DVD

PRENTICE HALL

The Official ubuntu Server Book

Kyle Rankin
Benjamin Mako Hill

Library of Congress Cataloging-in-Publication Data

Rankin, Kyle.
The official Ubuntu server book / Kyle Rankin and Benjamin Mako Hill.
p. cm.
Includes index.
ISBN 0-13-702118-6 (pbk. : alk. paper)
1. Ubuntu (Electronic resource) 2. Linux. 3. Operating systems (Computers) I. Hill, Benjamin Mako, 1980– II. Title.
QA76.76.O63R3685 2009
005.4'32—dc22 2009021260

ISBN-13: 978-0-13-702118-5
ISBN-10: 0-13-702118-6
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, July 2009

Preface

WELCOME to *The Official Ubuntu Server Book*!

When most people talk about Ubuntu these days, they tend to talk about the Ubuntu Desktop. After all, it's the easy-to-use "just works" approach to the desktop that has made Ubuntu one of the most popular desktop Linux distributions. What has gotten less attention, although even this is starting to change, is Ubuntu Server. It turns out that desktop Linux users aren't the only ones who want their distribution to "just work"—system administrators appreciate that on their servers as well. In Ubuntu Server you will find all of the powerful server infrastructure from the Debian project plus that extra bit of Ubuntu polish, innovation, and focus on ease of use.

About This Book

This book is the result of the collaborative effort of not just the principal authors, but of the Ubuntu Server team itself. As it is the official, authorized book on Ubuntu Server, the focus has been on a server guide based on our collective experience. Beyond that, the goal is to have something to offer to both the beginner system administrator and the battle-hardened senior sysadmin. On the surface it might seem a tough balance to achieve, but in reality both groups ultimately want the same thing: for their servers to work. Now it's true that some administrators revel in doing things the hard way. Some even treat it as a point of pride. The thing is, all of us who have administered servers for years can do and have done things the hard way as well, but ultimately you realize that there's nothing particularly impressive in doing everything by hand—in the end you just have too much to do and any time-saving steps are welcome.

As you will see, most of this book takes a pragmatic approach to server management. Where Ubuntu offers new programs or features to ease administration and save time, you will find them mentioned here. If you are a beginner administrator, you will find that administering an Ubuntu server isn't nearly as difficult as you might think. Experienced administrators, especially those coming from other platforms, will find numerous time-saving tips and programs, as well as where Ubuntu has updated how a service is organized (Apache being a good example); you can treat this book as a map to point you to all of the right directories.

One great thing about Ubuntu as a server is that there are so many great server packages available for it. Of course, this creates a dilemma for us as writers: It's just not possible to feature every available e-mail and IMAP/POP3 server, for instance. In these cases we've tried to pick out programs that are easy to install, configure, and use under Ubuntu as well as highlight programs that are preferred by the authors and server team. While doing that, there's a good chance that your favorite program for X, Y, and Z was left out. It's certainly no slight against any of those programs—we just had to draw the line somewhere.

How the Book Is Organized

Different people read tech books differently. Some people read them cover to cover, and others skip right ahead to the topic they need immediate help with. You will find that the way this book is organized lends itself well to both approaches. The first few chapters lay the foundation so you can install Ubuntu and navigate the system even if it's your first time. After that the chapters focus on particular server topics, from security to monitoring to system rescue.

- Chapter 1—Installation: In the first chapter you will learn how to use the default Ubuntu Server CD to install Ubuntu on a server. This guide will include a complete walk-through of the installation process, from the initial boot screen to partitioning to your first login prompt.
- Chapter 2—Essential System Administration: If you are new to Ubuntu system administration, the amount of learning ahead of you might seem daunting. In this chapter you will find not only a solid foundation of instructions on how to navigate the Linux command

line, but also an introduction to the Ubuntu boot process and the standards behind all of the directories on an Ubuntu system. By the end of the chapter you should have a good basis to continue with the rest of the book.

- Chapter 3—Package Management: This chapter introduces you to packages and the packaging system—the way that Ubuntu handles the installation, removal, and management of software. We provide a solid foundation in what packages do and how they do it before drilling down into the details of how an administrator can manage software the Ubuntu way. In the final pages, we cover the way that administrators can switch from consumers to producers and begin making their own packages.
- Chapter 4—Automated Ubuntu Installs: While you can certainly install Ubuntu step by step from the install CD, that method doesn't work so well when you have tens or hundreds of servers to install. This chapter covers the preseed method for automating Ubuntu installs along with Kickseed—Ubuntu's port of Kickstart. In addition to a description of how to use both of these technologies independently, you will find out it's even better when you use them together.
- Chapter 5—Guide to Common Ubuntu Servers: There is an enormous number of services you can run on an Ubuntu server. In this chapter we highlight some of the more popular servers, from Web to e-mail to file services. If you are a new administrator, you'll find a simple guide on how to install and configure these services for the first time. If you are an experienced administrator coming from another distribution, you'll find this chapter a handy guide to find out how Ubuntu organizes all of the configuration files for your favorite services.
- Chapter 6—Security: Security is an important topic for any administrator. Ubuntu Server already is pretty secure by default, and in this chapter we will highlight some of these mechanisms along with steps you can take to increase your security even further. Some of the security topics include `sudo`, firewall configuration, an introduction to forensics, and even Ubuntu's AppArmor software.
- Chapter 7—Backups: There are two kinds of administrators: those who back up their servers and those who haven't lost valuable data yet. Backup software abounds for Linux as a whole and for Ubuntu

specifically, and in this chapter you will see a few easy-to-set-up approaches to keeping your data secure.

- Chapter 8—Monitoring: Monitoring is one of the most valuable systems an administrator can set up while simultaneously being the most annoying (why do servers always seem to page you in the middle of the night?). In this chapter we will cover some different approaches to monitoring systems both for trending purposes and to alert you to any problems. By the end of the chapter you will no longer lose sleep wondering if a server is up—you'll lose sleep only when it goes down.
- Chapter 9—Virtualization: Virtualization is one of the hot topics in system administration today. With more and more powerful hardware out there, virtualization provides you with a way to squeeze the most efficiency out of your servers. In this chapter we will cover two of the most popular server-based virtualization tools out there: KVM and VMware Server.
- Chapter 10—Fault Tolerance: If a lot is riding on your servers and your downtime is measured in dollars and not minutes, you realize very quickly that your servers need fault tolerance. The fault tolerance chapter covers Ubuntu software RAID, including steps to migrate from one type of RAID to another. Then we will cover how to set up redundant network connections and finish up with a guide to setting up your own Linux cluster.
- Chapter 11—Troubleshooting: No matter how great an administrator you are, eventually something on your servers will fail. Over the years you develop a series of troubleshooting steps you go through whenever you find a problem on your systems. In this chapter we've condensed years of troubleshooting experience into a series of step-by-step guides to walk you through common server and network problems and how to use standard Ubuntu tools and techniques to diagnose them.
- Chapter 12—Rescue and Recovery: We've often said that we've learned more about Linux from fixing a broken system than in any other way. In some environments when a system won't boot, an administrator might just install a new operating system. Under Ubuntu, however, you'll find that most common boot problems also

have a common, easy solution. In this chapter we'll discuss how to use different stages of rescue modes both on Ubuntu and the Ubuntu Server install CD itself to repair your system.

- Chapter 13—Help and Resources: One great thing about Ubuntu is just how many support avenues there are when you need help. Whether it's documentation on the machine itself, guides on the official Ubuntu site, forums, or even professional Canonical support, when you are stuck you aren't alone. In this chapter we cover all of the different ways to get support for your Ubuntu server.

Media with This Book

This book includes two versions of Ubuntu Server—Ubuntu 8.04, the LTS release, as well as the latest Ubuntu Server edition (as of publishing time), 9.04. This way you can take advantage of Canonical's five-year support for LTS releases, or if you take a more conservative approach to your servers, you can use 8.04 LTS. If you want the latest and greatest that Ubuntu Server has to offer, then 9.04 is right at your fingertips.

We have decided to include the 32-bit versions of these releases so that the CDs will work for the largest number of computers. Ubuntu also ships 64-bit versions of Ubuntu Server, so if you have a 64-bit server you might want to take advantage of 64-bit features such as improved performance and memory management on your system. To get a 64-bit Ubuntu Server CD, just go to http://ubuntu.com and either download the CD image or request a copy to be sent to you. No matter which Ubuntu Server CD you pick, it's relatively easy to use the CDs. Just insert the version you want to install into your computer and boot from the CD-ROM. When the CD boots, you will see a number of options on the screen, but to install Ubuntu Server just select Install Ubuntu Server. The installer that launches will ask some fairly straightforward questions common to most install discs, and if you get stuck just turn to Chapter 1 for a more in-depth walk-through of the install process.

Introduction

THIS INTRODUCTION GIVES AN overview of Ubuntu and Ubuntu Server. After a quick welcome, it includes a brief history of free software, open source, and GNU/Linux and of the Ubuntu project itself with a focus on some of the major players on the Ubuntu scene. This introduction ends where the rest of this book will continue: with a history of the Ubuntu Server project and an overview of that project's goals and accomplishments.

Welcome to Ubuntu Server

In the just over four years of its life, Ubuntu has become one of the most popular GNU/Linux-based operating systems. In the process, however, public perception has been disproportionately focused on Ubuntu's role as a desktop-based operating system. While all popularity is certainly welcome for those of us involved in the project, this success has, at times, overshadowed the rock-solid server operating system that Ubuntu has been constructed to be. For those of us who have helped build out Ubuntu's server-specific features and who use it daily, this is both unfortunate and undeserved. Designed and used as a server since day one, Ubuntu has supported a server team that was one of the first active teams in the Ubuntu community and has been one of the most successful. Although perceptions have changed in large part, many prospective users—and even some current Ubuntu users—often continue to think of Ubuntu as something for desktops.

Perhaps it is just that people are so surprised at the usability of Ubuntu on the desktop—especially in the early days when expectations for desktop GNU/Linux distributions were low—that the public focus naturally has drifted away from Ubuntu's server offering. Lots of other GNU/Linux distributions run great on servers, but a solid desktop experience continues

to be surprising to many users. As a result, when people talk about Ubuntu, they often tend to talk about desktops. Perhaps, on the other hand, people just figured that such a well-polished desktop must have come at the cost of the server-oriented features and support. Of course, no such sacrifices were made.

To a large extent, times have changed. The Ubuntu Server team has continued its tireless work both to improve the experience for server users of Ubuntu and to help promote Ubuntu as a server solution. Documentation, testimonials, certification of server-based software, support contracts from a variety of sources, training courses, and more have all contributed to remaking Ubuntu into a powerful player on the server. Although its desktop credentials have not been diminished, Ubuntu's server chops are increasingly difficult to overlook. Over the past two years, Ubuntu has begun to become a major player in the GNU/Linux server market.

More than anything else, testimonials have spread and the small group of early Ubuntu Server users has spread the word. More and more people choose Ubuntu for their servers every day. In fact, this book is simply the latest striking example of just how far Ubuntu on servers has come. Not only do people now know that Ubuntu runs on a server, they know it runs well. This book is publishable only because there is a market for it. That market is made up of people who have heard good things about Ubuntu on the server and who are getting ready to take the plunge themselves. Welcome. We hope we can help make the process easier. We've come a long way, and we're still only just beginning.

Free Software, Open Source, and Linux

A history of Ubuntu Server must, in large part, be a history of Ubuntu itself. A history of Ubuntu must, in large part, be a history of the free software movement and of the Linux kernel. While thousands of individuals have contributed in some form to Ubuntu, the project has succeeded only through the contributions of many thousands more who have indirectly laid the technical, social, and economic groundwork for Ubuntu's success. While introductions to free software, open source, and GNU/Linux can be found in many other places, no introduction to Ubuntu is complete with-

out a brief discussion of these concepts and the people and history behind them. It is around these concepts and within these communities that Ubuntu was motivated and born. Ultimately, it is through these ideas that it is sustained.

