

Linux for Networking Professionals

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Early Access Publication: Linux for Networking Professionals

Early Access Production Reference: B22171

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83588-110-1

www.packtpub.com

Table of Contents

[Linux for Networking Professionals: Strengthen your networking and security efforts with Linux](#)

1. [1 Welcome to The Linux Family](#)
 - I. [Join our book community on Discord!](#)
 - II. [Why is Linux a Good Fit for a Networking Team?](#)
 - i. [Why is Linux Important?](#)
 - ii. [The History of Linux](#)
 - III. [Mainstream Linux datacenter](#)
 - i. [Redhat](#)
 - ii. [Oracle/Scientific Linux](#)
 - iii. [SUSE](#)
 - iv. [Ubuntu](#)
 - v. [Running Linux on Windows](#)
 - vi. [BSD / FreeBSD / OpenBSD](#)
 - IV. [Specialty Linux Distributions](#)
 - i. [Open Source NAS and SAN providers](#)
 - ii. [Open Source Firewalls](#)
 - iii. [BusyBox Linux](#)
 - iv. [Kali Linux](#)
 - v. [SIFT](#)
 - vi. [Security Onion](#)
 - V. [Understanding virtualization](#)
 - i. [Linux and Cloud Computing](#)
 - VI. [Picking a Linux Distribution for Your Organization](#)
 - VII. [Summary](#)
 - VIII. [Further Reading](#)
2. [2 Basic Linux Network Configuration and Operations – Working with Local Interfaces](#)
 - I. [Join our book community on Discord!](#)
 - II. [Technical Requirements](#)
 - III. [Working with your Network Settings - Two Sets of Commands](#)
 - IV. [Displaying Interface IP Information](#)
 - i. [Displaying Routing Information](#)

- V. [IPv4 Addresses and Subnet Masks](#)
 - i. [Special Purpose Addresses](#)
 - ii. [Private Addresses – RFC 1918](#)
- VI. [Assigning an IP Address to an Interface](#)
 - i. [Adding a Route](#)
 - ii. [Adding a route using Legacy Approaches](#)
 - iii. [Disabling and Enabling an Interface](#)
 - iv. [Setting the MTU on an Interface](#)
 - v. [More on the nmcli Command](#)
- VII. [Summary](#)
- VIII. [Questions and answers](#)
- IX. [Further Reading](#)
- 3. [Using Linux and Linux Tools for Network Diagnostics](#)
 - I. [Join our book community on Discord!](#)
 - II. [Technical requirements](#)
 - III. [Network basics – the OSI model](#)
 - IV. [Layer 2 – relating IP and MAC addresses using ARP](#)
 - i. [MAC address OUI values](#)
 - V. [Layer 4 – how TCP and UDP ports work](#)
 - i. [Layer 4 – TCP and the three-way handshake](#)
 - VI. [Local port enumeration – what am I connected to? What am I listening for?](#)
 - VII. [Remote port enumeration using native tools](#)
 - VIII. [Remote port and service enumeration – Nmap](#)
 - i. [Nmap scripts](#)
 - ii. [Are there limits to Nmap?](#)
 - IX. [Wireless diagnostic operations](#)
 - X. [Summary](#)
 - XI. [Questions](#)
 - XII. [Further reading](#)

Linux for Networking Professionals: Strengthen your networking and security efforts with Linux

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: Welcome to The Linux Family
2. Chapter 2: Basic Linux Network Configuration and Operations – Working with Local Interfaces
3. Chapter 3: Using Linux and Linux Tools for Network Diagnostics

1 Welcome to The Linux Family

Join our book community on Discord!



<https://packt.link/mqfS2>

This book explores the Linux Platform and various Linux-based Operating Systems, in particular how Linux can work well for Networking Services. We'll start by discussing some of the history of the Operating System, then move quickly into basic configuration and troubleshooting. From there we'll work through building various network-related services on Linux that you might commonly see in most organizations. As we progress, we'll be building real services on real hosts, with an emphasis on securing and troubleshooting each service as we go. By the time we're done, you should be familiar enough with each of the services to start implementing some or all of them in your own organization. As they say: "*Every journey begins with a single step*", so let's take that step and start with a general discussion of the Linux platform. This is the second edition of this book, and for good reason. Linux is an ever changing ecosystem, and things have certainly changed since the first edition! Some of the tools we highlighted in the first book have changed significantly, for instance the wireless tool Kismet was completely re-written. Some basic system mainstays are in the process of being retired – the ISC DHCP server that was in the first edition is now deprecated, so we're looking at the new server in this book. Finally, we added complete new chapters on how virtualization can be leveraged by someone who works in networking. In this chapter, we'll start our journey by exploring Linux as a family of operating systems. The various distributions

of Linux are all related, but each is unique in its own way, with different strengths and features. We'll cover the following topics:

- Why Linux is a Good Fit for a Networking Team
- Mainstream Datacenter Linux
- Specialty Linux Distributions
- Virtualization
- Picking a Linux Distribution for your Organization

Why is Linux a Good Fit for a Networking Team?

In this book, we'll explore how to support and troubleshoot your network using Linux and Linux based tools, as well as how to securely deploy common networking infrastructure on Linux Platforms. Why would you want to use Linux for these purposes? To begin with, the architecture, history, and culture of Linux *steers* administrators towards scripting and automating processes. While carrying this to extremes can get people into funny situations, scripting routine tasks can be a real time-saver. In fact, scripting non-routine tasks, for instance, something that needs doing once per year, can be a life-saver as well – it means that administrators don't need to re-learn how to do that thing they did 12 months ago. Scripting routine tasks is even a bigger win. Over many years, Windows Administrators have learned that doing one task hundreds of times in a **Graphical User Interface (GUI)** is a guarantee to mis-click at least a few times. Scripting tasks like that, on the other hand is a guarantee of consistent results. Not only that, but over a network, where routinely administrators perform operations for hundreds or thousands of stations, scripting is often the only way to accomplish tasks at larger scales. Another reason that network administrators prefer Linux platforms is that Linux (and before that Unix) has been around since there were networks of which to be a part. On the server side, Linux (or Unix) services are what defined those services, where the matching Windows services are mere copies that have grown over time mainly to feature parity. On the workstation side, if you need a tool to administer or diagnose something on your network, it's probably already installed. If the tool that you seek isn't installed, it's a one line command to get it installed and running, along with any other tools, libraries or other dependencies required. And adding that tool did not require a license fee -

both Linux and any tools installed on Linux are almost without exception free and open source. Lastly, on both the server and desktop side, Linux historically has been free. Even now, when for-profit companies have License fees for some of the main supported distributions (for instance, Red Hat or Suse), those companies offer free versions of those distributions. Red Hat offers Fedora Linux and CentOS, both of which are free and to one extent or another act as test-bed versions for new features in Red Hat Enterprise Linux. OpenSUSE (free) and SUSE Linux (chargeable) are also very similar, with the SUSE distro being more rigorously tested and seeing a more regular cadence for version upgrades. The enterprise versions typically are term-licensed, that license granting the customer access to technical support and in many cases to OS updates. Many companies do opt for the licensed **enterprise ready** versions of the OS, but many other companies choose to build their infrastructures on free versions of OpenSuse, CentOS, or Ubuntu. The availability of free versions of Linux means that many organizations can operate with substantially lower IT costs, which has very much influenced where we have gone as an industry.

Why is Linux Important?

Over the years, one of the jokes in the information technology community is that next year was always going to be "*the year of the Linux Desktop*" – where we'd all stop paying license fees for desktops and business applications, and everything would be free and open source. Instead, what has happened is that Linux has been making steady inroads into the server and infrastructure side of many environments. Linux has become a mainstay in most data centers, even if those organizations think they are a *Windows only* environment. Many infrastructure components run Linux under the covers, with a nice web front-end to turn it into a vendor solution. If you have a **Storage Area Network (SAN)**, it likely runs Linux, as do your **Load Balancers, Access Points and Wireless Controllers**. Many **Routers** and **Switches** run Linux, as do pretty much all of the new *Software Defined Networking* solutions. Almost without fail, Information Security products are based on Linux. Traditional Firewalls and *Next Generation* Firewalls, **Intrusion Detection and Prevention Systems (IDS/IPS)**, **Security Information and Event Management (SIEM)** systems, and Logging

servers – Linux, Linux, Linux! Why is Linux so pervasive? There are many reasons:

- It's a mature operating system.
- It has an integrated patching and updating system.
- The basic features are simple to configure. The more complex features on the Operating System can be more difficult to configure than in Windows though, look ahead to our chapter on DNS or DHCP for instance.
- On the other hand, many features that might be for-sale products in a Windows environment are free to install on Linux.
- Since Linux is almost entirely file-based, it's fairly easy to keep it to a known baseline if you are a vendor basing their product on Linux.
- You can build just about anything on top of Linux, given the right mix of (free and open source) packages, some scripting and maybe some custom coding.
- If you pick the right distribution, the OS itself is free, which is a great motivator for a vendor trying to maximize profit or a customer trying to reduce their costs.

If the new **Infrastructure as Code** movement is what draws you, then you'll find that pretty much every coding language is represented in Linux and is seeing active development – from new languages like **Go** and **Rust**, all the way back to **Fortran** and **Cobol**. Even **PowerShell** and **.Net**, which grew out of Windows, are completely supported on Linux. Most infrastructure orchestration engines (for instance, **Ansible**, **Puppet**, or **Terraform**) started on and supported Linux first. On the Cloud side of today's IT infrastructure, the fact that Linux is free has seen the Cloud Service Providers push their clients towards that end of the spectrum almost from the start. If you've subscribed to any cloud service that is described as *Serverless* or *as a Service*, likely behind the scenes that solution is almost always a Linux instance or shared instance. Finally, as we've seen the server and infrastructure side of IT move towards Linux, today's cell phones are steadily becoming the largest *desktop* platform in today's computing reality. In today's world, cell phones are generally either iPhones or Android based, both of which are (you guessed it) Unix/Linux based! So the "*Year of the Linux Desktop*" has snuck up on us by changing the definition of

desktop. All of this combines to make Linux very important to today's Networking or IT professionals. This book focuses on using Linux both as a desktop *toolbox* for the Networking Professional, as well as securely configuring and delivering various network services on a Linux platform.

The History of Linux

To understand the origins of Linux, we have to first discuss the origins of Unix. Unix was developed in the late 1960's and early 1970's at Bell Labs. Dennis Ritchie and Ken Thompson were Unix's main developers. The name Unix was actually a pun on the name of *Multics*, an earlier operating system which inspired many of the Unix features. In 1983, Richard Stallman and the Free Software Foundation started the GNU (a recursive acronym - GNU's Not Unix) project, which aspired to create a Unix-like operating system available to all for free. Out of this effort came *GNU Hurd* kernel, which most would consider the precursor to today's Linux versions (the SFS would prefer we called them all GNU/Linux). In 1992, Linus Torvalds released Linux, the first full released a fully realized GNU kernel. It's important to note that mainstream Linux is normally considered to be a kernel that can be used to create an operating system, rather than an operating system on its own. Linux is still maintained with Linus Torvalds as the lead developer, but today there is a much larger team of individuals and corporations acting as contributors. So while technical Linux only refers to the kernel, in the industry "Linux" generally refers to any of the Operating Systems that are built upon that kernel. Since the 1970's, hundreds of separate flavors of Linux have been released. Each of these is commonly called a *distribution* (or *distro* for short). These are each based on the Linux kernel of the day, along with an installation infrastructure and a repository system for the OS and for updates. Most are unique in some way, either in the mix of base packages or the focus of the distro – some might be small in size to fit on smaller hardware platforms, some might focus on security, and some might be intended as a general purpose enterprise *workhorse* operating system. Some distros have been *mainstream* for a period of time, and some have waned in popularity as time has gone by. The thing they all share is the Linux kernel, which they have each built upon to create their own distribution. Many distro's have based their operating system on another distro, customizing that enough to justify

calling their implementation a new distribution. This trend has given us the idea of a *Linux Family Tree* – where dozens of distributions can grow from a common *root* - this is nicely explored on the *distrowatch* website, at <https://distrowatch.com/dwres.php?resource=family-tree>. An alternative to Linux, especially in the Intel/AMD/ARM hardware space, is **BSD** (which stands for **Berkeley Software Distribution**) Unix. BSD Unix is a descendent of the original **Bell Labs Unix**, it is not based on Linux at all. However, BSD and many of its derivatives are still free, and share many characteristics (and a fair amount of code) with Linux. To this day, the emphasis of both Linux and BSD Unix is that both are freely available operating systems. While commercial versions and derivatives are certainly available, almost all of those commercial versions have matching free versions. In this section, we looked at both the history and importance of Linux in the computing space. We've understood how Linux emerged and how it found popularity in certain sections of the computing landscape. Now, we'll start to look at the different versions of Linux that are available to us so that we can build on the information we need to make choices for which distro to use later in the chapter.

Mainstream Linux datacenter

As we've discussed, Linux is not a monolithic thing, but rather a varied or even splintered ecosystem of different distributions. Each Linux distribution are all based on the same GNU/Linux kernel, but then are packaged by groups with different goals and philosophies, making for a wide variety of choices when an organization wants to start standardizing on their server and workstation platforms. The main distributions that we commonly see in modern datacenters are **Redhat**, **SUSE** or **Ubuntu**, with **FreeBSD** Unix being another alternative (albeit much less popular now than in the past). This is not to say that other distributions don't crop up on desktops or datacenters, but these are the ones you'll see most often. These all have both desktop and server versions – the server versions often being more stripped down, with the office productivity, media tools and often the GUI removed.

Redhat

Redhat has recently been acquired by IBM (in 2019), but still maintains Fedora as one of its main projects. Fedora has both server and desktop versions, and remains freely available. The commercial version of Fedora is **Redhat Enterprise Linux (RHEL)**. RHEL is commercially licensed and has a formal support channel. CentOS started as a free, community supported version of Linux that was functionally compatible with the Redhat Enterprise version. This made it very popular for server implementations in many organizations. In January 2014, Redhat pulled CentOS into its fold, becoming a formal sponsor of the distro. In late 2020, it was announced that CentOS would no longer be maintained as a RHEL compatible distribution, but would rather fit somewhere between Fedora and RHEL – not so new as to be bleeding edge, but not as stable as RHEL either. As part of this change, CentOS was renamed **CentOS Stream**. The final picture is that Fedora is the distro that has the latest features and code, where new features get tried and tested. The CentOS Stream distro is more stable, but is still upstream of RHEL. RHEL is the stable, fully tested operating system with formal support offerings.

