtap (4729)
▷ GSM TAP Header, ARFCN: 166 (Downlink), TS: 0, Channel: SDCCH/4 (0)
▷ Link Access Procedure, Channel Dm (LAPDm)
▷ GSM A-I/F DTAP - Authentication Request
```

```
0010  00 49 74 bc 40 00 40 11  c7 e5 7f 00 00 01 7f 00   .It.@.@. ........
0020  00 01 97 42 12 79 00 35  fe 48 02 04 01 00 00 a6   ...B.y.5 .H......
0030  00 00 00 17 27 71 07 00  00 00 03 22 4d 05 12 00   ....'q.. ..."M...
0040  21 32 1b cc 8a 02 14 9c  d9 0d 18 eb 31 c7 46 36   !2...... ....1.F6
0050  2b d9 9b ab 1b 97 00                               +......
```

GSM A-I/F DTAP (gsm_a_dta... | Pa... | Profile: Default

Figure C-1. GSMTAP data in Wireshark

As you can see, GSMTAP combined with tcpdump and Wireshark makes for a very powerful tool.

Glossary

ARFCN (Absolute Radio Frequency Channel Number)
: A pair of uplink and downlink frequencies on which the mobile network is operating. The bandwidth on these frequencies is approximately 270 KHz. A single base station can operate on multiple ARFCNs to increase capacity. Typically pronounced “arfcon.”

BCC (Base Station Color Code)
: A 3-bit value unique among neighboring base stations from the same operator in a given area. Usually assigned from a color map so each neighboring base station has a unique “color” value and no two adjacent base stations have the same color.

BSIC (Base Station Identity Code)
: A 6-bit value composed of two 3-bit pieces: the NCC and BCC. Commonly pronounced as “bee-sick,” it is one of the first things a Mobile Station uses to identify a network and quickly ascertain if it can expect to receive service on it.

BTS (Base Transceiver Station)
: The “access point” in a GSM network to which Mobile Stations can attach. Usually referred to as the base station.

CI (Cell Identity)
: A 16-bit value unique to every single base station in the network.

GSM (Global System for Mobile)
: The standard for mobile network infrastructure used throughout most of the world. Many other standards exist such as CDMA, UMTS, and LTE.

IMEI (International Mobile Equipment Identity)
: A 15-digit number that uniquely identifies the handset hardware. It encodes the manufacturer and model and is analogous to a MAC address. The IMEI is usually only used to detect stolen hardware but can, for example, be used as an identity when making an emergency call, where having a functional SIM is not required.

IMSI (International Mobile Subscriber Identity)
: A 14–15 digit number stored in the SIM that identifies the subscriber. Analogous to a username. It also embeds information about the issuing operator network.

IMSI	**310150123456789**
MCC	**310**150123456789
MNC	310**150**123456789
MSIN	310150**123456789**

LAC (Location Area Code)
: A 16-bit value that identifies the paging area of a base station. In production carrier networks, a single LAC could, for example, be

assigned to an area of roughly 400,000 inhabitants.

MCC (Mobile Country Code)
: A three-digit number that identifies the network's country of operation. The International Telecommunications Union (ITU) assigns this number. A single country can have multiple MCCs. The code 001 is reserved for test networks.

MNC (Mobile Network Code)
: A two- or three-digit number the national regulator assigns that uniquely identifies the carrier. Commonly through acquisitions and mergers, a single carrier will have multiple MNCs. The code 01 is reserved for test networks.

MS (Mobile Station)
: The client that connects to the mobile network. This is usually referred to as a "handset" or "phone" but could also be a smart parking meter, point-of-sale machine, or telemetric sensor—anything that can connect to mobile network infrastructure.

MSIN (Mobile Subscription Identification Number)
: A nine-digit number contained within the IMSI that represents the subscriber's unique account ID within that operator network.

MSISDN (Mobile Subscriber Integrated Services Digital Number)
: This is simply the subscriber's telephone number. Handsets do not know their own MSISDN and the mapping between IMSI and MSISDN exists only in the network. This is how number portability is possible.

NCC (Network Color Code)
: A 3-bit value unique to each operator network in a given area that the national regulator usually assigns.

PSTN (Public Switched Telephone Network)
: The global circuit-switched telephony network which has been in use for nearly 100 years.

RAN (Radio Access Network)
: The base stations in a provider's network, seen as a whole, form the Radio Access Network.

RTP (Real-time Transfer Protocol)
: A media protocol in many VoIP implementations. It encapsulates audio and video streams between endpoints.

SDR (Software defined radio)
: Typically a USB or Ethernet connected computer peripheral that presents a raw radio interface for applications to send and receive signals through, instead of implementing an FM radio in hardware and a separate AM radio in hardware. One must now only have a single SDR and write two applications: one for demodulating FM signals and another for demodulating AM signals.

SIP (Session Initiation Protocol)
: A signaling protocol in many VoIP implementations. It is responsible for establishing calls, describing media, and routing among transport hops.

TMSI (Temporary Mobile Subscriber Identity)
: A 32-bit number the operator network assigns that uniquely identifies a subscriber. Used to support anonymous transactions by exchanging the IMSI for a random TMSI upon registration and then periodically exchanging TMSIs for new TMSIs.

VoIP (Voice over Internet Protocol)
: The method of transmitting speech as packetized data over an IP network.

Index

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

About the Author

After first experimenting with VoIP in 2004, **Michael Iedema** worked on IMS proof-of-concept projects until 2007, when he founded the AskoziaPBX project, an easy-to-use telephony system now being sold and deployed in over 100 countries. In 2011, he moved on to do VoIP-related research and development for Ubiquiti Networks. Michael is currently a Senior Engineer at Range Networks, the creators of OpenBTS.

Colophon

The animal on the cover of *Getting Started with OpenBTS* is a sun conure, or sun parakeet (*Aratinga solstitialis*), a medium-sized, brightly colored parrot. Adult males and females are similarly decked out with golden-yellow plumage and orange underparts and face, with red around the ears. Their tails are olive green with blue tips. Juvenile sun conures' plumage is predominantly green, changing to its distinctive yellow, orange, and red coloration as it matures. They reach sexual maturity at two years of age with a lifespan of 25 to 30 years. Hens lay clutches of 3 to 5 eggs; incubation lasts 23 days.

The sun conure is native to northeastern South America. Though its exact habitat requirements are largely unknown, sightings suggest it lives at the edge of humid forests growing in foothills in the Guiana Shield. Common among other members of the *Aratinga* genus, sun parakeets are social and typically found in groups of up to 30 individuals. They mainly feed on fruit, flowers, berries, and nuts.

They were previously categorized as Least Concern, but recent surveys in southern Guyana and Roraima (a Brazilian state) have shown rarity and possible extirpation. They are endangered and threatened by habitat loss and pet trade trapping. Though they are regularly bred in captivity, the capture of wild sun conures remains a serious threat.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from FILL IN CREDITS. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.