Free Software and GNU

In a series of events that have almost become legend through constant repetition, Richard M. Stallman created the concept of free software in 1983. Stallman grew up with computers in the 1960s and 1970s, when computer users purchased very large and extremely expensive mainframe computers, which were then shared among large numbers of programmers. Software was, for the most part, seen as an add-on to the hardware, and every user had the ability and the right to modify or rewrite the software on his or her computer and to freely share this software. As computers became cheaper and more numerous in the late 1970s, producers of software began to see value in the software itself. Producers of computers began to argue that their software was copyrightable and a form of intellectual property much like a music recording, a film, or a book's text. They began to distribute their software under licenses and in forms that restricted its users' abilities to use, redistribute, or modify the code. By the early 1980s, restrictive software licenses had become the norm.

Stallman, then a programmer at MIT's Artificial Intelligence Laboratory, became increasingly concerned with what he saw as a dangerous loss of the freedoms that software users and developers had up until that point enjoyed. He was concerned with computer users' ability to be good neighbors and members of what he thought was an ethical and efficient computer-user community. To fight against this negative tide, Stallman articulated a vision for a community that developed liberated code—in his words, "free software." He defined free software as software that had the following four characteristics—labeled as freedoms 0 through 3 instead of 1 through 4 as a computer programmer's joke:

- The freedom to run the program for any purpose (freedom 0)
- The freedom to study how the program works and adapt it to your needs (freedom 1)

- The freedom to redistribute copies so you can help your neighbor (freedom 2)
- The freedom to improve the program and release your improvements to the public so that the whole community benefits (freedom 3)

Access to source code—the human-readable and modifiable blueprints of any piece of software that can be distinguished from the computer-readable version of the code that most software is distributed as—is a prerequisite to freedoms 1 and 3. In addition to releasing this definition of free software, Stallman began a project with the goal of creating a completely free OS to replace the then-popular UNIX. In 1984, Stallman announced this project and called it GNU—another joke in the form of a recursive acronym for "GNU's Not UNIX."

Linux

By the early 1990s, Stallman and a collection of other programmers working on GNU had developed a near-complete OS that could be freely shared. They were, however, missing a final essential piece in the form of a kernel—a complex system command processor that lies at the center of any OS. In 1991, Linus Torvalds wrote an early version of just such a kernel, released it under a free license, and called it Linux. Linus's kernel was paired with the GNU project's development tools and OS and with the graphical windowing system called X. With this pairing, a completely free OS was born—free both in terms of price and in Stallman's terms of freedom.

All systems referred to as Linux today are, in fact, built on the work of this collaboration. Technically, the term Linux refers only to the kernel. Many programmers and contributors to GNU, including Stallman, argue emphatically that the full OS should be referred to as GNU/Linux in order to give credit not only to Linux but also to the GNU project and to highlight GNU's goals of spreading software freedom—goals not necessarily shared by Linus Torvalds. Many others find this name cumbersome and prefer calling the system simply Linux. Yet others, such as those working on the Ubuntu project, attempt to avoid the controversy altogether by referring to GNU/Linux only by using their own project's name.

Open Source

Disagreements over labeling did not end with discussions about the naming of the combination of GNU and Linux. In fact, as the list of contributors to GNU and Linux grew, a vibrant world of new free software projects sprouted up, facilitated in part by growing access to the Internet. As this community grew and diversified, a number of people began to notice an unintentional side effect of Stallman's free software. Because free software was built in an open way, *anyone* could contribute to software by looking through the code, finding bugs, and fixing them. Because software ended up being examined by larger numbers of programmers, free software was higher in quality, performed better, and offered more features than similar software developed through proprietary development mechanisms. In many situations, the development model behind free software led to software that was *inherently better* than proprietary alternatives.

As the computer and information technology industry began to move into the dot-com boom, one group of free software developers and leaders, spearheaded by two free software developers and advocates—Eric S. Raymond and Bruce Perens—saw the important business proposition offered by a model that could harness volunteer labor or interbusiness collaboration and create intrinsically better software. However, they worried that the term *free software* was problematic for at least two reasons. First, it was highly ambiguous—the English word *free* means both gratis, or at no cost (e.g., as in "free beer") and liberated in the sense of freedom (e.g., as in "free speech"). Second, there was a feeling, articulated most famously by Raymond, that all this talk of freedom was scaring off the very business executives and decision makers whom the free software movement needed to impress in order to succeed.

To tackle both of these problems, this group coined a new phrase—*open source*—and created a new organization called the Open Source Initiative. The group set at its core a definition of open source software that overlapped completely and exclusively both with Stallman's four-part definition of free software and with other community definitions that were also based on Stallman's.

One useful way to understand the split between the free software and open source movements is to think of it as the opposite of a schism. In religious

schisms, churches separate and do not work or worship together because of relatively small differences in belief, interpretation, or motivation. For example, most contemporary forms of Protestant Christianity agree on *almost* everything but have separated over some small but irreconcilable difference. However, in the case of the free software and open source movements, the two groups have fundamental disagreements about their motivation and beliefs. One group is focused on freedom, while the other is focused on pragmatics. Free software is most accurately described as a social movement, whereas open source is a development methodology. However, the two groups have no trouble working on projects hand in hand.

In terms of the motivations and goals, open source and free software diverge greatly. Yet in terms of the software, the projects, and the licenses they use, they are completely synonymous. While people who identify with either group see the two movements as being at odds, the Ubuntu project sees no conflict between the two ideologies. People in the Ubuntu project identify with either group and often with both. In this book, we may switch back and forth between the terms as different projects and people in Ubuntu identify more strongly with one term or the other. For the purposes of this book, though, either term should be read as implying the other unless it is stated otherwise.

A Brief History of the Ubuntu Project

A history of Ubuntu, born in April 2004, may seem premature. However, the last five years have been full ones for Ubuntu. With its explosive growth, it is difficult even for those involved most closely with the project to track and record some of the high points. Importantly, there are some key figures whose own history must be given for a full understanding of Ubuntu. This brief summary outlines the high points of Ubuntu's history to date and gives the necessary background knowledge to understand where Ubuntu comes from.

Mark Shuttleworth

No history of Ubuntu can call itself complete without a history of Mark Shuttleworth. Shuttleworth is, undeniably, the most visible and important person in Ubuntu. More important from the point of view of history,

Shuttleworth is also the originator and initiator of the project—he made the snowball that would eventually roll on and grow to become the Ubuntu project.

Shuttleworth was born in 1973 in Welkom, Free State, in South Africa. He attended Diocesan College and obtained a business science degree in finance and information systems at the University of Cape Town. During this period, he was an avid computer hobbyist and became involved with the free and open source software community. He was at least marginally involved in both the Apache project and the Debian project and was the first person to upload the Apache Web server, perhaps the single most important piece of server software on GNU/Linux platforms, into the Debian project's archives.

Seeing an opportunity in the early days of the Web, Shuttleworth founded a certificate authority and Internet security company called Thawte in his garage. Over the course of several years, he built Thawte into the second-largest certificate authority on the Internet, trailing only the security behemoth VeriSign. Throughout this period, Thawte's products and services were built and served almost entirely from free and open source software. In December 1999, Shuttleworth sold Thawte to VeriSign for an undisclosed amount that reached into the hundreds of millions in U.S. dollars.

With his fortune made at a young age, Shuttleworth might have enjoyed a life of leisure—and probably considered it. Instead, he decided to pursue his lifelong dream of space travel. After paying approximately $20 million to the Russian space program and devoting nearly a year to preparation, including learning Russian and spending seven months training in Star City, Russia, Shuttleworth realized his dream as a civilian cosmonaut aboard the Russian Soyuz TM-34 mission. On this mission, Shuttleworth spent two days on the Soyuz rocket and eight days on the International Space Station, where he participated in experiments related to AIDS and genome research. In early May 2002, Shuttleworth returned to Earth.

In addition to space exploration and a less-impressive jaunt to Antarctica, Shuttleworth played an active role as both a philanthropist and a venture capitalist. In 2001, he founded the Shuttleworth Foundation (TSF), a non-profit organization based in South Africa. The foundation was chartered

to fund, develop, and drive social innovation in the field of education. Of course, the means by which TSF attempts to achieve these goals frequently involves free software. Through these projects, the organization has been one of the most visible proponents of free and open source software in South Africa and even the world. In the venture capital area, Shuttleworth worked to foster research, development, and entrepreneurship in South Africa with strategic injections of cash into start-ups through a new venture capital firm called HBD, an acronym for "Here Be Dragons." During this period, Shuttleworth was busy brainstorming his next big project—the project that would eventually become Ubuntu.

The Warthogs

There has been no lack of projects attempting to wrap GNU, Linux, and other pieces of free and open source software into a neat, workable, and user-friendly package. Mark Shuttleworth, like many other people, believed that the philosophical and pragmatic benefits offered by free software put it on a course for widespread success. That said, none of the offerings were particularly impressive. Something was missing from all of them. Shuttleworth saw this as an opportunity. If someone could build the great free software distribution that helped push GNU/Linux into the mainstream, he or she would come to occupy a position of strategic importance.

Shuttleworth, like many other technically inclined people, was a huge fan of the Debian project (discussed in depth later). However, many things about Debian did not fit with Shuttleworth's vision of an ideal OS. For a period of time, Shuttleworth considered the possibility of running for Debian project leader as a means of reforming the Debian project from within. With time, though, it became clear that the best way to bring GNU/Linux into the mainstream would not be from within the Debian project—which in many situations had very good reasons for being the way it was. Instead, Shuttleworth would create a new project that worked in symbiosis with Debian to build a new, better GNU/Linux system.

To kick off this project, Shuttleworth invited a dozen or so free and open source software developers he knew and respected to his flat in London in April 2004. It was in this meeting (alluded to in the first paragraphs of this introduction) that the groundwork for the Ubuntu project was laid. By

that point, many of those involved were excited about the possibility of the project. During this meeting, the members of the team—which would in time grow into the core Ubuntu team—brainstormed a large list of the things that they would want to see in their ideal OS. The list is now a familiar list of features to most Ubuntu users. Many of these traits are covered in more depth later in this chapter. The group wanted

- Predictable and frequent release cycles
- A strong focus on localization and accessibility
- A strong focus on ease of use and user-friendliness on the desktop
- A strong focus on Python as the single programming language through which the entire system could be built and expanded
- A community-driven approach that worked with existing free software projects and a method by which the groups could give back as they went along—not just at the time of release
- A new set of tools designed around the process of building distributions that allowed developers to work within an ecosystem of different projects and that allowed users to give back in whatever way they could

There was consensus among the group that actions speak louder than words, so there were no public announcements or press releases. Instead, the group set a deadline for itself—six short months in the future. Shuttleworth agreed to finance the work and pay the developers full-time salaries to work on the project. After six months, they would both announce their project and reveal the first product of their work. They made a list of goals they wanted to achieve by the deadline, and the individuals present took on tasks. Collectively, they called themselves the Warthogs.

What Does *Ubuntu* Mean?

At this point, the Warthogs had a great team, a set of goals, and a decent idea of how to achieve most of them. The team did not, on the other hand, have a name for the project. Shuttleworth argued strongly that they should call the project Ubuntu.

Ubuntu is a concept and a term from several South African languages, including Zulu and Xhosa. It refers to a South African ideology or ethic that, while difficult to express in English, might roughly be translated as "humanity toward others," or "I am because we are." Others have described ubuntu as "the belief in a universal bond of sharing that connects all humanity." The famous South African human rights champion Archbishop Desmond Tutu explained ubuntu in this way:

> A person with ubuntu is open and available to others, affirming of others, does not feel threatened that others are able and good, for he or she has a proper self-assurance that comes from knowing that he or she belongs in a greater whole and is diminished when others are humiliated or diminished, when others are tortured or oppressed.

Ubuntu played an important role as a founding principle in postapartheid South Africa and remains a concept familiar to most South Africans today.

Shuttleworth liked the term *Ubuntu* as a name for the new project for several reasons. First, it is a South African concept. While the majority of the people who work on Ubuntu are not from South Africa, the roots of the project are, and Shuttleworth wanted to choose a name that represented this. Second, the project emphasizes the definition of individuality in terms of relationships with others and provides a profound type of community and sharing—exactly the attitudes of sharing, community, and collaboration that are at the core of free software. The term represented the side of free software that the team wanted to share with the world. Third, the idea of personal relationships built on mutual respect and connections describes the fundamental ground rules for the highly functional community that the Ubuntu team wanted to build. *Ubuntu* was a term that encapsulated where the project came from, where the project was going, and how the project planned to get there. The name was perfect. It stuck.