Oracle/Scientific Linux

Oracle/Scientific Linux is also seen in many datacenters (and in Oracle's cloud offerings). Oracle Linux is based on Redhat, and in fact advertise their product as fully compatible with Redhat Enterprise Linux. Oracle Linux is free to download and use, but support from Oracle is subscription based.

SUSE

OpenSUSE is the community distribution that SUSE Linux is based on, similar to how Redhat Enterprise Linux is based on Fedora. **SUSE Linux Enterprise Server (SLES)** was in the early days of Linux the mainly European competitor for the US based Redhat distribution. Those days are past however, and SUSE Linux is (almost) as likely to be found in Indiana as it is in Italy in modern datacenters. Similar to the relationship between Redhat and CentOS, SUSE maintains both a Desktop and a Server version. In addition, they also maintain a *High Performance* version of the OS, which comes with optimizations and tools pre-installed for parallel

computing. OpenSUSE occupies an "upstream" position to SLES, where changes can be introduced in a distro that is somewhat more "forgiving" to changes that might not always work out the first time. The OpenSUSE Tumbleweed distro has the newest features and versions, where OpenSUSE Leap is closer in versioning and stability to the SLE versions of the Operating System. It is no accident that this model is similar to the Redhat family of distros.

Ubuntu

Ubuntu Linux is maintained by Canonical, and is freely downloadable, with no separate commercial or "upstream" options. It is based on Debian, and has a unique release cycle. New versions of both the server and desktop versions are released every 6 months. A **Long Term Support (LTS)** version is released every two years, with support for LTS versions of both the server and desktop running for 5 years from the release date. As with the other larger players, support is subscription based, though free support from the community is a viable option as well. As you would expect, the server version of Ubuntu is focused more on the core OS, network and datacenter services. The GUI is often de-selected during the installation of the server version. The desktop version however has several packages installed for office productivity, media creation and conversion, as well as some simple games.

Running Linux on Windows

Microsoft has for many years had a set of executables that duplicated many of the common Unix/Linux command line tools. This has been expanded in Windows 10 and 11 to a full Linux shell and distribution called **WSL** (Windows Services for Linux). The default distribution for this install is Ubuntu, but other distributions can certainly be installed (OpenSUSE, any of the Redhat variants, Kali, Debian, Arch etc). While much more complete than in previous iterations, the current WSL is still a work in progress. Most things work as expected, but it is not difficult to find things that work differently within WSL as compared to a native installation of the same distribution.

BSD / FreeBSD / OpenBSD

As discussed, the *BSD Tree* of the family is derived from Unix rather than from the Linux kernel, but there is lots of shared code, especially once you look at the packages that aren't part of the kernel. FreeBSD and OpenBSD were historically viewed as more secure than the earlier versions of Linux. Because of this, many firewalls and network appliances were built based on the BSD OS family, and remain on this OS to this day. One of the more visible BSD variants is Apple's commercial operating system **OS X** (now **MacOS**), which is based on Darwin, which is in turn a fork of BSD. As time marched on however, Linux has grown into most of the same security capabilities as BSD, until it became a difference in, for any particular setting, BSD perhaps having the more secure default setting than most Linux alternatives. Linux now has security modules available that significantly increase its security posture. **SELinux** or **AppArmor** are the two main options available. SELinux grew out of the Redhat distros, and is fully implemented for SUSE, Debian, and Ubuntu as well. AppArmor is typically viewed as a simpler-to-implement option, with many (but not all) of the same features. AppArmor is available on Ubuntu, Suse and most other distros (with the notable exception of RHEL). Both options take a policy-based approach to significantly increase the overall security posture of the OS they are installed on. With the evolution of Linux to be more security-focused, in particular with SELinux or AppArmor available (and recommended) for most modern Linux distributions, the *more secure* argument of BSD vs Linux is now mainly a historic perception rather than fact. In addition to these mainstream Linux / Unix distributions, there are a myriad of specialty distros. Since most of these specialty distros are special-purpose versions of one or another of the mainstream options, thinking of this relationship as extended families can really help keep things straight.

Specialty Linux Distributions

Aside from the mainstream Linux Distributions, there are several distros that have been purpose-built for a specific set of requirements. They are all built on a more mainstream distro, but are tailored to fit a specific set of

needs. We'll describe a few here that you are most likely to see or use as a network professional.

Open Source NAS and SAN providers

Most commercial NAS and SAN providers are based on Linux or BSD. The front runner on open source NAS/SAN services at the time of this writing seems to be **TrueNAS** (formerly **FreeNAS**) and **XigmaNAS** (formerly **NAS4Free**). Both have free and commercial offerings. These distros are general purpose storage that a networking professional might use, or on the other hand that you may need to provision network resources for. Other offerings in this space focus on sharing music or video media, which of course you'd see less of in most organizational settings.

Open Source Firewalls

Networking and Security companies offer a wide variety of Firewall appliances, most of which are based on Linux or BSD. Many companies do offer free firewalls, some of the more popular being **pfSense** (free versions and pre-built hardware solutions available), **OpnSense** (freely available, with donations) and **Untangle** (which also has a commercial version). **Smoothwall** is another alternative, with both free and commercial versions available. In this book we'll explore using the on-board firewall in Linux to secure individual servers, or to secure a network perimeter.

BusyBox Linux

While FreeBSD serves as the underlying OS for many commercial appliances, that still makes for a fully featured OS. Busybox is a much more stripped down, small footprint operating system. Not all Linux commands are available, and some are limited or modified. This makes BusyBox attractive to many appliance vendors, and it has become more and more popular as a foundation for other products. Of particular note, VMware chose Busybox to act as its ESXi command line in all versions starting from version 4.0 .

Kali Linux

Descended from **Backtrack**, and **Knoppix** before that, **Kali Linux** is a distribution based on Debian that is focused on Information Security. The underlying goal of this distribution is to collect as many useful Penetration Testing and Ethical Hacking tools as possible on one platform, and ensure that they all work reliably without interfering with each other. The newer versions of the distribution have focused on maintaining this tool-interoperability as the OS and tools get updated (using the `apt` toolset).

SIFT

SIFT is a distribution authored by the forensics team at the SANS institute, focused on digital forensics and incident response tools and investigations. Similar to Kali, the goal of SIFT is to be a "one stop shop" for free/open source tools in one field – **Digital Forensics and Incident Response (DFIR)**. Historically this was a distribution based on Ubuntu, but in recent years this has changed – SIFT is now also distributed as a script that installs the tools on Ubuntu Desktop or Windows Services for Linux (which is Ubuntu based).

Security Onion

Security Onion is also similar to Kali Linux in that it contains several information security tools, but its focus is more from the defender's point of view. This distribution centers on threat hunting, network security monitoring, and log management. Tools included in this distribution include Suricata, Zeek, and Wazuh just to name a few. Security Onion is unique in that you normally would not run all available tools within the same installation, if only because of resource utilization. Even in a moderately sized network, for instance, it would be highly recommended to run your Suricata IPS (Intrusion Prevention System) on a dedicated host, and your Zeek monitoring instance on a different dedicated host. Aside from direct network applications though, where else might you see Linux? As we discussed in the section on BusyBox Linux, any ESXi Hypervisor servers in your infrastructure are definitely Linux based. It's also likely that many of the guests VMs within your virtual infrastructure will also be Linux installs or Linux based virtual appliances.

Understanding virtualization

Virtualization has played a major role in the changes in IT since the mid-2000's, and Linux plays a huge part in that. Virtualization allows one physical hosts to run multiple virtual machines, each with their own independent operating system and virtual hardware. Not only are most hypervisors Linux based, but the hypervisor concept allows the ability to work with multiple distributions at once, all on the same hardware. With a local Hypervisor, a network professional is able to run dozens of different machines on their laptop, desktop or server computers. While **VMware** was the pioneer in this space (desktop and dedicated virtualization), they have since been joined by **XEN**, **KVM**, **VirtualBox**, and **QEMU**, just to name a few. The VMware **ESXi** product has historically been available as a free install, which has led many IT professionals to install ESXi in their home networks. This changed recently (early 2024, after the Broadcom acquisition), and this free version is no longer available. This change has made distributions like **Proxmox** (based on Debian GNU/Linux) much more popular for home and lab networks. We'll discuss Proxmox in more depth later in this book.

Linux and Cloud Computing

The increasing stability of Linux and the fact that Virtualization is now mainstream has in many ways made our modern-day Cloud ecosystems possible. Add to this the increasing capabilities of automation in deploying and maintaining back end infrastructure and the sophistication available to developers of web applications and APIs (Application Programming Interfaces), and what we have possible is the Cloud infrastructures of today. Key features of this are:

- A multi-tenant infrastructure, where each customer maintains their own instances (virtual servers and virtual datacenters) in the cloud
- Granular costing either by month or more commonly by resources used over time
- Reliability that is as good or better than many modern datacenters (though recent outages have shown what happens when we put too many eggs in the same basket).

- APIs that make automating your infrastructure relatively easy, so much so that for many companies provisioning and maintaining their infrastructure has become a coding activity (often called **Infrastructure as Code**).
- These APIs make it possible to scale up (or down) on capacity as needed, whether that is storage, computing, memory, session counts, or all four.

Cloud services are in business for a profit though, any company that has decided to "forklift" their datacenter as-is to a cloud service has likely found that all those small charges add up over time, eventually reaching or surpassing what the costs of their on-premise datacenter. It's still often attractive on the dollars side, as those dollars are spent on operational expenses that can be directly attributed more easily than the on-premise capital expenditure model (commonly called Cap-Ex vs Op-Ex models). As you can see from what defines a cloud service, moving a datacenter to a cloud service does bring lots of benefits to an organization that likely wouldn't be options in the on-premise model. This only becomes more apparent as more cloud-only features are utilized. With all that background covered, how then should we pick what flavours or distros to run within our organization? Is there a best choice?

Picking a Linux Distribution for Your Organization

In many ways, which distribution you select for your datacenter is not important – the main distributions all have similar functions, often have identical components, and have similar vendor or community support options. However, because of the differences between the distros, what is important is that one distribution (or a set of similar distros) is selected. The desired outcome is that your organization standardizes one distribution that your team can develop their expertise with. This also means that you can work with the same escalation team for more advanced support and troubleshooting, whether that is a consulting organization, a paid vendor support team, or a group of like-minded individuals in various internet forums. Many organizations purchase support contracts with one of *the big*

three (Redhat, SUSE, or Canonical, depending on their distribution). Where you don't want to be is the situation I've seen a few clients end up in. Having hired a person who is eager to learn, a year later they found that each of the servers built that year were on a different Linux distribution, each built slightly differently. This is a short road to your infrastructure becoming the proverbial science experiment that never ends! A common proverb in designing networks and infrastructure is "*always think of the person who will replace you, because when you move you will be that person*". What this boils down to is to standardize as much as possible, so that new members of your team only have to learn things once. It's understood that dedicated hosts such as firewalls and packaged vendor solutions will likely vary from your standard, but there's often nothing you can do about that. Contrast this with another client – their first server was a **SUSE Linux for SAP**, which is as the name indicates, a SUSE Linux Server, packaged with the SAP application that the client purchased (SAP HANA). As their Linux footprint grew with more services, they stuck with the SUSE platform, but went with the real SLES distribution. This kept them on a single Operating System, and equally important for them, a single support license with SUSE. They were able to focus their training and expertise on SUSE. Another key benefit for them was that as they added more servers, they were able to apply a single stream of updates and patches in a phased approach. In each patch cycle, less critical servers got patched first, leaving the core business application servers to be patched a few days later, after their testing was complete. With one distribution in play, this is a single test / deploy process that can be completed fairly quickly, rather than a separate process for each distribution that might be installed. The main advice in picking a distribution is to stick to one of the larger distributions. If you have desktop or laptop Linux installs, try to keep the servers and the workstations using the same distribution. If people on your team have strong feelings about one of another of these, then definitely take that into consideration. You will likely want to stay fairly close to one of the mainstream distributions for use within an organization, something that is regularly maintained and has a paid subscription model available for support – even if you don't feel you need paid support today, that may not always be the case.

Summary

Now that we've discussed the history of Linux, along with several of the main distributions, I hope you are in a better position to appreciate the history and the central importance of Operating System in our society. In particular, I hope that you have some good criteria to help you choose a distro for your infrastructure. In this book, we've chosen Ubuntu as our example distribution. It's a free distribution, which in its LTS version has an OS that we can depend on being supported as you work through the various scenarios, builds, and examples that we'll discuss. It's also the distribution that is the default install for WSL (Windows Services for Linux). This makes it an easy distro to get familiar with, even if you don't have server or workstation hardware to spare or even a virtualization platform to test with. In our next chapter, we'll discuss getting your Linux Server or Workstation on the network. We'll illustrate working with the local interfaces, adding IP addresses, subnet masks, and any routes required to get your Linux host working in a new or an existing network.

Further Reading

- Redhat Linux: <https://www.redhat.com/>
- Fedora: <https://getfedora.org/>
- CentOS: <https://www.centos.org/>
- SUSE Linux: <https://www.suse.com/>
- OpenSuse: <https://www.opensuse.org/>
- Ubuntu Linux: <https://ubuntu.com/>
- FreeBSD Unix: <https://www.freebsd.org/>
- OpenBSD Unix: <https://www.openbsd.org/>
- Proxmox Linux: <https://www.proxmox.com/>
- Busybox Linux: <https://www.proxmox.com/>
- Windows Services for Linux: <https://learn.microsoft.com/en-us/windows/wsl/>
- Linux / BSD Differences: <https://www.howtogeek.com/190773/htg-explains-whats-the-difference-between-linux-and-bsd/>
- TrueNAS: <https://www.truenas.com/>
- XigmaNAS: <https://www.xigmanas.com/>
- pfSense: <https://www.pfsense.org/>
- OpnSense: <https://opnsense.org/>
- Untangle: <https://www.untangle.com/untangle>

- Smoothwall: <https://smoothwall.com/>
- Kali Linux: <https://www.kali.org/>
- SIFT: <https://digital-forensics.sans.org/community/downloads>;
<https://www.sans.org/webcasts/started-sift-workstation-106375>
- Security Onion: <https://securityonionsolutions.com/software>
- Kali Linux: <https://www.kali.org/>

2 Basic Linux Network Configuration and Operations – Working with Local Interfaces

Join our book community on Discord!