Creating Canonical

In order to pay developers to work on Ubuntu full-time, Shuttleworth needed a company to employ them. He wanted to pick some of the best people for the jobs from within the global free software and open source communities. These communities, inconveniently for Shuttleworth, know no national and geographic boundaries. Rather than move everyone to a

single locale and office, Shuttleworth made the decision to employ these developers through a virtual company. While this had obvious drawbacks in the form of high-latency and low-bandwidth connections, different time zones, and much more, it also introduced some major benefits in the particular context of the project. On one hand, the distributed nature of employees meant that the new company could hire individuals without requiring them to pack up their lives and move to a new country. More important, it meant that *everyone* in the company was dependent on IRC, mailing lists, and online communication mechanisms to do their work. This unintentionally and automatically solved the water-cooler problem that plagued many other corporately funded free software projects—namely, that developers would casually speak about their work in person and cut the community and anyone else who didn't work in the office out of the conversation completely. For the first year, the closest thing that Canonical had to an office was Shuttleworth's flat in London. While the company has grown and now has several offices around the world, it remains distributed, and a large number of the engineers work from home. The group remains highly dependent on Internet collaboration.

With time, the company was named Canonical. The name was a nod to the project's optimistic goals of becoming the canonical place for services and support for free and open source software and for Ubuntu in particular. *Canonical*, of course, refers to something that is accepted as authoritative. It is a common word in the computer programmer lexicon. It's important to note that being canonical is like being standard; it is not coercive. Unlike holding a monopoly, becoming the canonical location for something implies a similar sort of success—but *never* one that cannot be undone and never one that is exclusive. Other companies will support Ubuntu and build operating systems based on it, but as long as Canonical is doing a good job, its role will remain central.

The Ubuntu Community

By now you may have noticed a theme that permeates the Ubuntu project on several levels. The history of free software and open source is one of a profoundly effective *community*. Similarly, in building a GNU/Linux distribution, the Ubuntu community has tried to focus on an ecosystem model—an organization of organizations—in other words, a community.

Even the definition of the term *ubuntu* is one that revolves around people interacting in a community.

It comes as no surprise, then, that an "internal" community plays heavily into the way that the Ubuntu distribution is created. While the Ubuntu 4.10 version (Warty Warthog) was primarily built by a small number of people, Ubuntu achieved widespread success only through contributions by a much larger group that included programmers, documentation writers, volunteer support staff, and users. While Canonical employs a core group of several dozen active contributors to Ubuntu, the distribution has, from day one, encouraged and incorporated contributions from *anyone* in the community, and rewards and recognizes contributions by all. Rather than taking center stage, paid contributors are *not* employed by the Ubuntu project—instead they are employed by Canonical, Ltd. These employees are treated simply as another set of community members. They must apply for membership in the Ubuntu community and have their contributions recognized in the same way as anyone else. All non-business-related communication about the Ubuntu project occurs in public and in the community. Volunteer community members occupy a majority of the seats on the two most important governing boards of the Ubuntu project: the Technical Board, which oversees all technical matters, and the Community Council, which approves new Ubuntu members and resolves disputes. Seats on both boards are approved by the relevant community groups, developers for the Technical Board, and Ubuntu members for the Community Council.

In order to harness and encourage the contributions of its community, Ubuntu has striven to balance the important role that Canonical plays with the value of empowering individuals in the community. The Ubuntu project is based on a fundamental belief that great software is built, supported, and maintained only in a strong relationship with the individuals who use the software. In this way, by fostering and supporting a vibrant community, Ubuntu can achieve much more than it could through paid development alone. The people on the project believe that while the contributions of Canonical and Mark Shuttleworth have provided an important catalyst for the processes that have built Ubuntu, it is the community that has brought the distribution its success to date. The project members believe that it is only through increasing reliance on the community that

the project's success will continue to grow. The Ubuntu community won't outspend the proprietary software industry, but it is very much more than Microsoft and its allies can afford.

Finally, it is worth noting that, while this book is official, neither of the authors is a Canonical employee. This book, like much of the rest of Ubuntu, is purely a product of the project's community.

Ubuntu Promises and Goals

So far, this introduction has been about the prehistory, history, and context of the Ubuntu project. After this chapter, the book focuses on the distribution itself. Before proceeding, it's important to understand the goals that motivated the project.

Philosophical Goals

The most important goals of the Ubuntu project are philosophical in nature. The Ubuntu project lays out its philosophy in a series of documents on its Web site. In the most central of these documents, the team summarizes the charter and the major philosophical goals and underpinnings:

> Ubuntu is a community-driven project to create an operating system and a full set of applications using free and Open Source software. At the core of the Ubuntu Philosophy of Software Freedom are these core philosophical ideals:
>
> 1. Every computer user should have the freedom to run, copy, distribute, study, share, change, and improve their software for any purpose without paying licensing fees.
> 2. Every computer user should be able to use their software in the language of their choice.
> 3. Every computer user should be given every opportunity to use software, even if they work under a disability.

The first item should be familiar by now. It is merely a recapitulation of Stallman's free software definition quoted earlier in the section on free software history. In it, the Ubuntu project makes explicit its goal that every user of software should have the freedoms required by free software. This

is important for a number of reasons. First, it offers users all of the practical benefits of software that runs better, faster, and more flexibly. More important, it gives every user the capability to transcend his or her role as a user and a consumer of software. Ubuntu wants software to be empowering and to work in the ways that users want it to work. Ubuntu wants all users to have the ability to make sure it works for them. To do this, software *must* be free, so Ubuntu makes this a requirement and a philosophical promise.

Of course, the core goals of Ubuntu do not end with the free software definition. Instead, the project articulates two new, but equally important, goals. The first of these, that all computer users should be able to use their computers in their chosen languages, is a nod to the fact that the majority of the world's population does not speak English while the vast majority of software interacts only in that language. To be useful, source code comments, programming languages, documentation, and the texts and menus in computer programs must be written in *some* language. Arguably, the world's most international language is a reasonably good choice. However, there is no language that everyone speaks, and English is not useful to the majority of the world's population that does not speak it. A computer can be a great tool for empowerment and education, but only if the user can understand the words in the computer's interface. As a result, Ubuntu believes that it is the project's—and community's—responsibility to ensure that *every* user can easily use Ubuntu to read and write in the language with which he or she is most comfortable.

Finally, just as no person should be blocked from using a computer simply because he or she does not know a particular language, no user should be blocked from using a computer because of a disability. Ubuntu must be accessible to users with motor disabilities, vision disabilities, and hearing disabilities. It should provide input and output in a variety of forms to account for each of these situations and for others. A significant percentage of the world's most intelligent and creative individuals has disabilities. Ubuntu's impact should not be limited to any subset of the world when it can be fully inclusive. More important, Ubuntu should be able to harness the ability of these individuals as community members to build a better and more effective community.

Conduct Goals and Code of Conduct

If Ubuntu's philosophical commitments describe the *why* of the Ubuntu project, the Code of Conduct (CoC) describes Ubuntu's *how*. Ubuntu's CoC is, arguably, the most important document in the day-to-day operation of the Ubuntu community and sets the ground rules for work and cooperation within the project. Explicit agreement to the document is the only criterion for becoming an officially recognized Ubuntu activist—an Ubuntero—and is an essential step toward membership in the project.

The CoC covers "behavior as a member of the Ubuntu Community, in any forum, mailing list, wiki, Web site, IRC channel, install-fest, public meeting, or private correspondence." The CoC goes into some degree of depth on a series of points that fall under the following headings:

- Be considerate.
- Be respectful.
- Be collaborative.
- When you disagree, consult others.
- When you are unsure, ask for help.
- Step down considerately.

Many of these headings seem like common sense or common courtesy to many, and that is by design. Nothing in the CoC is controversial or radical, and it was never designed to be.

More difficult is that nothing is easy to enforce or decide because acting considerately, respectfully, and collaboratively is often very subjective. There is room for honest disagreements and hurt feelings. These are accepted shortcomings. The CoC was not designed to be a law with explicit prohibitions on phrases, language, or actions. Instead, it aims to provide a constitution and a reminder that considerate and respectful discussion is *essential* to the health and vitality of the project. In situations where there is a serious disagreement on whether a community member has violated or is violating the code, the Community Council is available to arbitrate disputes and decide what action, if any, is appropriate.

Nobody involved in the Ubuntu project, including Mark Shuttleworth and the other members of the Community Council, is above the CoC. The CoC is never optional and never waived. In fact, the Ubuntu community recently created a Leadership Code of Conduct (LCoC), which extends and expands on the CoC and describes additional requirements and expectations for those in leadership positions in the community. Of course, in no way was either code designed to eliminate conflict or disagreement. Arguments are at least as common in Ubuntu as they are in other projects and online communities. However, there is a common understanding within the project that arguments should happen in an environment of collaboration and mutual respect. This allows for *better* arguments with *better* results—and with less hurt feelings and fewer bruised egos.

While they are sometimes incorrectly used as such, the CoC and LCoC are not sticks to be wielded against an opponent in an argument. Instead, they are useful points of reference upon which consensus can be assumed within the Ubuntu community. Frequently, if a group in the community feels a member is acting in a way that is out of line with the code, the group will gently remind the community member, often privately, that the CoC is in effect. In almost all situations, this is enough to avoid any further action or conflict. Very few CoC violations are ever brought before the Community Council.

Technical Goals

While a respectful community and adherence to a set of philosophical goals provide an important frame in which the Ubuntu project works, Ubuntu is, at the end of the day, a technical project. As a result, it only makes sense that in addition to philosophical goals and a project constitution, Ubuntu also has a set of technical goals.

The first technical goal of the project, and perhaps the most important one, is the coordination of regular and predictable releases—something particularly important to server users. In April 2004, at the Warthogs meeting, the project set a goal for its initial proof-of-concept release six months out. In part due to the resounding success of that project, and in larger part due to the GNOME release schedule, the team has stuck to a regular and predictable six-month release cycle and has only once chosen

to extend the release schedule by six weeks and only after obtaining community consensus on the decision. The team then doubled its efforts and made the next release in a mere four and a half months, putting its release schedule back on track. Frequent releases are important because users can then use the latest and greatest free software available—something that is essential in a development environment as vibrant and rapidly changing and improving as the free software community. Predictable releases are important, especially to businesses, because predictability means that they can organize their business plans around Ubuntu. Through consistent releases, Ubuntu can provide a platform upon which businesses and derivative distributions can rely to grow and build.

While releasing frequently and reliably is important, the released software must then be supported. Ubuntu, like all distributions, must deal with the fact that *all* software has bugs. Most bugs are minor, but fixing them may introduce even worse issues. Therefore, fixing bugs after a release must be done carefully or not at all. The Ubuntu project engages in major changes, including bug fixes, *between* releases only when the changes can be extensively tested. However, some bugs risk the loss of users' information or pose a serious security vulnerability. These bugs are fixed immediately and made available as updates for the released distribution. The Ubuntu community works hard to find and minimize all types of bugs before releases and is largely successful in squashing the worst. However, because there is always the possibility that more of these bugs will be found, Ubuntu commits to supporting *every* release for 18 months after it is released. In the case of Ubuntu 6.06 LTS (Dapper Drake), released in 2006, the project went well beyond even this and committed to support the release for three full years on desktop computers and for five years in a server configuration (LTS stands for LongTerm Support). This proved so popular with businesses, institutions, and the users of Ubuntu servers that Ubuntu 8.04 (Hardy Heron) was named as Ubuntu's second LTS release with similar three- and five-year desktop and server extended support commitments. These five-year support commitments are specifically designed for server users and make Ubuntu a much more attractive option for an important class of server users.

This bipartite approach to servers and desktops implies the third major technical commitment of the Ubuntu project and, in a sense, the most

important for this book: support for both servers and desktop computers in separate but equally emphasized modes. While Ubuntu continues to be more well known, and perhaps more popular, in desktop configurations, there exist teams of Ubuntu developers focused on both server and desktop users. The Ubuntu project believes that both desktops and servers are essential and provides installation methods on every CD for both types of systems. Ubuntu provides tested and supported software appropriate to the most common actions in both environments and documentation for each. LTS releases in particular mark an important step toward catering to users on the server.

Finally, the Ubuntu project is committed to making it as easy as possible for users to transcend their roles as consumers and users of software and to take advantage of each of the freedoms central to the Ubuntu philosophy. As a result, Ubuntu has tried to focus its development around the use and promotion of a single programming language, Python. The project has worked to ensure that Python is widely used throughout the system. By ensuring that many applications and many of the "guts" of the system are written in or extensible in Python, Ubuntu is working to ensure that users need to learn only one language in order to take advantage of, automate, and tweak many parts of their systems.

Canonical and the Ubuntu Foundation

While Ubuntu is driven by a community, several groups play an important role in its structure and organization. Foremost among these are Canonical, Ltd., a for-profit company introduced as part of the Ubuntu history description, and the Ubuntu Foundation, which is introduced later in this section.

Canonical, Ltd.

As mentioned earlier, Canonical, Ltd., is a company founded by Mark Shuttleworth with the primary goal of developing and supporting the Ubuntu distribution. Many of the core developers on Ubuntu—although no longer a majority of them—work full-time or part-time under contract for Canonical, Ltd. This funding by Canonical allows Ubuntu to make the type of support commitments that it does. Ubuntu can claim that it will

release in six months because releasing, in one form or another, is something that the paid workers at Canonical can ensure. As an all-volunteer organization, Debian suffered from an inability to set and meet deadlines—volunteers become busy or have other deadlines in their paying jobs that take precedence. By offering paying jobs to a subset of developers, Canonical can set support and release deadlines and ensure that they are met.

In this way, Canonical ensures that Ubuntu's bottom-line commitments are kept. Of course, Canonical does not fund all Ubuntu work, nor could it. Canonical can release a distribution every six months, but that distribution will be made *much* better and more usable through contributions from the community of users. Most features, most new pieces of software, almost all translations, almost all documentation, and much more are created outside of Canonical. Instead, Canonical ensures that deadlines are met and that the essential work, regardless of whether it's fun, gets done.

Canonical, Ltd., was incorporated on the Isle of Man—a tiny island nation between Wales and Ireland that is mostly known as a haven for international businesses. Since Canonical's staff is sprinkled across the globe and no proper office is necessary, the Isle of Man seemed as good a place as any for the company to hang its sign.

Canonical's Service and Support

While it is surprising to many users, fewer than half of Canonical's employees work on the Ubuntu project. The rest of the employees fall into several categories: business development, support and administration, and development on the Bazaar and Launchpad projects.

Individuals involved in business development help create strategic deals and certification programs with other companies—primarily around Ubuntu. In large part, these are things that the community is either ill suited for or uninterested in as a whole. One example of business development work is the process of working with companies to ensure that their software (usually proprietary) is built and certified to run on Ubuntu. For example, Canonical worked with IBM to ensure that its popular DB2 database would run on Ubuntu and, when this was achieved, worked to have

Ubuntu certified as a platform that would run DB2. Similarly, Canonical worked with Dell to ensure that Ubuntu could be installed and supported on Dell laptops as an option for its customers. A third example is the production of this book, which, published by Pearson Education's Prentice Hall imprint, was a product of work with Canonical.

Canonical also plays an important support role in the Ubuntu project in three ways. First, Canonical supports the development of the Ubuntu project. For example, Canonical system administrators ensure that the servers that support development and distribution of Ubuntu are running. Second, Canonical helps Ubuntu users and businesses directly by offering phone and e-mail support. Additionally, Canonical has helped build a large commercial Ubuntu support operation by arranging for support contracts with larger companies and organizations. This support is over and above the free (i.e., gratis) support offered by the community—this commercial support is offered at a fee and is either part of a longer-term flat-fee support contract or is pay-per-instance. By offering commercial support for Ubuntu in a variety of ways, Canonical aims to make a business for itself and to help make Ubuntu a more palatable option for the businesses, large and small, that are looking for an enterprise or enterprise-class GNU/Linux product with support contracts like those offered by other commercial GNU/Linux distributions.

Finally, Ubuntu supports other support organizations. Canonical does not seek or try to enforce a monopoly on Ubuntu support; it proudly lists *hundreds* of other organizations offering support for Ubuntu on the Ubuntu Web pages. Instead, Canonical offers what is called second-tier support to these organizations. Because Canonical employs many of the core Ubuntu developers, the company is very well suited to taking action on many of the tougher problems that these support organizations may run into. With its concentrated expertise, Canonical can offer this type of backup, or secondary, support to these organizations.

The Ubuntu Foundation

Finally, in addition to Canonical and the full Ubuntu community, the Ubuntu project is supported by the Ubuntu Foundation, which was announced by Shuttleworth with an initial funding commitment of

$10 million. The foundation, like Canonical, is based on the Isle of Man. The organization is advised by the Ubuntu Community Council.

Unlike Canonical, the foundation does not play an active role in the day-to-day life of Ubuntu. At the moment, the foundation is little more than a pile of money that exists to endow and ensure Ubuntu's future. Because Canonical is a young company, many companies and individuals find it difficult to trust that Canonical will be able to provide support for Ubuntu in the time frames (e.g., three to five years) that it claims it will be able to. The Ubuntu Foundation exists to allay those fears.

If something bad were to happen to Shuttleworth or to Canonical that caused either to be unable to support Ubuntu development and maintain the distribution, the Ubuntu Foundation exists to carry on many of Canonical's core activities well into the future. Through the existence of the foundation, the Ubuntu project can make the types of long-term commitments and promises it does.

The one activity in which the foundation can and does engage is receiving donations on behalf of the Ubuntu project. These donations, and only these donations, are then put to use on behalf of Ubuntu in accordance with the wishes of the development team and the Technical Board. For the most part, these contributions are spent on "bounties" given to community members who have achieved important feature goals for the Ubuntu project.

History of Ubuntu Server

The first "production" machines to run Ubuntu were Canonical's own development machines in its data center in London. In this sense, Ubuntu has been used on servers since day one and Ubuntu has *always* been a server operating system. Of course, as we hinted in the welcome at the beginning of this chapter, this has not always been universally recognized. After the first release, public perception was tilted so far toward the idea of Ubuntu as a desktop release that when the developers convened the first of their biannual developer summits after the first full release cycle, one of the most important items on the agenda was thinking about Ubuntu on servers and how to support it.

The Ubuntu Server project, as a result, was at least as much a marketing project as it was a technical project. Sure, there were ways that the team could make Ubuntu better for servers—and they spent plenty of time working and thinking about that—but the biggest problem they faced was simply communicating the message that Ubuntu already was great for servers to all their users and potential users.

Eventually Canonical funded the creation of a graphical installer, but in the first few releases there was just a single, nongraphical installer based on Debian's very descriptively named Debian Installer project. In the initial Ubuntu release, a user installing Ubuntu was given a choice between two modes: "Desktop"—which was self-explanatory enough—and "Custom." Custom, in the minds of the early developers, was what anyone would want for a server. Custom installed just the bare minimum set of packages and then put the users into this base install and prompted them to install the packages that they wanted on their system. It provided users with the bare-bones system and encouraged them to customize it. The first action of the Ubuntu Server project was purely superficial: The "Custom" install was renamed "Server." Although no code had changed, Ubuntu Server almost immediately began getting more recognition. If one had to pick a single point in time that the Ubuntu Server project was born, it would be this moment.

Ubuntu Server isn't actually any different from other flavors of Ubuntu. As the desktop has moved on to a new graphical installer based on a live CD, Ubuntu Server has its own installer that gives users access to features like RAID and LVM that are much more interesting to server users. Certainly, there are some pieces of software that are likely to end up on servers and unlikely to end up on desktops—things like Web servers and mail servers. When we say that the server edition will be supported, we mean these applications plus the core, so it certainly seems most accurate to refer to these as being within the purview of Ubuntu Server.

But at the end of the day, the server and desktop packages come out of a single repository. This fact, plus the integration between the teams of people working on different parts of the project—most core developers work on bits and pieces that get used and reused in server, desktop, and other editions—introduces a fuzziness that makes it hard to pin down just

what Ubuntu Server *is*. Of course, it also means that Ubuntu Server gets to benefit from the work, bug reporting, and bug fixing in those core parts of the operating system that every Ubuntu user shares.

Ubuntu Server now can roughly be interpreted to refer a collection of resources that are particularly aimed at and used by server users. Most obviously, it involves the custom install discs that you'll be using when you install Ubuntu Server on your machine. It also refers to the collections of supported software that are installed primarily on servers—most of the software that the rest of this book will discuss in more detail. It also refers to a mass of documentation, to which this book represents the latest addition, that provides answers to questions. In a broader sense, certifications of software and training programs for administrators occupy another point in the growing Ubuntu Server constellation.

But most of all, and in the Ubuntu tradition, Ubuntu Server refers to a community. It's a community of developers who use Ubuntu on servers, who care deeply about Ubuntu on servers, and who work tirelessly to make sure that Ubuntu performs as well as possible on servers everywhere. Of course, Ubuntu Server also refers to the growing community of people who are primarily not contributing through code but who are at least as important. These people spend time in the support IRC channels, send e-mail to the mailing lists, and post in the forums. These users help other users, file bugs, may contribute their own fixes to documentation, and contribute in myriad ways and in a variety of venues.

When you "graduate" beyond what this book can teach you, Ubuntu represents those people who will help you take your next steps. They are the people described in more depth in the server resources chapter (Chapter 13) of this book. This is the group you will join when you participate in the Ubuntu project. Let us be the first to welcome to you to the Ubuntu Server community.

Simple, Secure, Supported

Early on, the initial core Ubuntu team—of which one of this book's authors was lucky enough to be a part—resisted the idea of the server version of Ubuntu. Or rather, they resisted the idea of a server distribution in

the way that other GNU/Linux distributions had produced them and the way in which they were commonly thought of. The team was more than happy with running Ubuntu on servers, of course, but they resisted the idea of "server distributions" because of the way that Red Hat, SuSE, and the other big distributions built their businesses around "enterprise Linux" distributions that were big, clunky, and expensive. The result was, in the eyes of many of the early Ubuntu core developers and Canonical employees, top-heavy monstrosities. That's not what Ubuntu is about.

The big server-based GNU/Linux distributions seemed to be competing over who included more services, more features, and more bells and whistles. Distribution 1 would have a Web server, an FTP server, a DNS server, several file servers, and a mail server. Distribution 2 would have all of those plus a DHCP server! A brand-new install of one of these "server distributions" would be running dozens of daemons—each taking up many megabytes of memory, loads of disk space, and (most important) lots of administrator time when they failed or interfered with something else. But worst of all, most of these daemons lay completely unused on most installs.

And if that wasn't enough, the server installs would then run firewalls to keep people from accessing all these now-open services and to prevent users from exposing security vulnerabilities from their newly installed machines. Of course, there would be regular upgrades, security releases, and the like, to update all these now-firewalled services that nobody was using. Debian provided one alternative model that focused on custom installations of just what people needed. Among an elite group of sysadmins in the late 1990s and the early 2000s, Debian had become the server OS of choice. Because nearly everyone on the early Ubuntu team was a Debian developer, it was to this model and to Debian technology that the Ubuntu team first turned.

Of course, the commercial GNU/Linux server market was not all horrible. For example, the early Ubuntu developers liked the idea of commercial support for its servers. They liked the idea of regular, predictable releases. As Debian developers, they all knew someone who wanted to install a simple, custom version of Debian on a server but who, because of the lack of commercial support and accountability, had been rejected by a higher-

up in the company or organization. They liked the idea of a company using Debian's technology to offer simple, custom server installs but could offer a commercial support contract. The Warthogs, and lots of folks like them, had waited years for this, but nobody had stepped up to the plate.

As we described in the previous section, an Ubuntu server install was simply a bare-bones installation. We were all administrators—at least of our own machines—and when *we* installed servers, we started out with "naked" machines so that we could choose every application, every daemon, every service that would go onto the machine. As administrators, we wanted the *options* of the big enterprise distributions, but we wanted to be able to choose those options ourselves. Like all administrators, we used servers to solve problems and to offer services to our users. These problems and needs are unique and, as a result, the cookie-cutter model of GNU/Linux servers was always a poor match.