<https://packt.link/mqfS2>

In this chapter, we'll explore how to display and configure the local interfaces and routes on your Linux host. As much as possible we'll discuss both the newer and legacy commands for performing these operations. This will include displaying and modifying IP addressing, local routes, and other interface parameters. Along the way we'll discuss how IP addresses and Subnet addresses are constructed, using a binary approach. This chapter should give you some solid ground to stand on as we progress in further chapters, troubleshooting networking problems, hardening our host, and installing secure services. The topics covered in this chapter are:

- Working with your Network Settings - Two Sets of Commands
- Displaying Interface IP Information
- IPv4 Addresses and Subnet Masks
- Assigning an IP Address to an Interface

Technical Requirements

In this and every other chapter, as we discuss various commands, you are encouraged to try them on your own computer. The commands in this book are all illustrated on Ubuntu Linux, version 22 (a *Long Term Support* version) but should for the most part be identical or very similar on almost any Linux distribution.

Working with your Network Settings - Two Sets of Commands

For most of the Linux lifespan that people are familiar with, the `ifconfig` (Interface config) and related commands have been a mainstay of the Linux Operating System, so much so that even now that it's deprecated in most distributions, it still *rolls off the fingers* of many system and network administrators. Why were these older network commands replaced? There are several reasons actually. Some newer hardware (in particular, Infiniband network adapters) are not well supported by the older style commands. In addition, as the Linux kernel has changed over the years, the operation of the older commands has become less and less consistent over time, but pressure around backward compatibility made resolving this difficult – for instance changing command line switches for `ifconfig` could potentially break all kinds of things. The older commands are in the `net-tools` software package, and the newer commands are in the `iproute2` software package. Most newer Linux distributions have `iproute2` installed by default, but have the option available to install the older `net-tools` package. New administrators should focus on the newer commands, but having familiarity with the older commands is still a good thing to maintain. It's still very common to find older computers running Linux, older machines that might never be updated that still use the older style commands. For this reason, we'll cover both toolsets. The lesson to be learned from this is that in the Linux world, change is constant. There is always something new that you should or must learn. In this case, the old commands are still available but are not installed by default. To install the legacy commands:

```
robv@ubuntu:~$ sudo apt install net-tools
[sudo] password for robv:
Reading package lists... Done
```

```

Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libfprint-2-tod1
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  net-tools
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 0 B/196 kB of archives.
After this operation, 864 kB of additional disk space will be used.
Selecting previously unselected package net-tools.
(Reading database ... 183312 files and directories currently installed.)
Preparing to unpack .../net-tools_1.60+git20180626.aebd88e-1ubuntu1_amd64.deb ...
Unpacking net-tools (1.60+git20180626.aebd88e-1ubuntu1) ...
Setting up net-tools (1.60+git20180626.aebd88e-1ubuntu1) ...
Processing triggers for man-db (2.9.1-1) ...

```

You'll want to notice a few things in this install command and its output:

- **sudo:** The `sudo` command was used – `sudo` essentially means "*do as the super user*" – the command executes with root (administrator) privileges. This needs to be paired with the password of the user executing the command. In addition, that user needs to be properly entered in the configuration file `/etc/sudoers`. By default, in most distributions, the `userid` defined during the install of the Operating System is automatically included in that file. Additional users or groups can be added using the command `visudo`.

Why was `sudo` used? Installing software or changing network parameters and many other system operations require elevated rights – on a multi-user corporate system, you wouldn't normally want people who weren't administrators to be making these changes. So, if `sudo` is so great, why don't we run everything as root? Mainly this is a security issue. Of course, everything will work if you have root privileges. However, any mistakes and typos can have disastrous results. Also, if you are running with the right privileges and happen to execute some malware, now the malware has those same privileges – which is certainly less than ideal! If anyone asks, yes, Linux malware definitely exists, and has sadly been with the OS almost from the start.

- **apt:** The `apt` command was used – `apt` stands for *Advanced Package Tool*, and installs not only the *package* requested, but also any required packages, libraries, or other dependencies required for that package to run. Not only that, but by default, it collects all of those components from online repositories (or *Repo's*). This is a welcome shortcut compared to the older process, where all the dependencies (at the correct versions) had to be manually collected, and then installed in the correct order to make any new features work.

`apt` is the default installer on Ubuntu, Debian, and related distros, but the package management application will vary between distributions. In addition to the `apt` and its equivalents, installing from downloaded files is still supported. Debian, Ubuntu, and related distros use `deb` files, while many other distributions use `rpm` files. This is summarized below:

Operating System File Format Installation Tool(s)

Debian	.deb	apt, apt-cache, apt-get, dpkg
Ubuntu	.deb	apt, apt-cache, apt-get, dpkg
Redhat / CentOS	.rpm	yum, rpm
SUSE	.rpm	zypper, rpm

Table 2.1: Common Distributions, Install Package Filetypes and Installation Tools So, now we have a boatload of new commands to look at, how do we get more information on these? The `man` (for manual) command has documentation for most commands and operations in Linux. The `man` command for `apt` for instance can be printed using the command `man apt` (output shown below):

```
APT(8)                                APT                                APT(8)

NAME
    apt - command-line interface

SYNOPSIS
    apt [-h] [-o=config string] [-c=config file] [-t=target release]
        [-a=architecture] {list | search | show | update |
        install pkg [{=pkg version number | /target release}]... |
        remove pkg... | upgrade | full-upgrade | edit-sources |
        {-v | --version} | {-h | --help}}
```

DESCRIPTION

apt provides a high-level commandline interface for the package management system. It is intended as an end user interface and enables some options better suited for interactive usage by default compared to more specialized APT tools like **apt-get(8)** and **apt-cache(8)**.

Much like **apt** itself, its **manpage** is intended as an end user interface and as such only mentions the most used commands and options partly to not duplicate information in multiple places and partly to avoid overwhelming readers with a cornucopia of options and details.

Manual page apt(8) line 1 (press h for help or q to quit)

Figure 2.1 – apt man Page

As we introduce new commands in this book, take a minute to review them using the `man` command – this book is meant more to guide you in your journey, not as a replacement for the actual Operating System documentation. Now that we've talked about the modern and legacy tools, and then installed the legacy net-tools commands, what are these commands, and what do they do?

Displaying Interface IP Information

Displaying interface information is a common task on a Linux workstation. This is especially true if your host adapter is set to be automatically configured, for instance using DHCP (Dynamic Host Configuration Protocol) or IPv6 Auto-Configuration. As we discussed, there are two sets of commands to do this. The `ip` command allows us to display or configure your host's network parameters on newer operating systems. On older versions, you will find that the `ifconfig` command is used. The `ip` command will allow us to display or update IP addresses, routing information, and other networking information. For instance, to display current IP address information, below is the command:

```
ip address
```

The `ip` command supports **command completion**, so `ip addr` or even `ip a` will give you the same results:

```
robv@ubuntu:~$ ip ad
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default c
    link/ether 00:0c:29:6b:14:e6 brd ff:ff:ff:ff:ff:ff
    altname enp2s1
    inet 192.168.229.128/24 brd 192.168.229.255 scope global dynamic noprefixroute ens33
```

```

    valid_lft 1702sec preferred_lft 1702sec
inet6 fe80::87e:cecb:ab2e:5dbe/64 scope link noprefixroute
    valid_lft forever preferred_lft forever

```

You'll see that even the simplest of commands will sometimes return much more information than you might want. For instance, you'll see both IP version 4 (IPv4) and IPv6 information returned – we can limit this to only version 4 or 6 by adding `-4` or `-6` to the command line options:

```

robv@ubuntu:~$ ip -4 ad
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default
    altname enp2s1
    inet 192.168.229.128/24 brd 192.168.229.255 scope global dynamic noprefixroute ens33
        valid_lft 1435sec preferred_lft 1435sec

```

In this output, you'll see that the `loopback` interface (a logical, internal interface) has an IP address of `127.0.0.1`, and the Ethernet interface `ens33` has an IP address of `192.168.229.128`. Now would be an excellent time to type `man ip` and review the various operations that we can do with this command:

```

IP(8)                                     Linux                                     IP(8)

NAME
    ip - show / manipulate routing, network devices, interfaces and tunnels

SYNOPSIS
    ip [ OPTIONS ] OBJECT { COMMAND | help }

    ip [ -force ] -batch filename

OBJECT := { link | address | addrlabel | route | rule | neigh | ntable
            | tunnel | tuntap | maddress | mroute | mrule | monitor | xfrm
            | netns | l2tp | tcp_metrics | token | macsec | vrf | mptcp |
            ioam }

OPTIONS := { -V[ersion] | -h[uman-readable] | -s[tatistics] |
              -d[etails] | -r[esolve] | -iec | -f[amily] { inet | inet6 |
              link } | -4 | -6 | -B | -0 | -l[oops] { maximum-addr-flush-at-
              tempts } | -o[neline] | -rc[vbuf] [size] | -t[imestamp] |
              -ts[hort] | -n[etns] name | -N[umeric] | -a[ll] | -c[olor] |
              -br[ief] | -j[son] | -p[retty] }

OPTIONS

```

Figure 2.2 – `ip` man page

The `ifconfig` command has very similar functions to the `ip` command, but as we noted, it is seen mostly on older versions of Linux. The legacy commands have all grown organically, with features bolted on as needed. This has landed us in a state where as more complex things are being displayed or configured, the syntax becomes less and less consistent. The more modern commands were designed from the ground up for consistency. Let's duplicate our efforts using the older/legacy command. To display the interface IP, just type `ifconfig`:

```

robv@ubuntu:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.229.128 netmask 255.255.255.0 broadcast 192.168.229.255
    inet6 fe80::87e:cecb:ab2e:5dbe prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:6b:14:e6 txqueuelen 1000 (Ethernet)
    RX packets 2848 bytes 2939902 (2.9 MB)

```

```

RX errors 1 dropped 1 overruns 0 frame 0
TX packets 1935 bytes 190857 (190.8 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
device interrupt 19 base 0x2000
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 257 bytes 24641 (24.6 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 257 bytes 24641 (24.6 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0 TX errors 0 dropped 0

```

As you can see, mostly the same information is displayed in a slightly different format. If you review the `man` page for both commands, you'll see that the options are more consistent in the `ip` command, and there isn't as much IPv6 support in `ifconfig` – for instance, natively you can't select an IPv4 or IPv6 only display.

Displaying Routing Information

In the modern network commands, we'll use the exact same `ip` command to display our routing information. And, as you'd expect, the command is `ip route`, which can be shortened to anything up to `ip r`:

```

robov@ubuntu:~$ ip route
default via 192.168.229.2 dev ens33 proto dhcp metric 100
169.254.0.0/16 dev ens33 scope link metric 1000
192.168.229.0/24 dev ens33 proto kernel scope link src 192.168.229.128 metric 100

```

From this output, we see that we have a *default route* pointed to `192.168.229.2`. The default route is just that – if a packet is being sent to a destination that isn't in the routing table, the host will send that packet to its default gateway. The routing table will always prefer the "most specific" route – the route that most closely matches the destination IP. If there is no match, then the most specific route goes to the default gateway, which routes for `0.0.0.0 0.0.0.0` (in other words, the if it doesn't match anything else route). The host assumes that the default gateway IP belongs to a *router*, which will (hopefully) then know where to send that packet to next. We also see a route to `169.254.0.0/16`. This is called a **Link-Local Address** as defined in the RFC 3927. **RFC** stands for **Request for Comment**, which serves as part of the informal peer review process that internet standards use as they are developed. The list of published RFCs is maintained by the **Internet Engineering Task Force (IETF)**, at <https://www.ietf.org/standards/rfcs/>. Link-Local Addresses only operate in the current subnet – if a host does not have a statically configured IP address, and DHCP does not assign an address, it will use the first two octets defined in the RFC (169.254), then compute the last two octets, semi-randomly assigning them. After a Ping/ARP test (we'll discuss ARP in Chapter 3) to ensure that this computed address is in fact available, the host is ready to communicate. This address is supposed to only communicate with other LLA addresses on the same network segment, typically using broadcast and multicast protocols such as ARP, Alljoyn and so on to "find" each other. Just for clarity, these addresses are almost never used on real networks, they're the address that gets used if there is absolutely no other alternative. And just for extra confusion, Microsoft calls these addresses something different – **Automatic Private Internet Protocol Addressing (APIPA)**. Finally we see a route to the local subnet, in this case `192.168.229.0/24`. This is called a **connected route** (since it's connected to that interface). This tells the host that no routing is needed to communicate with other hosts in its own subnet. This set of routes is very common in simple networks – a default gateway, a local segment and that's it. In many operating systems, you won't see the `169.254.0.0` subnet in your command output unless the host is actually using a Link-Local address. On the legacy side, there are multiple ways to show the current set of routes. The typical command is `netstat -rn` for *network status*, show routes, and numeric display. However, `route` is a command all to itself (we'll see why later on in this chapter):

```

robov@ubuntu:~$ netstat -rn
Kernel IP routing table
Destination      Gateway          Genmask         Flags   MSS Window  irtt Iface
0.0.0.0          192.168.229.2   0.0.0.0         UG      0 0          0 ens33
169.254.0.0      0.0.0.0         255.255.0.0     U        0 0          0 ens33
192.168.229.0    0.0.0.0         255.255.255.0   U        0 0          0 ens33
robov@ubuntu:~$ route -n
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface

```

0.0.0.0	192.168.122.1	0.0.0.0	UG	100	0	0 ens33
169.254.0.0	0.0.0.0	255.255.0.0	U	1000	0	0 ens33
192.168.122.0	0.0.0.0	255.255.255.0	U	100	0	0 ens33

These show the same information, but now we have two additional commands – `netstat` and `route`. The legacy set of network tools tends to have a separate, unique command for every purpose – and in this case we're seeing two of them with quite a bit of overlap. Knowing all of these commands and keeping their differing syntax straight can be a challenge for someone new to Linux, the `ip` set of commands makes this much simpler! No matter which set of tools you end up using, you now have the basics to establish and check IP addressing and routing, which between them will get you basic connectivity for your host.

IPv4 Addresses and Subnet Masks

In the previous section, we discussed IP addresses briefly, but let's discuss them in a bit more detail. What IP version 4 allows you to do is to address each device in a *subnet* uniquely, by assigning each device an address and a subnet mask. For instance, in our example, the IPv4 address is `192.168.122.18`. Each *octet* in an IPv4 address can range from 0-255, and the subnet mask is `/24`, which is also commonly represented as `255.255.255.0`. This seems complicated until we break things down to a binary representation. 255 in binary is `11111111` (8 bits), and 3 of those groupings makes 24 bits. So what our address and mask representation is saying is that, when masked, the network portion of the address is `192.168.122.0`, and the host portion of the address is `182`, and can range from 1-254. Breaking this down:

Address in Decimal	192	168	122	182
Address in Binary	11000000	10101000	01111010	10110110
Mask in Binary	11111111	11111111	11111111	00000000

Table 2.2 – IP Addresses and Binary Masking What if we needed a larger subnet? We can simply "slide" that mask over a few bits to the left. For instance, for a 20 bit subnet mask:

Address in Decimal	192	168	122	182
Address in Binary	11000000	10101000	01111010	10110110
Mask (in binary)	11111111	11111111	11110000	00000000

Table 2.3 – Adjusting the Subnet Mask Using Binary This makes the third octet of the mask to be `0b11110000` (note the shorthand `0b` for binary), which translates to `240` in decimal. This "*masks*" the third octet of the network to `0b01110000` or `112`. This increases the range of addresses for our hosts to 0-15 (0 – `0b1111`) in the third octet, and 0-255 (0 – `0b11111111`) in the fourth, or 3824 (15 x 255 – 1) in total (we'll get to the "-1" in the next section). You can see that having a calculator that does binary to decimal conversions is a handy thing for a networking professional! Be sure it does hexadecimal (base 16) as well, we'll dive into that in a few minutes as well. Now that we've got the knack of working with Addresses and Subnet Masks in decimal and especially binary, let's expand on that, and explore how that can be used to illustrate other addressing concepts.

Special Purpose Addresses

There are a few *special purpose* addresses that we'll need to cover to further explore how IP addresses work in a local subnet. First of all, if all the host *bits* in an address are set to 1, that is called the **broadcast** address. If you send information to the broadcast address, it is sent to and read by all network interfaces in the subnet. So, in our two examples, the broadcast for the `/24` network would be:

Address in Decimal	192	168	122	182
Address in Binary	11000000	10101000	01111010	10110110
Mask (in binary)	11111111	11111111	11111111	00000000
Broadcast	11000000	10101000	01111010	11111111

Table 2.4 – Broadcast Address for a /24 NetworkOr in other words, this network has a broadcast address of 192.168.122.255. For the /20 network:

Decimal	192	168	122	182
Binary	11000000	10101000	01111010	10110110
Mask (in binary)	11111111	11111111	11110000	00000000
Broadcast	11000000	10101000	01111111	11111111

Table 2.5 – Broadcast Address for the Same Network, but Different Subnet MaskOr, converting back to decimal for a broadcast address of 192.168.127.255 Moving the border between the network and host portions of the IPv4 address brings the concepts of **address class** to mind. When converted to binary, the first few bytes define what is called the *classful* subnet mask for that address. In most operating systems, if you set the IP Address in a GUI, this classful subnet mask is what is often filled in by default. These binary – to – subnet mask assignments work out to be:

Class	Leading Bits in Address	Subnet Mask (bits)	Subnet Mask (decimal)	First Address in Range	Last Address in Range
Class A	0	/8	255.0.0.0	0.0.0.0	127.255.255.255
Class B	10	/16	255.255.0.0	128.0.0.0	191.255.255.255
Class C	110	/24	255.255.255.0	192.0.0.0	223.255.255.255
Class D (Multicast Addresses)	1110	/4	240.0.0.0	224.0.0.0	239.255.255.255
Class E (Reserved, not in use)	1111	NA	NA	240.0.0.0	255.255.255.255

Table 2.6 What this defines is the default classful subnet masks for networks. We'll dig deeper into this in the next two sections. From all of this, you can see why most administrators use **classful boundaries** on subnets inside their organization. By far most internal subnets have masks of 255.255.255.0 or 255.255.0.0. Any other choice turns into confusion each time you add a new member to the team, with the potential for errors in server or workstation configurations. Plus, "doing math" every time you need to set or interpret a network address doesn't appeal to most people. The second type of "special" address, which we just touched on above, is **multicast** addresses. A multicast address is used to include several devices in a conversation. For instance, you might use a multicast address to send an identical video stream to a number of network-attached displays, or if you were setting up a conference call or meeting in a voice/video application. Multicast addresses local to a network take the form:

Possible Binary Values	11100000	00000000	00000xxx	xxxxxxx
Possible Decimal Values	224	0	Any number between 1-8	Any number between 1-255

Table 2.7: Multicast Address Represented in Binary... The last 11 bits (3+8) usually form "well known addresses" for various multicast protocols. Some commonly seen multicast addresses are:

224.0.0.1	All hosts on the subnet
224.0.0.2	All routers in the subnet
224.0.0.12	DHCP Servers and DHCP Relay Agents
224.0.0.18	Devices participating in the VRRP protocol
224.0.0.102	Devices participating in the HSRP (Hot Standby Router Protocol) protocol
224.0.1.1	All NTP (Network Time Protocol) Servers
224.0.0.113	All Joyn hosts (this is used by Windows for discovery of neighbor devices)

Table 2.8: Common Multicast Addresses The full list of well know, registered multicast addresses is maintained by the **IANA (Internet Assigned Numbers Authority)**, at <https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml>. While this may seem comprehensive, vendors will often create their own

"custom" multicast addresses in this address space. This serves as a basic introduction to multicast addressing – it's much more complex than this, to the point where entire books are devoted to the design, implementation and theory behind it. What we've covered is enough to get the general idea though, enough to get started. With broadcast and multicast addresses covered, let's discuss the IP address *families* that are most likely used in your environment.

Private Addresses – RFC 1918

Another set of "special" addresses that you are likely to see is the RFC 1918 address space. RFC 1918 describes a list of IP Subnets that are allocated for internal use within an organization. These addresses are not valid for use on the public internet, so must be translated using **Network Address Translation (NAT)** before traffic to or from them can be routed over the public internet. The RFC 1918 addresses are:

- 10.0.0.0/8 (Class A)
- 172.16.0.0 to 172.31.0.0 / 16 (Class B) (this can be summarized as 172.16.0.0/12)
- 192.168.0.0/16 (Class C)

These addresses give organizations a large IP space to use internally, all of which are guaranteed to not conflict with anything on the public Internet. For an interesting exercise, you can verify the address class for each of these RFC 1918 subnets. Translate the first octet of each to binary, then compare them to the table in the last section. The RFC 1918 specification is fully documented here: <https://tools.ietf.org/html/rfc1918> Now that we've covered off the binary aspects of IP addressing and subnet masks, as well as the more common *special* IP address groups, I'm sure that you're tired of theory and math and want to get back to playing with the command line of your Linux host! Good news, we still need to cover the bits and bytes of addressing for IPv6 (IP version 6). Even better news, it will be in an appendix, so that we can get you to a keyboard that much sooner! Now that we've got a firm grasp on displaying IP parameters and a good understanding of IP addressing, let's configure an IP interface for use.

Assigning an IP Address to an Interface

Assigning a permanent IPv4 address is something that you will likely need to do on almost every server that you build. Luckily, it's pretty simple to do. In the new command set, we'll use the `nmcli` command (**Network Manager Command Line**). We'll set the IP address, default gateway, and DNS Server; finally, we'll set the addressing mode to `manual`:

1. First, we'll display the network connections in the `nmcli` format:

```
robv@ubuntu:~$ sudo nmcli connection show
NAME                UUID                                  TYPE      DEVICE
Wired connection 1  354f24fe-0922-30d3-8bd3-24cdb7d7d528  ethernet  ens33
```

Our connection name is `wired connection 1`. We don't need to type this each time though, we can do tab completion on this, by typing `wi`, then pressing `Tab` to complete the name. Also keep in mind that `nmcli` will allow shortened command clauses, so we can use `mod` for `modify`, `con` for `connection` and so on.

1. Let's go forward with our sequence of commands (note how the parameters are shortened in the last command):

```
$ sudo nmcli connection modify 'Wired connection 1' ipv4.addresses 192.168.122.22/24
$
$ sudo nmcli connection modify 'Wired connection 1' ipv4.gateway 192.168.122.1
$
$ sudo nmcli connection modify 'Wired connection 1' ipv4.dns 8.8.8.8
$
$ sudo nmcli con mod 'Wired connection 1' ipv4.method manual
$
```

1. Now, let's save the changes and make them *live*:

```
$ sudo nmcli connection up 'Wired connection 1'
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnec
$
```

Using the legacy approach, all of our changes are done by editing files. And just for fun, the file names and locations will change from distro to distro. The most common edits and files are shown below:

- To change the DNS servers on a legacy Linux host, edit `/etc/resolv.conf` and change the `nameserver` line to reflect the desired server IP:

```
nameserver 8.8.8.8
```

- To change the IP address, subnet mask and so on, edit the file `/etc/sysconfig/network-scripts/ifcfg-eth0`, and update the values as below:

```
DEVICE=eth0
BOOTPROTO=none
ONBOOT=yes
NETMASK=255.255.255.0
IPADDR=192.168.122.22
```

- If your default gateway is on this interface, you can add:

```
GATEWAY=192.168.122.1
```

Again, note that on different distributions, the files to edit may vary, and note especially that **this approach is not backwards compatible**. On a modern Linux systems, this approach of editing the base files for network changes will often fail to work as expected. Now that we know how to assign an IP address to an interface, let's learn how to adjust routing on our host.

Adding a Route

To add a temporary static route, the `ip` command is our again our go-to:

1. In this example, we tell our host to route to `192.168.122.10` to get to the network `10.10.10.0/24`:

```
robv@ubuntu:~$ sudo ip route add 10.10.10.0/24 via 192.168.122.10
[sudo] password for robv:
robv@ubuntu:~$ ip route
default via 192.168.122.1 dev ens33 proto dhcp metric 100
10.10.10.0/24 via 192.168.122.10 dev ens33
169.254.0.0/16 dev ens33 scope link metric 1000
192.168.122.0/24 dev ens33 proto kernel scope link src 192.168.122.156 metric 100
```

You can also add the `egress` network interface to use for this, by tacking `dev <devicename>` on the end of that `ip route add` command. This just adds a temporary route though, which will not survive if the host is rebooted or if the network processes are restarted. You can add a permanent static route by using the `nmcli` command.

1. First we'll display the network connections in the `nmcli` format:

```
robv@ubuntu:~$ sudo nmcli connection show
NAME                UUID                                  TYPE      DEVICE
Wired connection 1  354f24fe-0922-30d3-8bd3-24cdb7d7d528  ethernet  ens33
```

1. Next, we'll add the route to `10.10.11.0/24` via `192.168.122.11` to the connection `Wired connection 1` using `nmcli`:

```
robv@ubuntu:~$ sudo nmcli connection modify 'Wired connection 1' +ipv4.routes '10.10.11.0/24 192
```

1. Again, let's save our `nmcli` changes:

```
$ sudo nmcli connection up 'Wired connection 1'
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnec
```

\$

1. Now, looking at our routing table, we see both of our static routes:

```
robv@ubuntu:~$ ip route
default via 192.168.122.1 dev ens33 proto dhcp metric 100
10.10.10.0/24 via 192.168.122.10 dev ens33
10.10.11.0/24 via 192.168.122.11 dev ens33 proto static metric 100
169.254.0.0/16 dev ens33 scope link metric 1000
192.168.122.0/24 dev ens33 proto kernel scope link src 192.168.122.156 metric 100
```

1. However, if we reload, we see that our temporary route is now gone, and the permanent one is still in place:

```
robv@ubuntu:~$ ip route
default via 192.168.122.1 dev ens33 proto dhcp metric 100
10.10.11.0/24 via 192.168.122.11 dev ens33 proto static metric 100
169.254.0.0/16 dev ens33 scope link metric 1000
192.168.122.0/24 dev ens33 proto kernel scope link src 192.168.122.156 metric 100
```

With the basics of adding routes completed, let's take a look at getting that same tasks done on an older Linux host, using the legacy *route* commands.

Adding a route using Legacy Approaches

Let's get started with adding a route using Legacy Approaches:

1. First, to add a route:

```
$ sudo route add -net 10.10.12.0 netmask 255.255.255.0 gw 192.168.122.12
```

To make this route permanent, things get complicated – permanent routes are stored in files, and the file names and locations will be different depending on the distro, which is why the consistency of the *iproute2* / *nmcli* commands make things so much easier on modern systems.

1. On an older Debian / Ubuntu family distribution, a common method would be edit the file */etc/network/interfaces*, and add the line:

```
up route add -net 10.10.12.0 netmask 255.255.255.0 gw 192.168.122.12
```

1. Or, on an older Redhat family distro, edit the file */etc/sysconfig/network-scripts/route-`<device name>`*, and add the line:

```
10.10.12.0/24 via 192.168.122.12
```

1. Or, to just add the routes as commands, edit the file */etc/rc.local* – this approach will work on just about any Linux system, but is considered by far to be less elegant - mainly because it's the last place the next administrator will look for the setting (since it's not a proper network settings file). The *rc.local* file simply executes on system startup, and runs whatever commands are in it. In this case we'll add our *route add* command, but it's a handy approach to keep in mind for just about any startup command you might want to add:

```
/sbin/route add -net 10.10.12.0 netmask 255.255.255.0 gw 192.168.122.12
```

At this point, we're well on our way to setting up networking on our Linux host. We've set the IP address, subnet mask, and routes. Especially in troubleshooting or initial setup though, it's common to have to disable or enable an interface, we'll cover that next.