And so that is what the Warthogs built and it is what Ubuntu Server remains today. At first, some people were confused. Ubuntu's server offering was panned in several reviews for not including a firewall by default. But Ubuntu installed *no open ports by default*, so there was nothing to firewall! Of course, Ubuntu provided several firewalls for users to install *if they wanted one*, but Ubuntu left the decision to install a firewall, just like the decision to install services that might require one, up to the server's administrator. For all installations but for server installations in particular, Ubuntu's goal is to make the default installation simple and secure and to put the user in the driver's seat. Ubuntu's job, as distribution producer, is to make it as close to drop-dead simple for system administrators to do their jobs. In an Ubuntu Server install, every machine is exactly as complicated as the administrator has requested but never any more than necessary. No extra services or unnecessary features are included—although they are waiting in the wings for when they become necessary and are easily installable in ways that will be described in Chapter 3.

One important effect of this simplicity is security. When there is less going on, there is simply less to go wrong. But, of course, the Ubuntu team has taken this many steps further and pursued proactive security in a number of other contexts. Ubuntu's first release was held up for one day because a single open port was found in the default release. The goal of a machine with

no open ports by default was more important than an on-time release. Ubuntu's CTO and the chairman of the Ubuntu technical board, Matt Zimmerman, is a longtime security-focused developer who made nearly all of Debian's security updates for more than a year before joining the Warthogs. As Ubuntu struggles over hard decisions about what to include or to pass up for inclusion in the distribution, the most important questions continue to be ones of security and support. "Can we—and we do want to—maintain security support and provide security releases for this software for the next 18 months?" Every piece of software included by default is subjected to this question, and many popular pieces of software are kept out because Ubuntu is reluctant to support them. Inclusion as an officially supported package means that a server admin can trust the software—both because Canonical has indicated that it trusts it and because Canonical has promised to clean up any security messes that occur through fixing important bugs and issuing a fixed package. Canonical's security guarantee goes beyond security bugs to other bugs that might result in data loss. While there are no guarantees beyond this, Canonical makes many dozens of new updates per release that fix other important bugs in the distribution as well. The result is a rock-solid system with a commitment to continue.

With customizability, security, and support, Ubuntu truly is ready for the data room. The rest of this book will show you how.

CHAPTER 3

Package Management

THIS CHAPTER WILL BEGIN WITH a discussion of packages in general while focusing on the core features of packages and package management systems that cross most GNU/Linux distributions. In this discussion, I will attempt to explain what packages are and what a package management system does. While I'll turn to examples from Ubuntu throughout, this discussion will focus on building a strong conceptual grounding. After establishing a solid grounding, I'll introduce Debian packages—the types of packages that Ubuntu uses—and give a brief view of the very different types of packages: source packages and binary packages. The majority of the rest of the chapter will focus on package management in Ubuntu using the command-line tools. While many users of Ubuntu on the desktop will be familiar with updating their system, this chapter will focus on the way this is done without a desktop system. In this description, I'll cover the basics and work up to some more advanced uses of a packaging system that many server administrators find useful. Finally, I'll touch on the process through which advanced users and administrators can create, modify, and finally redistribute their own packages.

Introduction to Package Management

On Ubuntu—and in other GNU/Linux environments—packages are the primary way that software is built, deployed, and installed. Nearly every major GNU/Linux operating system distributes software, both binary software and source code, in packages. These packages are usually either in the Rpm package format (RPM) or in the Debian package format (DEB) for binary software or in corresponding "source" RPM and DEB formats. With its close relationship to the Debian project as a project that continues to be based on Debian's work, Ubuntu naturally uses DEB format packages.

Very simply, packages are an alternative to downloading, building, and installing software from scratch. They offer a host of advantages in terms of installation, removal, monitoring, and handling interactions between pieces of software over the standard "build from source" model. Since packaging is not common outside of the GNU/Linux world—or at least

not described in the same terms—it is worth going into some background on packaging before I describe how it is done on Ubuntu systems.

Background on Packages

Nearly every GNU/Linux-based operating system—Fedora, RHEL, openSUSE, Slackware, Debian, etc.—includes an almost entirely overlapping core selection of software. By definition, each of these OSs includes Linus Torvald's Linux kernel and a large chunk of the GNU project's developer- and user-oriented applications that are necessary to build and use it. Most also include server-oriented software like OpenSSH and Apache, either the XFree86 or X.Org implementation of the X Windowing System, and what is often an extremely expansive collection of both command-line and graphically based applications. Although people often throw the term around, it is important to establish that it is this collection of software that is collectively referred to as a distribution. Ubuntu is a distribution. When people refer to "Linux" as an operating system, they are usually referring to a Linux or GNU/Linux distribution.

The primary goal of all distributions is the automatic installation, configuration, removal, maintenance, and update of software—both through the creation of infrastructure for this purpose and in the creation of modified versions of the preexistent software. The latter customization of existing software in this specialized way is the act of "packaging," and it constitutes the vast proportion of work done by Ubuntu developers. It constitutes, to a large degree, what Ubuntu is over and above the software that Ubuntu includes. And while packaging is primarily the work of distribution makers like Ubuntu, it can also be done by both the users of distributions, for the clean integration of "unpackaged" pieces of software into their systems, and by software vendors who wish to allow for easier installation and maintenance of software by their users.

What Are Packages?

The creation of a package—on Ubuntu or elsewhere—begins with the software in need of being packaged. In most, but not all, cases, this involves

the procurement of source code. In all situations, it will involve code from an original source usually referred to in the distribution world as an "upstream" source. The packager's first addition to the code here will be the creation of extra metadata, which usually includes

- The name of the software
- The name of the upstream author and the person creating the package
- The license of the software
- The upstream location of the software (or a description of where it was obtained)
- The architecture or architectures on which the software is guaranteed to run
- Information for classifying the software that often has to do with the use of the package, primarily to help people who are browsing for packages
- A description of the software in a computer-parsable format
- Information on the importance or "priority" of the package within the larger Ubuntu system (e.g., essential, optional)

This information will be used by either a packaging system or a series of package selection tools to allow users to search, sort, query, or interact with installed or available software—one of the package system's jobs. However, while this type of metadata is important in that it allows users to find (and find out about) their software, by far the most important group of metadata added to a package relates to the documentation of the relationship of the software in the package to software in other packages within the distribution. While the syntax and semantics of this vary widely between distributions, they include relationships to

- Other software that the software requires to be built
- Other software that the software requires to be installed or configured
- Other software that the software requires to be run
- Other software that the software cannot be installed or used with simultaneously

- Other software that the software can be used as a drop-in replacement for
- Other software that the software can be enhanced or improved by

Modern package systems record even more information. For example, configuration files, unlike normal files, cannot always be simply replaced with a new version when the software is upgraded. As a result, packaging systems have grown to include several pieces of infrastructure for querying users and for maintaining core configuration information over time and across upgrades of the package that requires changes to configuration files. Finally, a more recently realized goal of packages is to provide a structure around which package metadata—such as descriptions—can be translated to provide users with an interface to software localized to their language, script, and culture. Details on accessing and creating all of this metadata in Ubuntu packages will be included in the subsequent sections.

Basic Functions of Package Management

There is a wide range of functionality that can be considered core functions of package management systems. These are usually implemented by a low-level tool or suite of tools. This script is `dpkg` and associated scripts in the case of Ubuntu and Debian. These tools were, until several years ago, the primary way that most users manipulated packages, but with the creation of higher-level package management tools that provide "front ends" to these tools, most users of package-based systems rarely use them, although they are still highly central for developers or system administrators who build their own packages. Broadly and somewhat imprecisely, many of these tools are referred to as APT on Debian and Ubuntu.

The first goal of packaging is automating the compilation of software. DEB-format packages provide two formats: one for source packages and one for binary packages. These source packages are an excellent system for the distribution and compilation of source code. Packages are, in Ubuntu and elsewhere, designed to be built noninteractively and—in the case of official Ubuntu packages—can be built automatically on a range of different architectures by automatic package-building software called "autobuilders." Packages provide a simple—usually one command—method for building that is consistent across all packages. Issues of build configurations

and choices are addressed ahead of time by the packager. The cost is build-time configurability, but the payoffs, as you will see in the rest of the chapter, are huge. Necessary build-time dependencies are declared in the packages so that these can be satisfied automatically. For example, architecture-dependent source packages (i.e., packages that will need to be rebuilt for each architecture) are uploaded to Ubuntu as source and are, in most cases, automatically built on all architectures supported by Ubuntu without any changes to the source package.

Any number of binary packages can be created from a single source package. The creation of multiple binary packages from a single source package can be useful for large projects that release large or monolithic source packages containing a wide variety of different pieces of software—or even highly related pieces of software and/or documentation that it may be advantageous to split. An example of the former case is the XFree86 windowing system—now replaced by the already modularized X.Org—which was contained in one source package but would create upward of several dozen binary packages. Packaging, in this case, is what allowed users to distribute, install, and remove the Xserver independently from the terminal emulator, xlib library package, or window manager.

As can be inferred from the preceding discussion, a key benefit of packaging systems is that they help automate the installation of software. When a binary package is installed:

- The "contents" of the software can be verified to assure integrity of the package. The origin of the software can be verified using cryptographic authentication.
- The dependencies of the software can be analyzed and the system can be queried on the installation state of the software on which the software being installed depends. If the dependencies are unsatisfied, the user will be prompted as to the lack and the nature of the required software, and the installation will be aborted.
- The user installing the package can be queried for configuration options at some point during the installation process. Answers to these queries can be saved on the system and then used in the customization of a configuration file for the software being installed.

- The contents of the package will be stored on the system.
- Metadata and accounting information of a variety of forms will be placed in a per-system database to include both current information on the packages installed and their state of installation (e.g., installed but unconfigured), the list of files and to which package they belong, and other information.

Perhaps the most central element here is the check against dependencies of the package being installed and the list of packages already installed on the system. With information on dependencies, users can, at a glance, determine which software is required to run the software in the package. As a result, people writing software that will ultimately be packaged can easily write for and deploy software built against shared libraries. The success of package systems is one reason for the wide use of dynamically linked shared libraries in the GNU/Linux environment.

When a user wants to remove a piece of software, the packaging system, with its catalog of the files belonging to the package and the actions done during installation, is well suited to help ensure a clean uninstallation as well.

While similar to installation, the automatic upgrade of software is another area where the package system can be employed with similarly useful results: Users of package systems can safely and easily upgrade from one version of a piece of software to another. The upgrade of the software will work almost identically to the installation of the software. In most cases, software will be installed on top of the existing package, and files that are no longer provided by the package will be removed. Configuration files that were customized by the installation and have not been changed by the user since can be automatically regenerated by the user, or the user can be prompted to view and merge changes.

Dependency information can play an important role in the upgrade of packages involving shared libraries. In the case of ABI changes, a packaging system will alert users that an upgrade of a package cannot be completed without the installation of a new library, and users can also be alerted to other packages that will break in this upload. As a result, users can structure uploads—or the system can structure it for them—so that API and ABI

breakage is not unanticipated, and users can ensure that all packages that depend on a single shared library can be upgraded in tandem.

Finally, at any point, users can use the cryptographic signature on a package and the list of hashes (usually MD5 sums) of the files included in that package to verify the integrity of the files on their system against corruption or compromise by an attacker.

Advanced Functions of Package Management Systems

While these features lead to the powerful potential to manage software on a system, packaging systems with *only* these features—essentially, the state of packaging in the mid-1990s—introduced important limitations. Large-scale API and ABI transitions required downloading many packages and a high degree of coordination by the user. Users were forced to figure out the dependency status of programs during an installation or upgrade and then find, download, and do simultaneous installations of new pieces of software. For complex pieces of software with many dependencies, this process was often exceedingly tedious.

As a result, most system upgrades and ABI/API changes were done with large upgrade scripts between releases of a distribution. Users would be expected to install every package involved in a major transition at once with an upgrade script that would structure the order correctly and handle dependencies appropriately. While these problems are limitations of a limited package management system, they are mostly problems that exist outside of package management systems. Without a package management system, shared libraries that undergo API and ABI changes are either never or rarely approached (with dangerous consistency and security implications to each) or are subject to the same limitations without the warnings that a packaging system provides.

Spurred on by the Debian project's creation of a program called `dselect` and its frequently lauded Advanced Package Tools (APT, originally named *deity* and implemented primarily in a program called `apt-get`), the last half-decade has seen a major evolution in the scope and success of package managers. Most of these tools are levels of abstraction upon or "front ends" to the lower-level package management tools previously described.

Like most other DEB-based distributions, Ubuntu uses `apt-get`, Aptitude, `dselect` and the graphical front end, and Synaptic.