Disabling and Enabling an Interface

In the newer command world, we use the – you guessed it – the *ip* command. Here we'll bounce the interface, bringing it down and then back up again:

```
robov@ubuntu:~$ sudo ip link set ens33 down
robov@ubuntu:~$ sudo ip link set ens33 up
```

In the older command set, use `ifconfig` to disable or enable an interface:

```
robov@ubuntu:~$ sudo ifconfig ens33 down
robov@ubuntu:~$ sudo ifconfig ens33 up
```

When executing interface commands, always keep in mind that you don't want to "cut off the branch that you are sitting on". If you are connected remotely (using `ssh` for instance), if you change `ip` addressing, routes or disable an interface, you can easily lose your connection to the host after pressing a fateful *Enter* key. If you are doing maintenance of this type, you can often script all the changes so that you'll be able to reconnect after, or in other situations, you may need to configure using an alternate access – hypervisors are great for this (see the virtualization chapters later in this book). At this point, we've got many of the more common tasks covered that you'll need in your arsenal to configure your Linux host in a modern network. A big part of network administration though is diagnosing and setting configurations to accommodate special cases, for instance – adjusting settings to optimize traffic -0 for instance to adjust things to accommodate links where smaller or larger packet sizes might be needed.

Setting the MTU on an Interface

One operation that is more and more common in modern systems is setting the **Message Transfer Unit (MTU)**. This is the size of the largest **Protocol Datagram Unit (PDU)**, also called a *frame* in most networks) that the interface will send or receive. On Ethernet, the default MTU is 1500 bytes, which works out to a maximum packet size of 1500 bytes. The maximum packet size for a media is generally called the MSS (Maximum Segment Size). For Ethernet, the 3 values work out to:

Maximum Frame Size	1518 Bytes
MTU – Maximum Payload of a Frame	1500 Bytes
Maximum Packet Size (Same as MTU, since the packet is the Frame's Payload)	1500 Bytes
MSS (Maximum Segment Size) – Maximum payload of a single packet	1460 Bytes

Table 2.9: Relating Frame Size, MTU, Packet Size, and MSS for EthernetWhy would we need to change this? 1500 is a nice compromise for packet size, in that it's small enough that in the event of an error, that error is quickly detected and the amount of retransmitted data is relatively small. However, in datacenters especially there are a few exceptions. When dealing with storage traffic, in particular iSCSI, larger frame sizes are desired so that the packet size can accommodate more data. In these cases, the MTU is usually set to somewhere in the range of 9000 (often called a *jumbo packet*). These storage networks can be deployed on 1Gbps, but more often you'll see them on 10Gbps or faster networks. You'll also see larger packets used in traffic to accommodate backups or VM migration (for instance: VMotion in VMware or Live Migration in Hyper-V). At the other end of the spectrum, you'll also often see situations where smaller packets are needed. This is particularly important as not all hosts will detect this well, and many applications will set the "DF" (Don't Fragment) bit in their traffic. In that situation you might see a 1500 byte packet set with DF on a media that might only support a 1380 byte – in that case the application will simply fail, and often the error messages at the host won't be helpful in troubleshooting. Where might you see this? Any link that involves packets being encapsulated will usually involve this - tunnels or VPN solutions for instance. You may also see links that use double encapsulation – GRE tunnels for instance encapsulated within IPSEC. Dynamic Mode VPN (**DMVPN**) tunnels are also a good example of this. Links like this often are run over the public internet, which is usually delivered as a generic 1500 byte MTU ethernet. This means that the encapsulation generally starts with a 1500 byte packet, and each encapsulation in turn whittles that number down. This is usually pretty easy to compute – you'll have to tools to do this sort of calculation after you get to the chapter on Packet Capture. Older style satellite links are another common situation, they'll often default to 512 byte frames – packets in these situations are expensive and high latency, so you want to reduce the impact of any retransmissions that might be needed due to lost or "mangled" packets. In those situations the MTU and MSS sizes will be published by the service provider.How can you estimate your MTU without the math? The `ping` command is a great tool for this. Ping a device at the far end of your path, and set the command line switches to specify the packet size (-s). On Linux by default the DF bit (Don't Fragment) is set, so no switch is

required for this. The first gotcha with this approach is that in Linux, if you specified a size larger than the MTU, your ping command will simply stall, you'll need to Ctrl-C to adjust the size and start over with a smaller value. This can also be a process of some minutes as you guess high and low, and eventually get at what the number is. The final gotcha is that you may not control the routers in the path, and some router admin might have set a policy to clear the DF bit. This means that your ping of size 1500 will work and you'll think all is well, but under the covers every one of your 1500 byte packets will get fragmented into two smaller packets, and you'll have that incremental performance impact on your traffic for as long as you use that path. The safest approach is to calculate what the MTU and MSS should be, and set things based on that. Setting the MTU is as simple as you might think – we'll use `nmcli` again for this. Note in this example that we're shortening the command line arguments for `nmcli`, and we're saving the configuration change at the end – the MTU is changed immediately after the last command. Let's set the MTU to 9000 to optimize iSCSI traffic:

```
$ sudo nmcli con mod "Wired connection 1" 802-3-ethernet.mtu 9000
$ sudo nmcli connection up "Wired connection 1"
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnec
$
```

With our MTU set, what else can we do with the `nmcli` command?

More on the nmcli Command

The `nmcli` command can also be called interactively, and changes can be made in a real-time interpreter or shell. To enter this shell for an Ethernet interface, use the command `nmcli connection edit type ethernet`. In the shell, the `print` command lists all of the `nmcli` parameters that can be changed for that interface type. Note that this output is broken up into logical groups – we've edited this (very lengthy) output to show many of the settings you might need to adjust, edit or troubleshoot in various situations:

```
nmcli> print
=====
                        Connection profile details (ethernet)
=====
connection.id:                ethernet
connection.uuid:              e0b59700-8dcb-4801-9557-9dee5ab7164f
connection.stable-id:         --
connection.type:              802-3-ethernet
connection.interface-name:    --
...
connection.lldp:              default
connection.mdns:              -1 (default)
connection.llmnr:             -1 (default)
-----
```

Common Ethernet options:

```
802-3-ethernet.port:          --
802-3-ethernet.speed:         0
802-3-ethernet.duplex:        --
802-3-ethernet.auto-negotiate: no
802-3-ethernet.mac-address:   --
802-3-ethernet.mtu:           auto
...
802-3-ethernet.wake-on-lan:    default
802-3-ethernet.wake-on-lan-password: --
-----
```

Common IPv4 options:

```
ipv4.method:                  auto
ipv4.dns:                     --
ipv4.dns-search:              --
ipv4.dns-options:             --
ipv4.dns-priority:            0
ipv4.addresses:               --
ipv4.gateway:                 --
```

```

ipv4.routes:                --
ipv4.route-metric:          -1
ipv4.route-table:           0 (unspec)
ipv4.routing-rules:         --
ipv4.ignore-auto-routes:    no
ipv4.ignore-auto-dns:       no
ipv4.dhcp-client-id:        --
ipv4.dhcp-iaid:             --
ipv4.dhcp-timeout:          0 (default)
ipv4.dhcp-send-hostname:    yes
ipv4.dhcp-hostname:         --
ipv4.dhcp-fqdn:             --
ipv4.dhcp-hostname-flags:   0x0 (none)
ipv4.never-default:         no
ipv4.may-fail:              yes
ipv4.dad-timeout:           -1 (default)
-----

```

(Ipv6 options would go here, but are removed to keep this listing readable)Proxy settings:

```

-----
proxy.method:               none
proxy.browser-only:         no
proxy.pac-url:              --
proxy.pac-script:           --
-----

```

nmcli>

As noted, the listing above is somewhat abbreviated, we've shown the settings that you are most likely going to have to check or adjust in various setup or troubleshooting situations. Run the command on your own station to see the full listing. As we've illustrated, the `nmcli` command allows us to adjust several interface parameters either interactively or from the command line. The command line interface in particular allows us to adjust network settings in scripts, allowing us to scale up, adjusting settings on dozens, hundreds, or thousands of stations at a time.

Summary

With this chapter behind us, you should have a firm understanding of IP addressing from a binary perspective. With this, you should understand subnet addressing and masking, as well as broadcast and multicast addressing. You also have a good grasp of the various IP Address "classes". With all of this in hand, you should be able to display or set IP Addresses and routes on a Linux host using a variety of different commands. Other interface manipulations should also be easily accomplished, such as setting the MTU on an interface. With these skills in hand, you are well prepared to embark on our next topic: Using Linux and Linux Tools for Network Diagnostics.

Questions and answers

1. What purpose does the default gateway serve?

A. **Answer:** A default gateway is a special route, usually denoted as 0.0.0.0/0 (in other binary, this indicates *all networks*). A host always has a local routing table, with an order of precedence.

- Any network that is directly connected to an interface is processed first. These are called *connected* or *interface* routes.
- Routes that are defined in the routing table. These are routes you may have added with the `ip` command of the `route` command.
- Finally, the default route is referenced. If the traffic being sent does not match a connected route or a route in the routing table, it is sent to the IP defined in the default gateway. Usually this device will be a special router or firewall device, which in turn will usually have both a local table, statically defined routes and a default gateway (amongst a number of other routing mechanisms that are not in the scope of this book).

2. For a network 192.168.25.0/24, what is the subnet mask and broadcast address?

- A. **Answer:** For this network, the subnet mask is 255.255.255.0 (24 binary bits). The broadcast address is 192.158.25.255.
3. For this same network, how is the broadcast address used?
- A. **Answer:** Traffic sent to the broadcast address is sent to the entire subnet, and is processed by all hosts in that subnet. An example of this would be a standard ARP request (which we will cover in more depth in *Chapter 3*).
4. For this same network, what are the possible host addresses?
- A. **Answer:** The host addresses can range from 192.168.25.1 to 192.168.25.254. The "0" address is the network address, so cannot be used for a host. The "255" address is the broadcast address.
5. If you need to statically set the speed and duplex of an ethernet interface what command would you use?
- A. **Answer:** For instance, to set interface connection wired ethernet 1 to 100Mbps/Full Duplex:

```
$ sudo nmcli connection modify 'Wired connection 1' 802-3-ethernet.speed 100
$ sudo nmcli connection modify 'Wired connection 1' 802-3-ethernet.duplex full
```

Further Reading

- RFC 1918 - Address Allocation for Private Internets: <https://tools.ietf.org/html/rfc1918>
- RFC 791 – Internet Protocol: <https://tools.ietf.org/html/rfc791>

3 Using Linux and Linux Tools for Network Diagnostics

Join our book community on Discord!



<https://packt.link/mqfS2>

In this chapter, we'll cover some "how it works" networking basics, as well as how to use our Linux workstation in network troubleshooting. When you're done with this chapter, you should have tools to troubleshoot local and remote network services, as well as to "inventory" your network and its services. In particular, we'll cover the following topics:

- Networking basics – the OSI model.
- Layer 2 – relating IP and MAC addresses using ARP, with some detail more on MAC addresses.
- Layer 4 – how TCP and UDP ports work, including the TCP "three-way handshake" and how this appears in Linux commands.
- Local TCP and UDP port enumeration, and how these relate to running services.
- Remote port enumeration using native tools.
- Remote port enumeration using installed scanners (netcat and Nmap in particular).
- Finally, we'll cover some of the basics of wireless operations and troubleshooting.

Technical requirements

To follow the examples in this section, we'll use our existing Ubuntu host or **Virtual Machine (VM)**. We'll be touching on some wireless topics in this chapter, so if you don't have a wireless card in your host or VM, you'll want a Wi-Fi adapter to work through those examples. If your host is a VM, most hypervisors will allow you to "attach" a USB mounted device (like a network adapter) to the VM, this works nicely for the examples in this chapter. As we work through the various troubleshooting methods, we'll use a variety of tools, starting with some native Linux commands:

arp	Works with the Address Resolution Protocol (ARP) and the local ARP table in memory. This allows you to relate the physical MAC address on network cards to its IP address.
netplan	A YAML-based tool for configuring network settings.
ip and ifconfig	Configure and display various parameters on the local network interfaces of your host.
netstat and ss	Look at the listening TCP and UDP ports on your host, and relate them back to running processes. Also view the state of the TCP conversations to and from your host.
telnet	While using <code>telnet</code> as a terminal application is very much frowned upon because of its clear-text nature, in a pinch it can still be used as a reasonable troubleshooting tool.
nc (Netcat)	Netcat takes the troubleshooting that you can do in telnet and expands on that dramatically. It can be used to connect to and "poke around" in remote services, and can also be used to host services on the local host for testing.

We'll also use some installed applications:

- Nmap** Enumerate and test listening ports on remote hosts, as well as run various scripts against those ports.
- Kismet** View the details of local wireless networks without connecting to them. Kismet is a text-based tool, so can be run either locally or from an SSH session.
- WaveMon** View the details of a wireless network you are connected to, in particular the signal strength and other performance-related metrics.
- LinSSID** LinSSID is a step up from Kismet, with great graphical views of the signal strength and channel utilization of wireless networks in the local vicinity.

For the packages that aren't included in Ubuntu, be sure that you have a working internet connection so you can use the `apt` commands for installation.