As the ability to track and catalog dependencies is perhaps the single most important aspect of any package management system, the primary function of these advanced tools has been to add classes of functionality on top of the extant package tools and to operate on packages in a more-than-one-at-a-time manner. Each of these tools contains additional databases that describe not only the packages installed but also the packages that are *available* as candidates for installation through package archives stored locally, on CD, or (in almost all situations today) over a network.

These systems can automatically sort out dependencies and orders, download packages (including dependencies), install the dependencies first, and then install and configure the package in question using the lower-level tools detailed in the previous section.

Similarly, the same advanced tools can be used to uninstall packages. If, for example, a user wants to uninstall a shared library, he or she will be prompted with a screen that describes the consequences as a list of packages that will need to be uninstalled because their dependencies will no longer exist on the system after the uninstallation. Upgrades that involve changing dependencies (e.g., replaced packages) can also be handled through this system.

The real possibilities of such systems are visible when the dependency aspects of a package change over time or when multiple packages can act as drop-in replacements. A package that requires the ability to send mail can depend only on a virtual package "provided" by other packages. New versions of packages can conflict with and declare that they "replace" other packages or provide the functionality of the original package. If, for example, multiple packages are merged into a single package that obsoletes the three other packages, an advanced package system should be able to track the changing dependency information and make the correct decision during upgrade. Along these lines, most advanced package management tools give users the ability to do strategic "smart upgrades" of every package on the system to the newest version of the packages available using the data declared in the package dependencies.

Even more exciting for some users, it is possible to track an in-development version of a GNU/Linux operating system and upgrade every day to the latest version of everything. The package manager can figure out safe upgrade paths and take it from there. During these upgrades, ABI and API version changes can also be automatically handled as the system will refuse to do a full upgrade of a library until all of the packages installed on the system that depend on the package with the shared library can be upgraded at once. The system will not need to keep or track multiple versions of a shared library over time.

Debian Packages

As was mentioned earlier in this book, the Ubuntu project is based on the Debian GNU/Linux distribution. Among many other technological legacies, Ubuntu has inherited the Debian package system. In fact, many core Ubuntu developers involved early on will credit Debian's packaging system as the reason that Debian proved such an attractive point of departure and represented its major attraction over other GNU/Linux distributions As a result, almost all aspects of package management—from the formats to the tools—are identical on Ubuntu and Debian. In many situations, unmodified Debian packages can simply be installed on Ubuntu. In nearly all situations, unmodified Debian source packages can be built on Ubuntu. As a result, our first step will be to examine an Ubuntu DEB in some depth to understand the anatomy of the package and the way that it implements the features described in the preceding sections.

Source Packages

DEB source packages are clearly expressed in what is usually a three- or two-file format but may also include source packages that consist of many more files as well. This means that the package itself contains multiple files and downloading "a source package" may in fact involve downloading a small assortment of different files. Source packages can be broadly classified as either *native* DEB packages or *nonnative* DEB packages. A native DEB is a piece of software where there is no difference between the upstream version and the DEB package. In most cases, native packages are specific to either Ubuntu, Debian, or another Debian-based distribution. In other words, a native package will require no changes in order to create the package. A DEB source package will always consist of a "pristine"

source archive in the form of a gzip-compressed GNU tar file and a DSC file that will list the contents of the package and can be considered the "core" of a source package. An example DSC for a program called most that I maintain looks like this:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

Format: 1.0
Source: most
Binary: most
Architecture: any
Version: 5.0.0a-1
Maintainer: Benjamin Mako Hill <mako@debian.org>
Standards-Version: 3.7.3
Build-Depends: debhelper (>= 4), libslang2-dev
Files:
  30f2131b67f61716f6fe1f65205da48b 155233 most_5.0.0a.orig.tar.gz
  07e3eb05ad5524fe6d885f5cdc2eb902 20160 most_5.0.0a-1.diff.gz

-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQFH4IoAic1LIWB1WeYRAjyOAKCrLCfuZA7b8JcvYTFYeuHrF7r34wCfVTBS
/jGUfIrELNq173sM9CorZA4=
=/Cia
-----END PGP SIGNATURE-----
```

The file is signed with a GnuPG or PGP key to ensure the integrity of the file and the identity of the author. If you were to check this signature with GPG, you would see that it was signed by my GPG key. The DSC file also contains information on the version of the source format (in this case, it's the "old" format, 1.0), the name of the source package, the version of the package (split into the version of the upstream source and the version of the package after the final -), the name and e-mail of the maintainer, the architecture on which the software will run, the version of policy (marked as "standards") against which the software was created, the software that must be present to build the package, and a list of the other files this source package contains, identified by file size and MD5 hashes.

In a native DEB, there would be only a single compressed tar (tar.gz) file. In this nonnative package, there would be additional files that represent all changes to the package. This is so all the changes that the DEB packager

made are clearly visible. This is sometimes done for license reasons but is usually done just so that users can see exactly what the packager has done and what has been changed. This also makes it easy for the package maintainer to understand where a problem lies if there is an error. Changes made to a package are usually expressed in a gzip-compressed diff file that expresses all the differences between the package source and the pristine source. In the case above, it is listed as most_5.0.0a-1.diff.gz. In newer versions of the DEB source package format, additional tar files containing additions or changes to the pristine source archive are also permitted, as long as they are listed in the DSC file in the list of files.

When unpacked and with all necessary patches applied, every DEB source package will unpack into a single directory of the form packagename-version with a mandatory debian directory as a subdirectory. In the vast majority of packages, almost all changes to the source will be made inside this directory. This directory will contain a number of files—more than I have space to cover here. Most important among these are the control file and the rules file. The control file will include most of the information about the source package found in the DSC file (which is autogenerated using control file data) and additional information describing each binary package. The control file expresses all interpackage relationships including Depends, Conflicts, Provides, Replaces, Recommends, Suggests, and Enhances. As the "hard" requirements, the first four are most important. Suggests and Enhances are rarely used by any program. The file will also include both a single-line and a multiline description. A sample control file (the control for most) is shown below:

```
Source: most
Section: text
Priority: optional
Maintainer: Benjamin Mako Hill <mako@debian.org>
Standards-Version: 3.7.3
Build-Depends: debhelper (>=4), libslang2-dev

Package: most
Architecture: any
Depends: ${shlibs:Depends}
Description: Pager program similar to more and less
```

The long-form description was removed from the output but in fact followed the final description line and includes text that is indented by one space and where paragraphs are separated by a single ".". As mentioned

previously, the control file consists of a series of stanzas. The first stanza will always begin with `Source:` and will include information on the source package. Each following stanza will describe a single binary package. In this case, there is only one binary package, which, like the source package, is named `most`. This situation—a single source package creating a single binary package of the same name—is a very common case.

The rules file is a GNU Make makefile and contains all of the makefile rules to create and build a package. Running debian/rules binary from within the unpacked source package directory will result in the creation of a Debian package in one directory above (../) if your system has all the necessary dependencies installed. In most cases, the software will build and "install" into a series of subdirectories within the debian directory; these files in their temporary location will then be included as the package contents.

Additional files in the debian directory include the copyright file, the changelog for the package, optional scripts to be run after and before installation or removal of the package, and extra configuration data plus anything else that the packager would like to include.

Binary Packages

Debian binary packages are very simple in format, so it is unnecessary to spend much time on them here. More important, they are almost never manipulated by hand. Binary packages are merely installed and removed. Changes to a binary package are made first in the source package and then new, changed binary packages are rebuilt. In Ubuntu and Debian, binary packages are a single file in an archive in the ar format. In the archive is debian-binary, which contains a series of lines, separated by newlines. At the moment, only the format version number is included. The second member of the archive is named control.tar.gz, and it contains the package control information (as described previously). The third and last member is called data.tar.gz, and it contains the file system archive as a gzipped tar archive.

Package Management in Ubuntu

The administrator of every Ubuntu installation—servers and desktops—will need to learn the basic mechanics of package management. As administrators need to find new software to solve particular problems, metadata

in the packaging system can be a great place to start. When administrators want to install new software, the packaging system provides the best way to do so. The Ubuntu package system will also allow users to install and remove software, check for updates—and for security updates in particular—and install these updates. Finally, when a new release of Ubuntu is made, the packaging system will allow administrators to update their systems.

Ubuntu provides a variety of different tools for package management. On a desktop Ubuntu system, users' interaction with the package management system is primary through a little icon on the desktop that alerts them to new releases of software and through the graphical Add/Remove Programs application and a second graphical package management program called Synaptic that provides functionality to let users browse the package archives. Since these programs are covered in depth in *The Official Ubuntu Book* and because the focus of this book is servers, this section will focus on the command-line tools for package browsing and management.

Most server administrators primarily use tools in the APT family that handle high-level package management. The original tool developed for this purpose was `apt-get`. Aptitude is a frequently used alternative to `apt-get` that provides both an interactive front end and that takes most of the default `apt-get` commands. Many of the commands described in the rest of this chapter that call `aptitude` can also be used with `apt-get` with little or no difference in either output or behavior. The primary differences are in the ways that the systems resolve complicated dependency situations and certainly would not affect the reasonably simple operations described here.

Staying Up-to-Date

Each Ubuntu system stores a list of package repositories in /etc/apt/sources.list. This describes the list of "places" where your package managers—originally just APT but now several other tools—will look for updated versions of software. These sources may include local repositories on your file system, a CD in your computer, or—as is common in the vast majority of situations—a network location. To update the system, you can run `apt-get update` or `aptitude update`.

This command will download the latest updated package lists for all repositories listed in your /etc/apt/sources.list files and will check any

cryptographic signatures on these updates against the keys stored on your machine. On a new system, this will check only the Ubuntu package repositories that include the repositories you installed from and the security repositories.

Installing any new version of packages is as simple as running `aptitude safe-upgrade`, which is a replacement for the `apt-get upgrade` command that may be more familiar to more seasoned users. `safe-upgrade` simply tries to upgrade all installed packages to their most recent versions. Installed packages will not be removed unless they are unused, although additional packages may also be installed in order to resolve added dependencies.

APT can be configured to automatically download and upgrade packages with new versions. This is an attractive proposition to administrators who like the idea of not having to log in to their systems to keep them up-to-date. However, automatic package upgrades are subject to errors because of the particular status of software on the system or even particular configuration changes that have been made, so these automatic package upgrades can leave systems in unstable or unworkable states. As a result, automatic upgrades are neither covered in this book nor recommended by the authors.

Searching and Browsing

Historically, the primary way of searching for new packages was using the program `dselect`. Users of Ubuntu on the desktop will primary use the Add/Remove Programs application and the graphical program Synaptic. Users on the console have several other options.

First among these is the simple program `apt-cache`, which can provide statistics about and information on packages. If, for example, I decide I want a pager like less, I can search for one in the following way:

```
$ apt-cache search pager less
less - Pager program similar to more
wdiff - Compares two files word by word
console-log - Puts a logfile pager on virtual consoles
gdesklets-data - Applets for gdesklets
jless - A file pager program, similar to more(1) supporting ISO2022
most - Pager program similar to more and less
nagios-plugins-basic - Plugins for the nagios network monitoring
  and management system
```

As you can see from the previous list, the `apt-cache search` command returned eight "hits" for my search on the two keywords *pager* and *less* and returned a list of package names followed by short one-line descriptions. The keyword search looked through the full list of available packages and focused on the package names, short descriptions, and full descriptions that are not shown in the returned list. If I want to know more about a package, `apt-cache` can also show me more about the package with the `show` subcommand as in the example below:

```
$ apt-cache show most
Package: most
Priority: optional
Section: universe/text
Installed-Size: 172
Maintainer: Ubuntu MOTU Developers <ubuntu-motu@lists.ubuntu.com>
Original-Maintainer: Benjamin Mako Hill <mako@debian.org>
Architecture: i386
Version: 5.0.0a-1
Depends: libc6 (>= 2.7), libslang2 (>= 2.0.7-1)
Filename: pool/universe/m/most/most_5.0.0a-1_i386.deb
Size: 48092
MD5sum: e089c00005b536e1b8848b7087df2bae
SHA1: 4f4ab395f340be4804732452aa112007916f90cb
SHA256:
  ccf50fb49270e7ddf7735da23e699afcd11dcfc8e241973bb17ad03bf49e6f4a
Description: Pager program similar to more and less
 Most is a paging program that displays, one windowful at a time, the
 contents of a file on a terminal. A status line at the bottom of the
 screen displays the file name, the current line number, and the
 percentage of the file so far displayed.
 .
 Unlike other paging programs, most is capable of displaying an
 arbitrary number of windows as long as they all fit on the screen,
 and different windows could be used to view the same file in
 different positions.
 .
 In addition to displaying ordinary text files, most can also display
 binary files as well as files with arbitrary ascii characters.
Bugs: mailto:ubuntu-users@lists.ubuntu.com
Origin: Ubuntu
```

You may recognize that quite a bit of this information looks like the source package information and the corresponding stanza referring to this binary

package in the control file described previously. Sure enough, this is exactly where this metadata has been extracted.

Of course, the bulk of the output is made up of the long-form description that was omitted in the previous example. There are some other fields of potential interest, including the "Original-Maintainer" or the person who packaged the system in Debian, the "Maintainer" or the person or group to contact with questions about or issues with the package, and sizes and hashes (e.g., MD5Sum, SHA1, and SHA256), which describe ways to identify that a particular version of the package was downloaded correctly and has not been modified.

Called with no arguments, Aptitude also can provide users with a Curses-based text-based interface that allows for more interactive browsing of all the packages available. For users familiar with Synaptic, this can be thought of as a text-based version of the Synaptic interface. In this mode, many search results can be navigated through with the arrow keys and different applications can be "marked" for installation.

Before concluding this tour of the options for searching and browsing for packages, it is worth pointing to the Web site at http://packages.ubuntu.com. This interface lets users search in ways that are similar to some of the tools I have shown here but with several additional useful options. In particular, the Web site lets users search for particular files in *any* package in Ubuntu. Normally, users are able to find out only which package "owns" a file if they have the package on their system. If, for example, you need a particular header file or shared library and you know only the filename, you can search on the Web site for that filename throughout all packages available in the Ubuntu archive.

Installation and Removal

Installing and removing packages is another simple task that you will do frequently. To install a package, you can invoke `apt-get` or Aptitude in a similar way, although, unlike searching, a user must be running with root privileges to do so. The recommended way to do this would be to use the `sudo` command. Since prefixing each command in this section with `sudo`

would be tedious, I have assumed the user is root, although having the user logged in as root would not be considered the best form. If I want to install most, I can simply run the following command as root:

```
# aptitude install most
Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
Writing extended state information... Done
The following NEW packages will be installed:
  libslang2{a} most
0 packages upgraded, 2 newly installed, 0 to remove and 0 not
upgraded.
Need to get 0B/509kB of archives. After unpacking 1323kB will be
used.
Do you want to continue? [Y/n/?] y
Writing extended state information... Done
Selecting previously deselected package libslang2.
(Reading database ... 362131 files and directories currently
  installed.)
Unpacking libslang2 (from .../libslang2_2.1.3-3ubuntu1_i386.deb) ...
Setting up libslang2 (2.1.3-3ubuntu1) ...
Selecting previously deselected package most.
(Reading database ... 362143 files and directories currently
installed.)
Unpacking most (from .../most_5.0.0a-1_i386.deb) ...
Processing triggers for man-db ...
Setting up most (5.0.0a-1) ...

Reading package lists... Done
Building dependency tree
Reading state information... Done
Reading extended state information
Initializing package states... Done
Writing extended state information... Done
```

You can see in the output of the command above that `libslang2` was installed alongside most. In this case, Aptitude saw that most required the S-Lang library but that it was not installed. Aptitude prompted me for confirmation about the installation of the additional package (which I approved), downloaded both packages, and then installed and configured them on my system.

Removing a package is similar and similarly simple. If I decide to remove most, I can do so by running

```
# aptitude remove most
```

In this case, `libslang2` will *not* be removed (since I have not asked for it to be removed). If I were instead to try to remove `libslang2`, Aptitude would prompt me and explain that removing `libslang2` would also require removing all of the packages that depend on it—on this system, that would just be most, but for other packages or on other systems there could be quite a few packages. This type of dependency management means that, for example, users should not (and cannot easily) remove core or essential packages. Extra "unused" packages can be removed using the command `apt-get autoremove`.

Finally, while these examples both used Aptitude, the installation and removal of packages can also be done with the lower-level tool `dpkg`. In fact, in both cases Aptitude is simply calling `dpkg` on the downloaded package files behind the scenes. Aptitude—or `apt-get`—will always download packages and work out dependencies before turning to `dpkg`. If you have already installed existing dependencies, you can install a DEB directly with `dpkg` by using the `-i` command and passing the package filename as an argument. For example, if I were given a DEB file for most, I could install it with a command like this:

```
$ dpkg -i most_5.0.0a-1_i386.deb
```

`dpkg` will check dependencies and produce an error if there are missing dependencies but will not automatically download or install packages since it does not contain the functionality to do this. I could uninstall most with `dpkg` with the command `dpkg -r most`.

Manipulating Installed Packages

`dpkg` provides dozens of methods of querying, searching, and manipulating installed packages. It contains a database of information about packages

installed on the system. To get a quick overview of what this might look like, you could run the following command:

```
$ dpkg -l most
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Cfg-files/Unpacked/Failed-cfg/Half-inst/
  trig-aWait/Trig-pend
|/ Err?=(none)/Hold/Reinst-required/X=both-problems
  (Status,Err: uppercase=bad)
||/ Name         Version      Description
+++-============-============-=============+======================
ii  most         5.0.0a-1     Pager program similar to more and less
```

Run without any arguments, `dpkg -l` will show this basic information on the installation status, name, version, and description of *every* package on your system.

Another simple task is to get a list of files contained within the package. If you have a DEB file that you have not installed, you can get this information by running `dpkg --contents` as in the example below:

```
$ dpkg --contents /var/cache/apt/archives/most_5.0.0a-1_i386.deb
drwxr-xr-x root/root        0 2008-05-06 12:06 ./
drwxr-xr-x root/root        0 2008-05-06 12:06 ./usr/
drwxr-xr-x root/root        0 2008-05-06 12:06 ./usr/bin/
-rwxr-xr-x root/root    59940 2008-05-06 12:06 ./usr/bin/most
drwxr-xr-x root/root        0 2008-05-06 12:06 ./usr/share/
drwxr-xr-x root/root        0 2008-05-06 12:06 ./usr/share/man/
drwxr-xr-x root/root        0 2008-05-06 12:06 ./usr/share/man/man1/
-rw-r--r-- root/root     5912 2008-05-06 12:06 ./usr/share/man/
  man1/most.1.gz
drwxr-xr-x root/root        0 2008-05-06 12:06 ./usr/share/doc/
drwxr-xr-x root/root        0 2008-05-06 12:06 ./usr/share/doc/most/
-rw-r--r-- root/root     2989 2007-09-09 12:14 ./usr/share/doc/
  most/changelog.gz
-rw-r--r-- root/root     5544 2008-05-06 12:06 ./usr/share/doc/
  most/copyright
-rw-r--r-- root/root     3335 2007-09-06 10:15 ./usr/share/doc/
  most/README
-rw-r--r-- root/root     1386 2006-05-01 13:51 ./usr/share/doc/
  most/lesskeys.rc
-rw-r--r-- root/root      492 2006-05-01 13:51 ./usr/share/doc/
  most/most-fun.txt
-rw-r--r-- root/root     3086 2006-05-01 13:51 ./usr/share/doc/
  most/most.rc
```

```
-rw-r--r-- root/root      2028 2008-05-06 12:06 ./usr/share/doc/most/
  changelog.Debian.gz
drwxr-xr-x root/root         0 2008-05-06 12:06 ./usr/lib/
drwxr-xr-x root/root         0 2008-05-06 12:06 ./usr/lib/mime/
drwxr-xr-x root/root         0 2008-05-06 12:06 ./usr/lib/mime/
  packages/
-rw-r--r-- root/root        94 2008-05-06 12:06 ./usr/lib/mime/
  packages/most
```

Similar information for installed packages can be retrieved with `dpkg -L`. Working in the other direction, if you have a particular file and you want to know which package "owns" it, you can use `dpkg -S` to query the database for this information. For example:

```
dpkg -S /usr/bin/most
most: /usr/bin/most
```

The binary file /usr/bin/most belongs to—no surprise here for anyone who's gotten this far—the binary package called most. Since this command is searching through each of the file lists of every package on your system, it may take some time to complete.

Manipulating Repositories

The best way to install new software in the "Ubuntu way" is never to simply download new DEB packages and install them "by hand" with `dpkg`. But APT is only kept up-to-date with the packages that it already knows about. While `dpkg` works on packages, APT works on repositories of packages that contain information on different packages, their versions, and their dependencies. As a result, to manage a package through APT, one needs to add to the system not the package, but rather the repository that contains it. This is done by adding or editing the list of "sources." While the Ubuntu desktop distribution includes a graphical tool for manipulating repositories, it can be done easily by hand, which will be the default on most systems.

The source.list file, already mentioned several times in this chapter, is located at /etc/apt/sources.list on every Ubuntu and Debian system and is made up of a series of lines like this:

```
deb http://archive.ubuntu.com/ubuntu intrepid main universe
deb-src http://archive.ubuntu.com/ubuntu intrepid main universe
```

The first word will either be a # symbol marking the line as a comment or either `deb` or `deb-src`. This specifies whether the repository is a source package repository or a binary package repository. The second item is the location in the form of a URI. The third item is the name of the distribution or, as it might more accurately be described, the distribution version. In the example above, this distribution version is `intrepid`, which refers to the Ubuntu release of the Intrepid Ibex. The remaining arguments are the lists of the components. The components provided in the core Ubuntu repositories are detailed in the following section.

An example will help illustrate the process of adding a repository. If I want to install a version of Bazaar that is always the latest released version, I will need to do this from outside the default Ubuntu repositories, which will only be updated based on the Ubuntu release cycle. Luckily, the Bazaar developers provide their own "Personal Package Repository"—a subject I'll come back to at the end of this chapter. On their Web site, they provide the `deb` and `deb-src` lines that I can simply drop into my sources.list:

```
deb http://ppa.launchpad.net/bzr/ubuntu intrepid main
deb-src http://ppa.launchpad.net/bzr/ubuntu intrepid main
```

If I update, I am first greeted by an error that claims that I do not have the correct cryptographic key to verify that the packages in the repository are really coming from the Bazaar developers:

```
W: GPG error: http://ppa.launchpad.net intrepid Release: The
following signatures couldn't be verified because the public key is
not available: NO_PUBKEY FE8956A73C5EE1C9
```

I can easily install that by downloading the key from a trusted source like the PPA providers' Web site and saving it into a file (called /tmp/keyfile in the example below), verifying that is correct, and adding to the package manager's key database with a command such as

```
apt-key add - < /tmp/key
OK
```

The `apt-key` manual page gives more details on how keys for repositories can be managed with this useful command.

Ubuntu Default Repositories

The vast majority of packages that you will need have been packaged for Ubuntu. This is because, leveraging the work of Debian, Ubuntu provides access to a large majority of the most popular pieces of free software as packages in their own repositories.

These tens of thousands of packages are separated into a series of different sections or components. You can toggle these on and off by including them in the list of components in your sources.list. Because these have important consequences for the level of support you will receive for your software, it is worth understanding these different components so that you can decide from which areas you want to pull software. Available components on the Ubuntu server include main, restricted, universe, and multiverse. The following descriptions are adapted from the component descriptions on the Ubuntu Web site.

- **Main**
  The main distribution component contains applications that are free software, can freely be redistributed, and are fully supported by the Ubuntu team. These include the most popular and most reliable open source applications available, much of which is installed by default when you install Ubuntu. Software in main includes a hand-selected list of applications that the Ubuntu developers, community, and users feel are important and that the Ubuntu security and distribution teams are willing to support. When you install software from the main component, you are assured that the software will come with security updates and technical support.

- **Restricted**
  The restricted component is reserved for software that is very commonly used and that is supported by the Ubuntu team even though it is not available under a completely free license. Please note that it may not be possible for Ubuntu to provide complete support for this

software since the Ubuntu team is unable to fix the software but can only forward problem reports to the actual authors.

- **Universe**
  In universe one can find almost every piece of open source software and software available under a variety of less-open licenses, all built automatically from a variety of public sources. All of this software is compiled against the libraries and using the tools that form part of main, so it should install and work well with the software in main, but *it comes with no guarantee of security fixes and support*.

- **Multiverse**
  The multiverse component contains software that is not free, which means the licensing requirements of this software do not meet the "main" component license policy. The onus is on you to verify your rights to use this software and comply with the licensing terms of the copyright holder. This software is not supported and usually cannot be fixed or updated. Use it at your own risk.

Using Other Repositories

As you saw when I added the Bazaar repository several sections ago, users will still sometimes want to make use of a variety of outside repositories beyond what is provided in Ubuntu. For example, users might want to install new versions of particular applications or libraries from the development release of Ubuntu but might not want to upgrade all of their packages to the latest version.

The quasi-official "backports" repository in Ubuntu is a useful resource. It contains versions of software from the development version of Ubuntu that have been "backported" to install cleanly on stable versions of Ubuntu. You can add the backports by installing a DEB package by hand in a one-by-one with `dpkg` or by adding an extra line to your sources.list. Information on doing both can be found on the Ubuntu Web site at https://help.ubuntu.com/community/UbuntuBackports.

One reason that many users choose to go the à la carte method—that is, the method of downloading packages by hand and installing them with `dpkg`—as opposed to just adding the repository is because of a limitation in the

way that APT works: *APT and other tools will always install the newest version of any package available by default*. This means that if you add the backports repository, or the development repository for that matter, to your sources.list, the latest version of *everything* in that repository will be installed when you try to run an upgrade. For small repositories (like the Bazaar PPA described several sections ago that contained only Bazaar and several closely linked packages) this does not present a problem. However, in situations where you want to add a large repository of many packages like the backports repository or the development release of Ubuntu but only want a few packages, the effects will often not be what you want.

The general solution to this problem is called "pinning" or "apt pinning." Pinning is extraordinarily powerful but, in its advanced forms, can also be very complicated. As a result, a full discussion is outside the scope of this chapter. That said, an example is shown below for the situation where I have Intrepid installed but want APT to prefer packages in Jaunty. To change this, I would need to edit the file /etc/apt/preferences so that it included something like the following section:

```
Package: *
Pin: release a=intrepid
Pin-Priority: 700

Package: *
Pin: release a=jaunty
Pin-Priority: 600
```

Each stanza describes one release and, as is represented by the wildcard in the first line, applies to all packages. In the final line of each stanza, the pin-priority describes both relative position (i.e., in the example above, Intrepid is preferred to Jaunty) and weight that will be given to each. Weights can be tweaked so that packages will be installed, or not, except in special circumstances. Much more information on pinning is available in the `apt_preferences` manual page and in several excellent pieces of documentation on the Ubuntu and Debian wikis.

Upgrading a Whole System

A final basic task that every system administrator will need to do is to upgrade a full system. On desktop Ubuntu systems, the default way of

handling an upgrade is by using the update manager software. However, this software is designed specifically to upgrade graphical systems. Since the process can just as easily be done from the command line, that will probably be more appropriate on most servers.

In the past, upgrading most systems was a two-step process. First, the administrator would update the list of repositories (detailed in the previous section) so that references to the old release were replaced with the new release. For example, if I were upgrading from the Hardy Heron to the Gutsy Gibbon, I would replace every instance of *hardy* with *gutsy* in my source.list file. After doing this, I would run `aptitude update` exactly as I described in the section above on staying up-to-date. This would refresh my local package metadata cache with a list of all the packages in the new distribution.

Finally, I would run `aptitude full-upgrade` which, unlike `safe-upgrade`, described previously, would upgrade all installed packages to their most recent version and would remove or install additional packages as necessary. `full-upgrade` is less conservative than `safe-upgrade` and is much more likely to perform unwanted actions. However, it is capable of upgrading packages that `safe-upgrade` cannot. Because these sorts of situations are much more common between releases, using `full-upgrade` became the recommended course for upgrading between releases. However, neither method is supported anymore.

In current releases of Ubuntu, the correct way to upgrade systems is with the `do-release-upgrade` program. `do-release-upgrade` is a script that automates the process described above in addition to handling a number of corner cases and exceptions intelligently. It is the supported way to upgrade one's Ubuntu server.

Mirroring a System

One common task many system administrators want to accomplish is to mirror the installed software from one machine to another. Because all software on a default Ubuntu system is installed in packages, the packaging system can make this easy. Using `dpkg`, one can get a list of all packages on the machine with the following command:

```
# dpkg --get-selections > package_list
```

This will output a simple list of packages and then redirect that output into a file called package_list. I can copy this file to another machine and then use it to set the list of installable packages with the following command:

```
# dpkg --set-selections < package_list
```

Finally, I can install those selections onto the target system using the following command:

```
# apt-get dselect-upgrade
```

`dselect-upgrade` is a reference to APT's predecessor `dselect` but will simply work to upgrade packages on the system and install any new packages "marked" for upgrade by `dpkg --set-selections` in the process.

Making Your Own Packages

The power of a package management system is that you can track dependencies and conflicts, do automatic upgrades, and keep track of every file on the system and which piece of software it belongs to. Installing through packages is much easier than if one simply downloads and builds from scratch, but the package management system truly shines when it comes time to uninstall or upgrade. If you've installed from source, files may be in any number of places on your file system. If you've installed from a package, removing your package will be as simple as `apt-get remove`.

As a result, many responsible system administrators find it very convenient to ensure that *all* software on their systems is installed from packages. That sounds great, but sometimes a piece of software you want—or a version of a piece of software that you want—isn't packaged or isn't built for the version of Ubuntu that you are running. The result is that you'll need to build, in one way or another, your own packages. The rest of this chapter gives a brief overview of this process and provides a starting spot for the system administrator who wants to move beyond simply consuming packages and become a producer.

Rebuilding Packages

As I hinted earlier in this chapter, many users want to rebuild existing packages as part of "backporting" a version of a piece of software available

in one version of Ubuntu—or Debian—to a current one. Sometimes, if an ABI has changed, a piece of software won't work on a version of Ubuntu simply because it was compiled against a set of libraries that are no longer present. This is the easiest possible case to fix because adjusting for it is simply a matter of downloading the source and rebuilding it against the new version of the libraries. This section will cover doing exactly this.

Doing so will first require a source package. The source package, as you may remember from earlier in this chapter, consists of a DSC file and at least one other file. These can be downloaded as normal files from http://packages.ubuntu.com and unpacked with `dpkg-source -x` *`filename`*`.dsc`, or they can be installed automatically by using the `apt-get source` *`package`* command.

If one wanted to download and compile a package from a particular distribution—as is often the case—one could specify this explicitly with the `-t` option, which, behind the scenes, sets the default PIN for the distribution at a very high priority (990 in fact) by running (for example)

```
$ apt-get -t jaunty source --compile most
```

This would download and unpack the version of most source packages from Jaunty—assuming, of course, that the necessary `deb-src` line was included in /etc/apt/sources.list. The unpacked source code will be in a subdirectory of the current directory made up of the package name and version. In this case, the directory would be called most-5.0.0a since 5.0.0.a is the version of most that I've downloaded. By adding a `--compile` flag to the `apt-get` invocation above, the binary packages will also be built automatically—even if the program is in an interpreted language and there is no actual compiling taking place. If one does not use the compile flag, it can be invoked afterward in several ways. One of the simplest is by changing into the directory and then running `dpkg-buildpackage` like this:

```
$ cd most-5.0.0a
$ dpkg-buildpackage -us -uc -rfakeroot
```

This command will create an unsigned package (the `-us` and the `-uc` refer to unsigned source and unsigned changelog files) without needing root privileges (fakeroot is a program that allows packages to be built without

root). Of course, the package may also require build dependencies that are not installed by running a command in the following form:

```
# apt-get -t jaunty build-dep most
```

The `build-dep` subcommand to `apt-get` automates the process of installing all software necessary to build a given package. Running it is a frequent first step in rebuilding any package for the first time when that package is from an installed repository.

When the software in question is successfully rebuilt, the directory will contain a set of binary packages for this source package that end with .deb in the directory where it is run. In this case, the single binary package created was called most_5.0.0a-1_i386.deb. The -1 following the version number of the software refers to the version of the package and could be incremented each time we made a new version of the package. The i386 in this case simply refers to the architecture for which the binary package was built. In this case, I built it on an Intel machine. For many users, this will say amd64, which is an increasingly popular architecture. For most interpreted programs that will run on any architectures, this will say all.

New Upstream Versions

New upstream versions of packages are slightly more complicated than simply rebuilding an existing package with no modifications. Installing the package `devscripts` will provide the user with a program called `uupdate` which helps with this process. To use `uupdate`, a user first needs to download the source package with a command like `apt-get source most`. I don't want it to compile at the moment so I will leave off the compile option. Additionally, I will download the new upstream version tarball. There is no reason to unpack it at this point and, optionally, rename it into name-version.tar.gz format. Changing into the directory of the *old* package's source and running `uupdate` with the new upstream tarball as the argument will usually do the trick:

```
$ cd most-5.0.0a $ uupdate ../most-5.0.1.tar.gz
```

Usually, `uupdate` will then deduce the version number from the upstream tarball and apply all the changes made to the old version to the new

upstream source. If `uupdate` can't decode the version number, the new version number can be specified as a second argument to the command.

The output from `uupdate` should explain the process that it follows and will end with a description of the location of the new modified source. In this case, changing to ../most-5.0.1 will put me in the new "updated" package directory. It's a good idea to look around first to make sure that things worked well. Especially it is worth checking the debian/ subdirectory and paying attention to both the control file and the changelog file in that directory, the latter of which will have been updated automatically but will probably need a little bit of tweaking. The stanza at the top will include information on the new release and can be updated or tweaked to reflect changes that you made to the file. Once you are satisfied, you can build the package with `dpkg-buildpackage` in the way described in the previous section.

Building Packages from Scratch

Building packages from scratch is much more complicated and involves getting to know quite a bit about the internals of Debian packages. As a result, it is outside the scope of this chapter. As a hint, new packages can be most easily created using the package `dh-make`, which installs the program `dh_make`, which is invoked from inside the unpacked source tarball from the upstream developer. For many simple packages, `dh_make` does most of the hard work of creating workable packages.

Much more information on creating packages for Ubuntu can be found in the Ubuntu packaging guide, which goes in depth into the process of creating packages from scratch: https://wiki.ubuntu.com/PackagingGuide.

It is worth noting one important caveat to the Ubuntu documentation, which is that the packaging guide is focused on creating packages that are designed to be uploaded to Ubuntu. If you are creating packages that will be installed only on your own machine, the potential for harm is much less, and many of the guidelines in the packaging guide can be treated as just that—especially in the first version of a package. The difference is between workable packages and policy-compliant packages.

If you are going to proceed and create packages to be shared with others or perhaps even uploaded into the Ubuntu repositories eventually, it is a *very* good idea to follow the instructions in the packaging guidelines carefully and to use programs like lintian, which will check your packages for many common errors—useful steps in any situation. If you just want things to work, a brief trip through the guide and use of `dh_make` will probably put you in good enough shape to get by.

Hosting Your Own Packages

A final step in the creation of your packages will be hosting them in a place where others can get them in the simple "add a line to your source.list file" sort of manner to which I have referred throughout this chapter. There are several different ways to do this. The easiest one and the one most commonly practiced in the Ubuntu world is to use Launchpad—the infrastructure built by Canonical and used extensively in Ubuntu's own development—to host what's called a Personal Package Archives.

With a PPA, a developer can simply upload a source package to Launchpad and the package will then be built on a variety of architectures and posted into a PPA. PPAs work exactly the same way that developing for Ubuntu does, so using them is a great preview of what you will experience if you decide to eventually upload your software in Ubuntu and get involved on the development side of things. Earlier, when I showed how to add Bazaar packages to the list of packages, I entered the list of the Bazaar PPAs. More information on PPAs is available at the following URLs: https://help.launchpad.net/Packaging/PPA and https://launchpad.net/ubuntu/+ppas.

Alternatively, you can host your own repository on your own server with any of several different tools. Although the classic tool for running these is a package called `apt-ftparchive`, the newer project `reprepro` is probably a better fit. Installing the package with that name and looking in the documentation is a good way to get started.

Index

B

C

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z