Network basics – the OSI model

It's convenient to discuss network and application concepts in terms of layers, with each layer being roughly responsible for higher and more abstract functions at upper levels, and more *nuts and bolts* primitives as you travel *down the stack*. The following diagram describes the OSI model in broad terms:

Layer	Description / Constructs	Examples
Application	The application the end user interacts with	SMTP, HTTP, HTTPS, FTP, SSH, DNS
Presentation	Formatting Data for the application, encryption and decryption	ASCII, Unicode, SSL, TLS, HTTPS, IPSEC, DTLS
Session	Establish and maintain and end connections between hosts	APIs, Netbios, Tunneling (GRE, MPLS, PPTP)
Transport	End-to-End Connections, transport protocol and error handling	TCP, UDP
Network	Path Determination and Routing / IP Addresses, Packets	Routers, Layer 3 Switches, ICMP, Routing Protocols
Data Link	Communications on the local network / MAC Addresses, Frames	Switches, Wireless Access Points
Physical	Data encoding on the physical media / Bits on the media (wire or wireless)	Cables, Network Cards, W-Fi, Media Converters

Figure 3.1 – The OSI model for network communication, with some descriptions and examples

In regular usage, the layers are often referenced by number, counting from the bottom. So, a Layer 2 problem will usually involve MAC addresses and switches, and will be confined to the VLAN that the station is in (which usually means the local subnet). Layer 3 issues will involve IP addressing, routing, or packets (and so will involve routers and adjacent subnets or more distant networks). As with any model, there's always room for confusion. For instance, there's some longstanding *fuzziness* between layers 6 and 7. Between layers 5 and 6, while IPSEC is definitely encryption and so belongs in layer 6, it can also be considered a tunneling protocol (depending on your point of view and implementation). Even at layer 4, TCP has the concept of a session, so would seem to perhaps have one foot in the layer 5 side – though the concept of *ports* keeps it firmly in at layer 4. And of course, there's always room for humor – the common wisdom/joke is that *people* form Layer 8 in this model. So, a Layer 8 problem might involve a helpdesk call, budget discussions, or a meeting with your organization's management to resolve it! What we see in this next diagram illustrates the most important concept to keep in mind with this model. As data is received, it travels up the stack, from the most primitive constructs it encapsulates to more and more

abstract/high-level constructs (from bits to frames to packets to APIs to applications, for instance). Sending data moves it from the application layer toward the binary representation on the wire (from the upper layers to the lower ones). Layers 1-3 are often referred to as the **media** or **network** layers, whereas layers 4-7 are generally called the **Host** or **Application** layers:

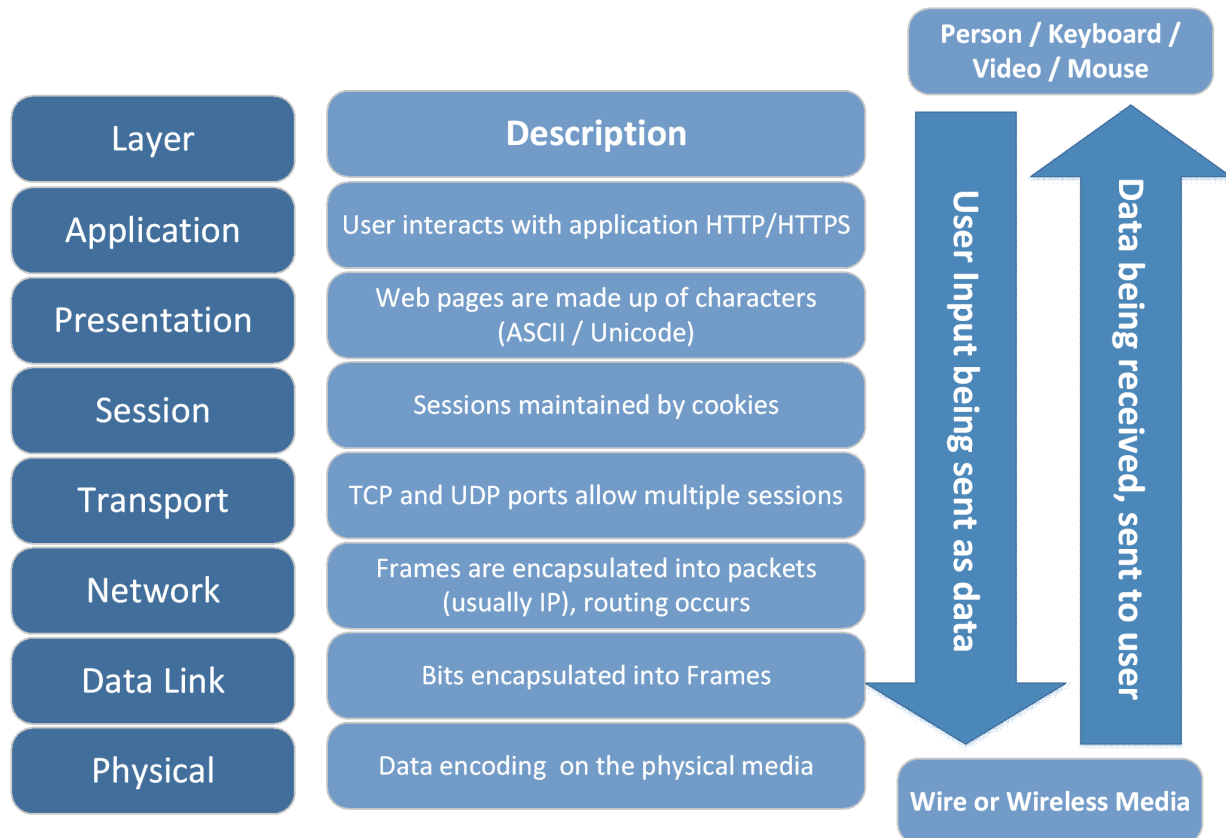


Figure 3.2 – Traveling up and down the OSI stack, encapsulating and decapsulating as we go

This concept is what makes it possible for a vendor to manufacture a switch that will interact with a network card from another vendor for instance, or for switches to work with routers. This is also what powers our application ecosystem – for the most part application developers do not have to worry about IP addresses, routing, or the differences between wireless and wired networks, all that is just taken care of – the network can be treated as a black box where you send data in one end, and you can be certain that it will come out at the other end in the right place and format. Now that we have established the basics of the OSI model, let's learn more about the data link layer (AKA "Layer 2") by exploring the ARP command and the local ARP table.

Layer 2 – relating IP and MAC addresses using ARP

With the OSI model firmly in place, we can see that our discussion so far around IP addresses has been clustered around Layer 3. This is where regular people, and even many IT and networking people, tend to consider the network path to *stop* in their understanding – they can follow the path down that far and consider the rest to be truly a black box. But as a networking professional, Layers 1 and 2 are extremely important – let's start with Layer 2. Theoretically, MAC addresses are the addresses that are *permanently burned into* each network interface. While this is normally true, we'll see later that it's also an easy thing to change. What is the MAC address though? It's a 12 digit (6 byte/48 bit) address, normally shown in hexadecimal. When displayed, each byte or double-byte is usually separated by . or -. So typical MAC addresses might be 00-0c-29-3b-73-cb or 9a93.5d84.5a69 (showing both common representations). In practice, these addresses are used to communicate between hosts in the same VLAN or subnet. If you look at a packet capture (we'll get on to this later in the book, in *Chapter 11, Packet*

Capture), at the start of a TCP conversation you'll see the sending station send a broadcast (a request sent to all stations in the subnet) **ARP request** saying who has IP address x.x.x.x. The **ARP reply** from the host that has that address will include That's me, and my MAC address is aaaa.bbbb.cccc. If the target IP address is on a different subnet, the sender will instead "ARP for" the gateway for that subnet (which will usually be the default gateway, unless there are local routes defined). Going forward, the sender and receiver then communicate using MAC addresses. The switch infrastructure that the two hosts are connected to uses MAC addresses only within each VLAN, which is one reason why switches are so much faster than routers. When we look at the actual packets (in the chapter on *Packet Capture*), you'll see both the sending and receiving MAC address as well as the IP addresses in each packet. The ARP request is cached on each host in an **ARP cache** or **ARP table**, which can be displayed using the `arp` command:

```
$ arp -a
? (192.168.122.138) at f0:ef:86:0f:5d:70 [ether] on ens33
? (192.168.122.174) at 00:c3:f4:88:8b:43 [ether] on ens33
? (192.168.122.5) at 00:5f:86:d7:e6:36 [ether] on ens33
? (192.168.122.132) at 64:f6:9d:e5:ef:60 [ether] on ens33
? (192.168.122.7) at c4:44:a0:2f:d4:c3 [ether] on ens33
_gateway (192.168.122.1) at 00:0c:29:3b:73:cb [ether] on ens33
```

You can see this is pretty simple. It just relates the Layer 3 IP address to the Layer 2 MAC address, then to the Layer 1 **Network Interface Card (NIC)**. MAC address table entries are typically "learned" from traffic – both from ARP requests and replies. They do expire – typically if no traffic is seen to or from a MAC address it'll be cleared from the table after a short interval. You can view your timeout value by listing the right file in the `/proc` directory:

```
$ cat /proc/sys/net/ipv4/neigh/default/gc_stale_time
60
$ cat /proc/sys/net/ipv4/neigh/ens33/gc_stale_time
60
```

Note that there is both a default value (in seconds), and a value for each network adapter (these usually match). This may seem short to you – the matching MAC address table on switches (commonly called the CAM table) is normally at 5 minutes, and the ARP table on routers is normally 14,400 seconds (4 hours). These values are all about resources. In aggregate, workstations have the resources to frequently send ARP packets. The switches *learn* MAC addresses from traffic (including ARP requests and replies), so having that timer be slightly longer than the workstation timer makes sense. Similarly, having a lengthy ARP cache timer on routers conserves its CPU and NIC resources. That timer is so long on routers because in years past, routers were constrained by bandwidth and CPU, compared to just about everything else on the network. While that has changed in modern times, the lengthy default value for ARP cache timeout on routers remains. This is an easy thing to forget during router or firewall migrations – I've been involved in many maintenance windows of that type where a `clear arp` command on the right router magically "fixed everything" after the migration. We haven't talked about the `/proc` directory in Linux yet – this is a "virtual" directory of files, which contain the current settings and statuses of various things on your Linux host. These are not "real" files, but they are represented as files so we can use the same commands that we use for files: `cat`, `grep`, `cut`, `sort`, `awk`, and so on. You can look at network interface errors and values, such as in `/proc/net/dev`, for instance (note how things don't quite line up correctly in this listing):

```
$ cat /proc/net/dev
Inter-|   Receive                                     |   Transmit
face |bytes  packets errs drop fifo frame compressed multicast|bytes  packets errs drop fifo
lo:   208116    2234    0    0    0      0      0      0      0    208116    2234    0    0    0
ens33: 255945718  383290    0  662    0      0      0      0    12013178  118882    0    0    0
```

You can even look in the memory stats (note that `meminfo` contains **lots** more info):

```
$ cat /proc/meminfo | grep Mem
MemTotal:      8026592 kB
MemFree:       3973124 kB
MemAvailable:  6171664 kB
```

Back to ARP and MAC addresses. You can add a static MAC address – one that won't expire and might be different from the real MAC of the host you want to connect to. This is often done for troubleshooting purposes.

Or you can clear an ARP entry, which you might often want to do if a router has been swapped out (for instance if your default gateway router has the same IP but now has a different MAC). Note that you don't need special rights to view the ARP table, but you sure do need rights to modify it! To add a static entry, do the following (note the PERM status when we display it):

```
$ sudo arp -s 192.168.122.200 11:11:22:22:33:33
$ arp -a | grep 192.168.122.200
? (192.168.122.200) at 11:11:22:22:33:33 [ether] PERM on ens33
```

To delete an ARP entry, do the following (note that the `-i interfacename` parameter is routinely skipped for this command):

```
$ sudo arp -i ens33 -d 192.168.122.200
```

To masquerade as a given IP address – for instance, to answer ARP requests for IP `10.0.0.1` – do the following:

```
$ sudo arp -i eth0 -Ds 10.0.0.2 eth1 pub
```

Finally, you can also easily change an interface's MAC address. You might think that this would be done to deal with duplicated addresses, but that situation is exceedingly rare. Legitimate reasons to change a MAC address might include the following:

- You have migrated a firewall and the ISP has your MAC hardcoded.
- You have migrated a host, host NIC, firewall or router and the upstream router isn't accessible to you, and you can't wait 4 hours for the ARP cache to expire on that router.
- You have migrated a host, and there is a DHCP reservation for the old MAC address that you need to use, but you don't have access to "fix" that DHCP entry.

Malicious reasons to change a MAC address include the following:

- You are attacking a wireless network, and have figured out that once authenticated, the only checks that the access point does are against the client MAC addresses. The attack generally involves "de-authenticating" the real host, then using that host's MAC address to gain network access.
- The same as the previous point, but against an Ethernet network that's secured with 802.1x authentication, but with an insecure or incomplete configuration (we'll get into this in more detail in a later chapter).
- You are attacking a wireless network that has MAC address permissions on it.

Hopefully this illustrates that using MAC addresses for security purposes isn't usually a wise or effective decision. To find your MAC addresses, we'll show four different methods. First, using `ip link show`:

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
   link/ether 00:0c:29:33:2d:05 brd ff:ff:ff:ff:ff:ff
```

Next, using the same, but narrowing things down to just the one interface, and the `grep` command to narrow down the results:

```
$ ip link show ens33 | grep link
link/ether 00:0c:29:33:2d:05 brd ff:ff:ff:ff:ff:ff
```

Using `ifconfig` will be the preferred method on older style Linux distros:

```
$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1400
    inet 192.168.122.22 netmask 255.255.255.0 broadcast 192.168.122.255
    inet6 fe80::1ed6:5b7f:5106:1509 prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:33:2d:05 txqueuelen 1000 (Ethernet)
    RX packets 384968 bytes 256118213 (256.1 MB)
    RX errors 0 dropped 671 overruns 0 frame 0
    TX packets 118956 bytes 12022334 (12.0 MB)
```

```

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 2241 bytes 208705 (208.7 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 2241 bytes 208705 (208.7 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Finally, you can again use `grep` make that `ifconfig` output much less verbose, giving you just the answer you are looking for:

```

$ ifconfig ens33 | grep ether
    ether 00:0c:29:33:2d:05 txqueuelen 1000 (Ethernet)

```

To change a Linux host's MAC address, we also have several options:

- In the Linux GUI, you could start by clicking the network icon on the top panel, then select **Settings** for your interface. For instance, for a host with one Ethernet card, choose **Wired**, then **Wired Settings**:

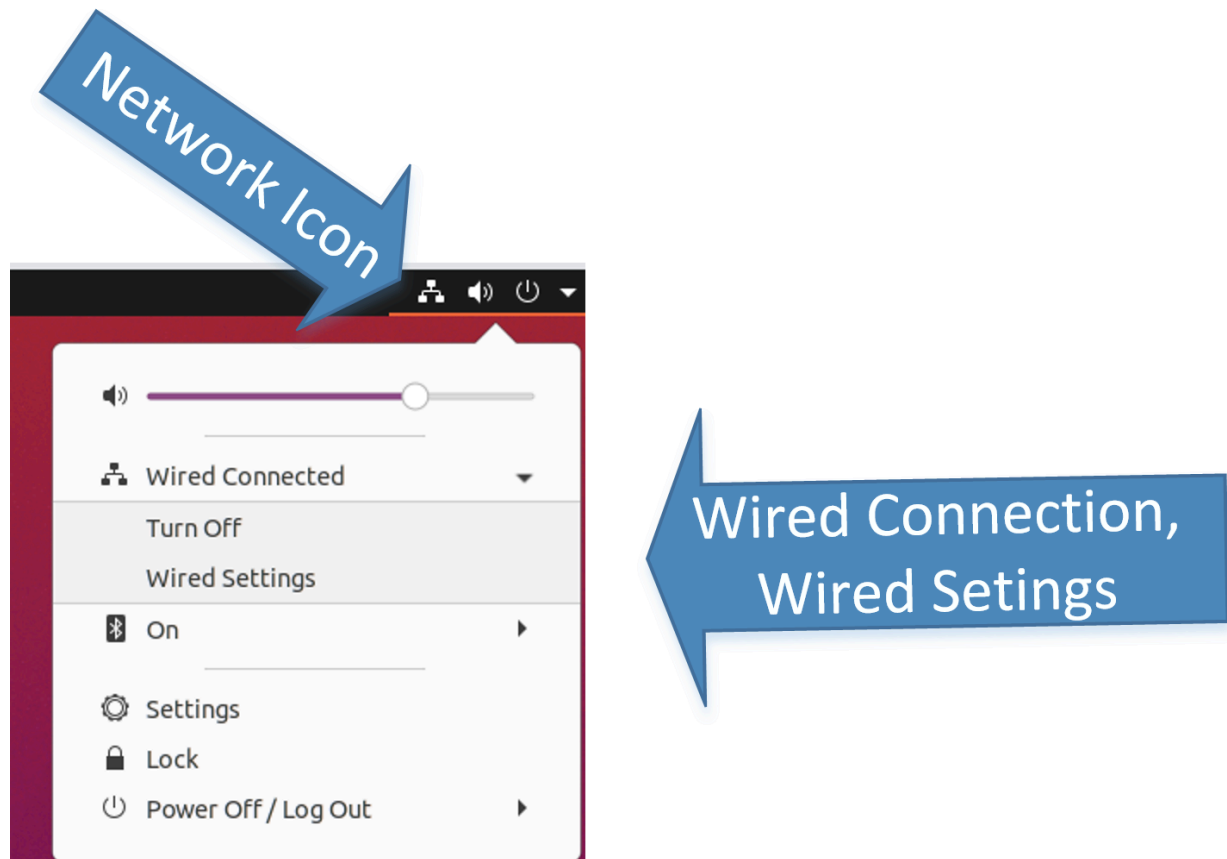


Figure 3.3 – Changing the MAC address from the GUI, step 1

From the interface that pops up, open the **New Profile** dialog box by clicking the + icon, then simply add the MAC in the **Cloned Address** field:

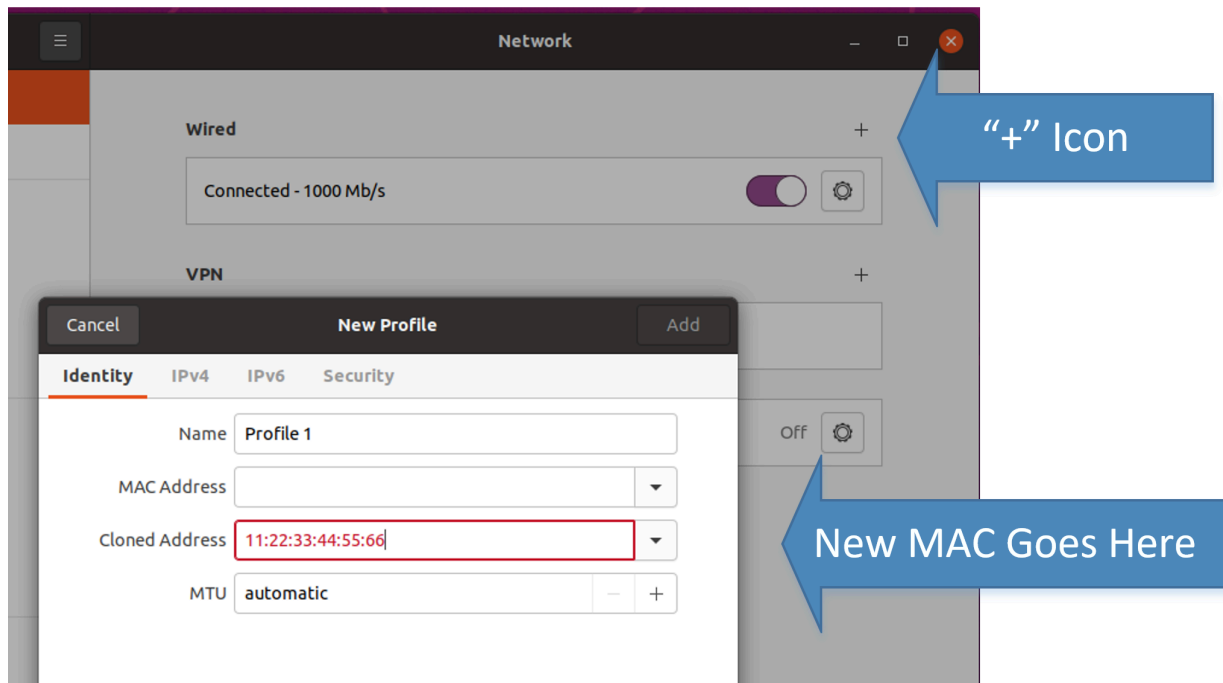


Figure 3.4 – Changing the MAC address from the GUI, step 2

- Or, from the command line or using a script, you can do the following (use your own interface names and target MAC address, of course):

```
$ sudo ip link set dev ens33 down
$ sudo ip link set dev ens33 address 99:88:77:66:55:44
$ sudo ip link set dev ens33 device here> up
```

- There is also the `macchanger` package with which can change your interface's MAC address to a target value or to a pseudo-random value.
- To make a permanent MAC address change, you can also use `netplan` and its associated configuration files. First, make a backup of the configuration file `/etc/netplan/01-network-manager-all.yaml`, then edit it. Note that to change the MAC, you need a `match` statement for the hardware **Burned-In Address (BIA)** MAC address value, then the line after sets the new MAC. Note that the MAC address in these configuration files use colons (:) as a delimiter between the byte boundaries :

```
network:
  version: 2
  ethernets:
    ens33:
      dhcp4: true
      match:
        macaddress: b6:22:eb:7b:92:44
      macaddress: xx:xx:xx:xx:xx:xx
```

You can test your new configuration with `sudo netplan try`, and apply it with `sudo netplan apply`.

- Alternatively, you can create or edit the file `/etc/udev/rules.d/75-mac-spoof.rules`, which will execute on every startup. Add the following:

```
ACTION=="add", SUBSYSTEM=="net", ATTR{address}=="XX:XX:XX:XX:XX:XX", RUN+="/usr/bin/ip link set c
```

With the basics of MAC address usage in ARP mastered, let's dig a bit deeper into MAC address' and their relationship to the manufacturers of the various network adapters.

MAC address OUI values

So now that we've covered timeouts and ARP, do we know everything we need to about Layer 2 and MAC addresses? Not quite yet – let's talk about **Organizationally Unique Identifier (OUI)** values. If you remember our discussion about how IP addresses are split into network and host sections using the subnet mask, you'll be surprised to know that there's a similar dividing line in MAC addresses! The leading bits of each MAC address are supposed to identify the manufacturer – this value is called the OUI. OUIs are registered, with the formal registry maintained by the IEEE and posted at <http://standards-oui.ieee.org/oui.txt>. However, the Wireshark project maintains a more complete listing, located at <https://www.wireshark.org/download/automated/data/manuf>. Wireshark also offer a lookup web application for this listing at <https://www.wireshark.org/tools/oui-lookup.html>. Normally a MAC address is split equally, with the first 3 bytes (6 characters) being allocated to the OUI, and the last 3 bytes being allocated to uniquely identify the device. It's up to the manufacturer to ensure that all devices have unique MAC addresses (which effectively means those last 3 bytes, or 16,777,213 usable MAC addresses). However, organizations are able to purchase longer OUIs (for a lower fee), which give them fewer device addresses to allocate. OUIs are valuable tools in network troubleshooting – when problems arise or unknown stations appear on the network, the OUI values can help in identifying these culprits. We'll see OUIs crop up later in this chapter, when we discuss network scanners (Nmap in particular).

Note

If you need a command-line OUI parser for Linux or Windows, I have one posted at <https://github.com/robvandenbrink/ouilookup>.

This concludes our first adventures in Layer 2 of the OSI model and our examination of its relationship to Layer 3, so let's venture higher into the stack into layer 4, by looking at the TCP and UDP protocols and their associated services.

Layer 4 – how TCP and UDP ports work

Transmissions Control Protocol (TCP) and **User Datagram Protocol (UDP)** are normally what is meant when we discuss Layer 4 communications, in particular how they use the concept of *ports*. When a station wants to *talk* to another station in the same subnet using its IP address (the IP usually gets determined in the application or presentation layers), it will check its ARP cache to see if there's a MAC address that matches that IP. If there's no entry for that IP address, it will send an ARP request to the local broadcast address (as we discussed in the last section). The next step is for the protocol (TCP or UDP) to establish port-to-port communications. The station picks an available local port, above 1024 and below 65535 (the maximum port value), called an **ephemeral port** (since it is not assigned to any running service). It then uses that port to connect to the fixed server port on the server. The combination of these ports, combined with the IP addresses at each end and the protocol in use (either TCP or UDP) will always be unique (because of the way the source port is chosen), and is called a **tuple**. This tuple concept is expandable, notably in NetFlow configurations, where other values can be "bolted on", such as **Quality of Service (QOS)**, **Differentiated Services Code Point (DSCP)** or **Type of Service (TOS)** values, application names, interface names, and routing information such as **Autonomous System Numbers (ASNs)**, MPLS, or VLAN information and bytes of traffic sent and received. Because of this flexibility, the basic tuple of 5 values (source ip, destination ip, source port, destination port and protocol) that all others are built on is often referred to as the **5-tuple**. The first 1024 ports (numbered 0-1023) are almost never used as source ports – these are designated specifically as server ports, and need root privileges to work with. Ports in the range of 1024 - 49151 are designated "user ports" and 49152 - 65535 are dynamic or private ports. Servers are not however forced to use ports numbered below 1024 though (almost every database server for instance uses port numbers above 1024), and these groupings of ports are just a historical convention that dates back to when TCP and UDP were being developed and all server ports were below 1024. If you look at many of the servers that date back that far, you'll see that convention:

```
DNS    udp/53 , tcp/53
Telnet  tcp/23
```

SSH	tcp/22
FTP	tcp/20 and tcp/21
HTTP	tcp/80
HTTPS	tcp/443
SNMP	udp/162
Syslog	tcp/443

Table 3.1: Commonly seen protocols and the ports they normally use. A full listing of the ports that are formally assigned is maintained by the IANA, and is posted at <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>. This list was originally posted as an RFC (currently RFC1340, Assigned Numbers), but these ports aren't really "assigned", really you can run any protocol on any port. For this reason, IANA has taken over the task of maintaining a list of "well known ports", with the understanding that it is incomplete and depending on the implementation (or the inclination of an attacker) the list may not match the reality of the packets on the network. In practice though, *assignment* is a strong word for this list. While it would be foolish to put a web server on TCP port 53, or a DNS server on UDP port 80, many applications are not on this list at all, so the vendor simply chooses a port that is normally free and uses that. It's not unusual to see vendors select a port that is actually assigned to someone else on this list, but assigned to a more obscure or less used service. So for the most part, this list is a set of strong suggestions, with the unspoken implication that we'll consider any vendor who selects a well-known port for their own use to be... let's say, "foolish." It's also fairly common for administrators to change service ports – especially in web servers where you will see servers on port 80 (the standard), but also on 81 or 82 if multiple servers need to run on the same host (and host headers aren't sufficient to differentiate them). Similarly you'll see https servers on the standard 443, but also on 444, 4443 or 4433. 445 is typically not used for web services as that is commonly used for **SMB (Server Message Block)**, which is commonly used for instance for file and printer sharing in the Microsoft world (on Linux if you have SAMBA installed). The common theme is that services have default ports, but the system administrator can certainly easily change them as required.

Layer 4 – TCP and the three-way handshake

UDP simply picks up from working out the 5-tuple and starts sending data. It's up to the receiving application to take care of receiving that data, or to check the application's packets to verify that things arrive in order and to do any error checking. In fact, it's because of this lack of overhead that UDP is used so often for time-critical applications such as **VoIP (Voice over IP)** and video streaming. If a packet is missed in those types of applications, normally backtracking to retry it will interrupt the stream of data and be noticed by the end user, so in those applications specifically, errors in small volumes are logged for error statistics but are otherwise often simply ignored. Voice or video quality is generally deemed acceptable up to a certain error rate, which will vary based on the application or **codec (Coder-Decoder)**. TCP however negotiates a sequence number, and maintains a sequence count as the conversation progresses. This allows TCP-based applications to keep track of dropped or corrupted packets, and retry those in parallel while more data from the application is still being sent and received. The initial negotiation of this is usually called the **three-way-handshake** – graphically it looks something like this:

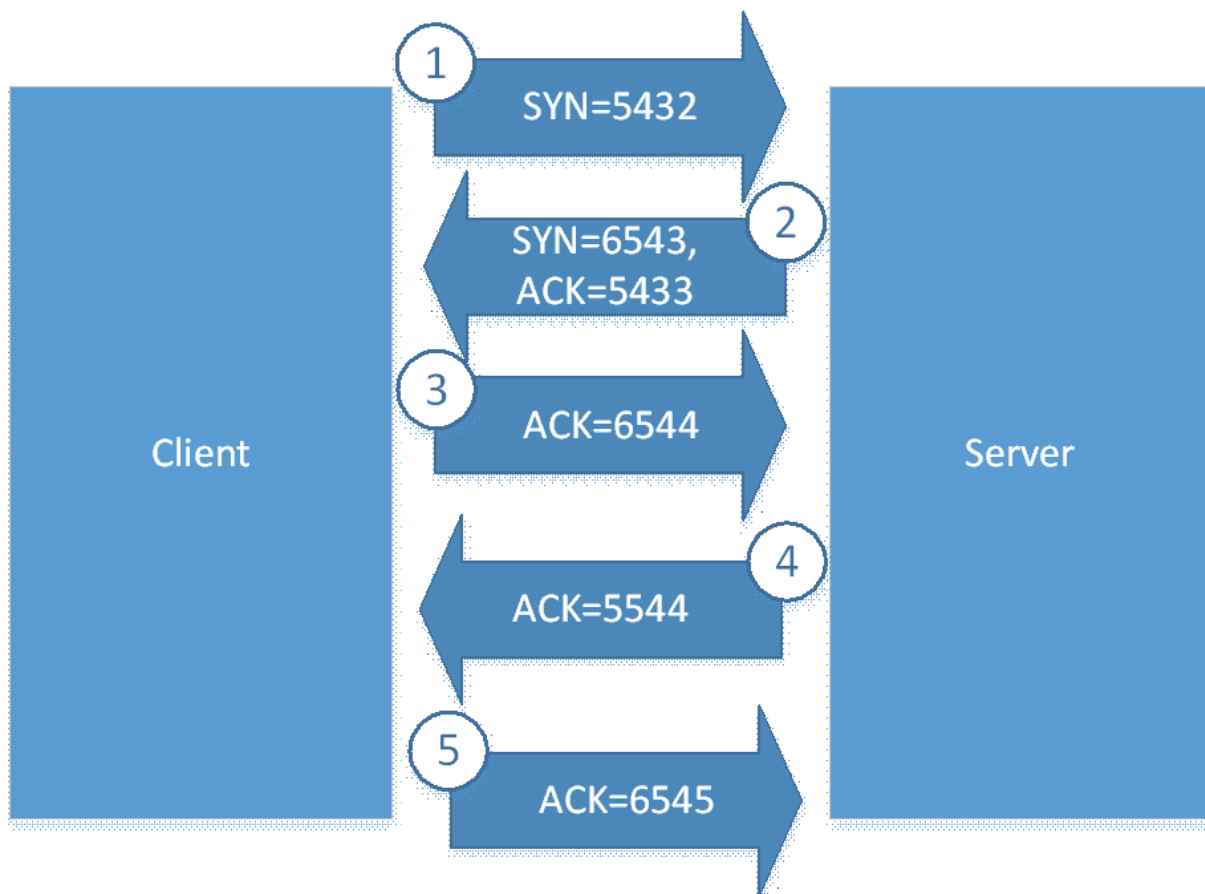


Figure 3.5 – The TCP three-way handshake, with a TCP session established

This works as follows:

1. The first packet comes from the client from an ephemeral port, to the server's (usually) fixed port. It has the **SYN** (synchronize) bit set, and has a randomly assigned **SEQ** (initial sequence) number, in this case **5432**.
2. The reply packet from the server has the **ACK** (acknowledge) bit set, with a number of **5433**, and also has the **SYN** bit set with its own random **SYN** value, in this case **6543**. This packet may already contain data in addition to the handshake information (all subsequent packets may contain data).
3. The third packet is the **ACK** to the server's first **SYN**, with the number **6544**.
4. Going forward, all packets are **ACK** packets sent to the other party so that every packet has a unique sequence number and a direction.

Technically packet number 2 could be two separate packets, but normally they're combined into a single packet, and 5 is the first data packet after the negotiation, so that leaves us with 3 packets in the negotiation itself. The graceful end of a conversation works exactly the same way. The party that's ending the conversation sends a **FIN**, the other replies with a **FIN-ACK**, which gets an **ACK** from the first party, and they're done. An ungraceful end to the conversation is often initiated with a **RST** (reset) packet – once a **RST** is sent, things are over, and the other party shouldn't send a reply to that. We'll use these topics later in this chapter, and also throughout the book. So if you're still fuzzy on this, have another read, especially of the three-way handshake diagram, until this looks right to you. Now that we have some idea of how TCP and UDP ports connect to each other and why your application might use one over the other, let's look at how your host's applications "listen" on various ports.

Local port enumeration – what am I connected to? What am I listening for?

Many fundamental troubleshooting steps in networking happen at one end or the other of a communication link – namely on the client or server host. For instance, if a web server isn't reachable, it's of course useful to see whether the web server process is running and is "listening" on the appropriate ports for client requests, or if the client is "connected" to the server. The `netstat` command is the traditional method to assess the state of network conversations and services on the local host. To list all listening ports and connections, use the following options:

- t TCP ports.
- u UDP ports.
- a All ports (both listening and non-listening).
- n Specifies not to do DNS resolution on the IP addresses involved. Missing this parameter can add significant delay to your output, as the command will attempt to resolve each IP in the list.

Table 3.2: `netstat` command parameters to display various protocols. All four parameters are illustrated as follows:

```
$ netstat -tuan
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 127.0.0.53:53          0.0.0.0:*               LISTEN
tcp      0      0 0.0.0.0:22             0.0.0.0:*               LISTEN
tcp      0      0 127.0.0.1:631          0.0.0.0:*               LISTEN
tcp      0      0 192.168.122.22:34586   13.33.160.88:443        TIME_WAIT
tcp      0      0 192.168.122.22:60862   13.33.160.97:443        TIME_WAIT
tcp      0      0 192.168.122.22:48468   35.162.157.58:443       ESTABLISHED
tcp      0      0 192.168.122.22:60854   13.33.160.97:443        TIME_WAIT
tcp      0      0 192.168.122.22:50826   72.21.91.29:80          ESTABLISHED
tcp      0      0 192.168.122.22:22      192.168.122.201:3310    ESTABLISHED
tcp      0      0 192.168.122.22:60860   13.33.160.97:443        TIME_WAIT
tcp      0      0 192.168.122.22:34594   13.33.160.88:443        TIME_WAIT
tcp      0      0 192.168.122.22:42502   44.227.121.122:443      ESTABLISHED
tcp      0      0 192.168.122.22:34596   13.33.160.88:443        TIME_WAIT
tcp      0      0 192.168.122.22:34588   13.33.160.88:443        TIME_WAIT
tcp      0      0 192.168.122.22:46292   35.244.181.201:443      ESTABLISHED
tcp      0      0 192.168.122.22:47902   192.168.122.1:22        ESTABLISHED
tcp      0      0 192.168.122.22:34592   13.33.160.88:443        TIME_WAIT
tcp      0      0 192.168.122.22:34590   13.33.160.88:443        TIME_WAIT
tcp      0      0 192.168.122.22:60858   13.33.160.97:443        TIME_WAIT
tcp      0      0 192.168.122.22:60852   13.33.160.97:443        TIME_WAIT
tcp      0      0 192.168.122.22:60856   13.33.160.97:443        TIME_WAIT
tcp6     0      0 :::22                  :::*                     LISTEN
tcp6     0      0 :::1:631                :::*                     LISTEN
udp      0      0 127.0.0.53:53          0.0.0.0:*               *
udp      0      0 0.0.0.0:49345          0.0.0.0:*               *
udp      0      0 0.0.0.0:631            0.0.0.0:*               *
udp      0      0 0.0.0.0:5353           0.0.0.0:*               *
udp6     0      0 :::5353                 :::*                     *
udp6     0      0 :::34878                :::*                     *
```

Note the varying states (you can review all of these in the `man` page for `netstat`, using the command `man netstat`). The most common states that you will see are listed in the following table. If the descriptions for either seem confusing, you can skip forward to the next page to work this through using the diagrams (*Figures 3.6 and 3.7*):

LISTEN

Indicates that this is a process running on the host, listening on the port indicated, waiting for someone to connect to it.

ESTABLISHED

Indicates either a local process that is a client to a remote server, or a remote host that is a client to a local

(listening) service.

You can usually tell which is which by the port number. If it's a *well-known* port that's local, the service is local. If that well-known port is on the Foreign address, then that is the server, and you are the client.

If neither port is well known, look at the `LISTEN` ports on this list. If there's a match then you are the server to the remote client.

In packet-speak, this indicates that the TCP three-way handshake has been completed, the TCP session has been established, and both the client and the server are ready to communicate.

TIME_WAIT

This indicates a session that is closed but still waiting for the packets to arrive. In other words, either the client or the server has sent a `FIN` packet. After this happens, the other party sends a `FIN-ACK` packet, followed by a `ACK` packet responding to this. This sounds complicated, but we'll revisit this in a diagram (*Figure 3.7*).

If the final `FIN` handshake doesn't complete, the session will time out.

The less commonly seen states (mostly because these usually only last for a short time) are shown in the following table. If you consistently see any of these states you may have a problem to troubleshoot:

<code>SYN_SENT</code>	The client has sent a <code>TCP SYN</code> packet, and is waiting for a response from the server. This should be followed immediately by the server replying with <code>SYN-ACK</code> , so this state should only be in play for milliseconds.
<code>SYN_RECV</code>	The server has received that <code>SYN</code> packet from the client. It should immediately respond with a <code>SYN-ACK</code> packet, so this <code>SYN_RECV</code> state should only be there for a few milliseconds.
<code>FIN_WAIT1</code> and <code>2</code>	The session is being shut down, and the connection is shutting down. The client or server is waiting for a <code>FIN</code> packet from the other party.
<code>CLOSE</code>	The session has been terminated.
<code>CLOSE_WAIT</code>	The remote end has shut down and is waiting for the application to close its socket. This is normally very fast, essentially this is Layer 4 (session) communicating up the OSI stack to Layer 7 (application), telling it that "the party is over."
<code>LAST_ACK</code>	The session is almost closed. The remote end has sent a <code>FIN</code> packet to start shutting things down, and this station has sent the <code>FIN-ACK</code> confirming it. This station is waiting for the final <code>ACK</code> packet from the other end (in other words, the first two phases of the three-way session shutdown handshake are completed). Again, this should take milliseconds to complete.
<code>CLOSING</code>	Both sockets are shut down but we still don't have all our data sent.

How do these states relate to the handshake we just discussed? Let's put them in a diagram – note again that in most cases the intermediate steps should only exist for a very short time. If you see a **`SYN_SENT`** or **`SYN_RECV`** state for more than a few milliseconds you likely have some troubleshooting to do:

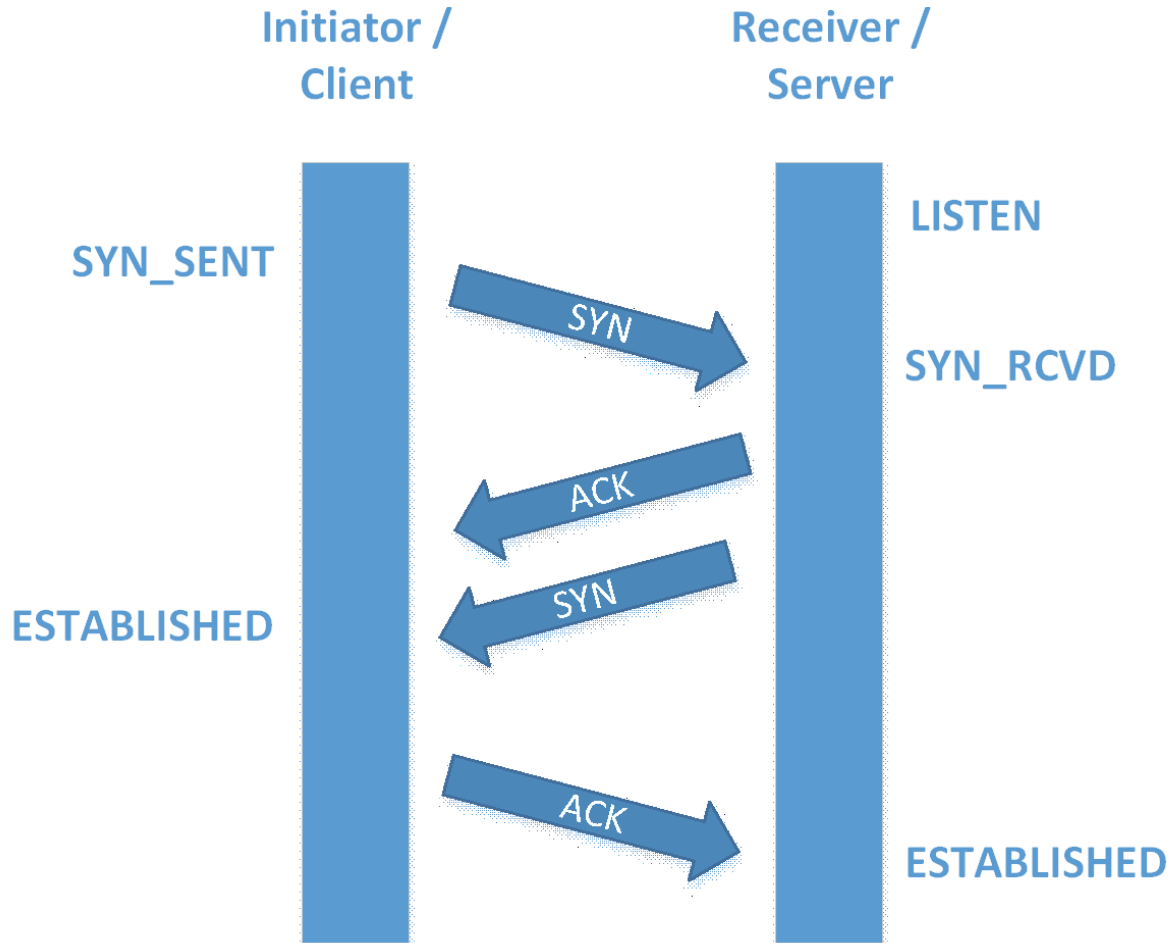


Figure 3.6 – TCP session status at various points as the session is established

You'll see similar states as a TCP session is torn down. Note again that many of the intermediate states should only last a short time. Poorly written applications often don't do session teardown correctly, so you may see states such as `CLOSE_WAIT` in those situations. Another case where session teardown isn't done well is when an in-path firewall has a maximum TCP session length defined. This setting is usually in place to handle poorly written applications that don't close out correctly, or perhaps never close out at all. A maximum session timer however can also interfere with long-running sessions such as older-style backup jobs. If you have such a situation and the long-running session doesn't recover well (for instance a backup job that errors out instead of resuming the session), you may need to work with the firewall administrator to increase this timer, or work with the backup administrator to look at more modern backup software (with multiple parallel TCP sessions and better error recovery for instance):

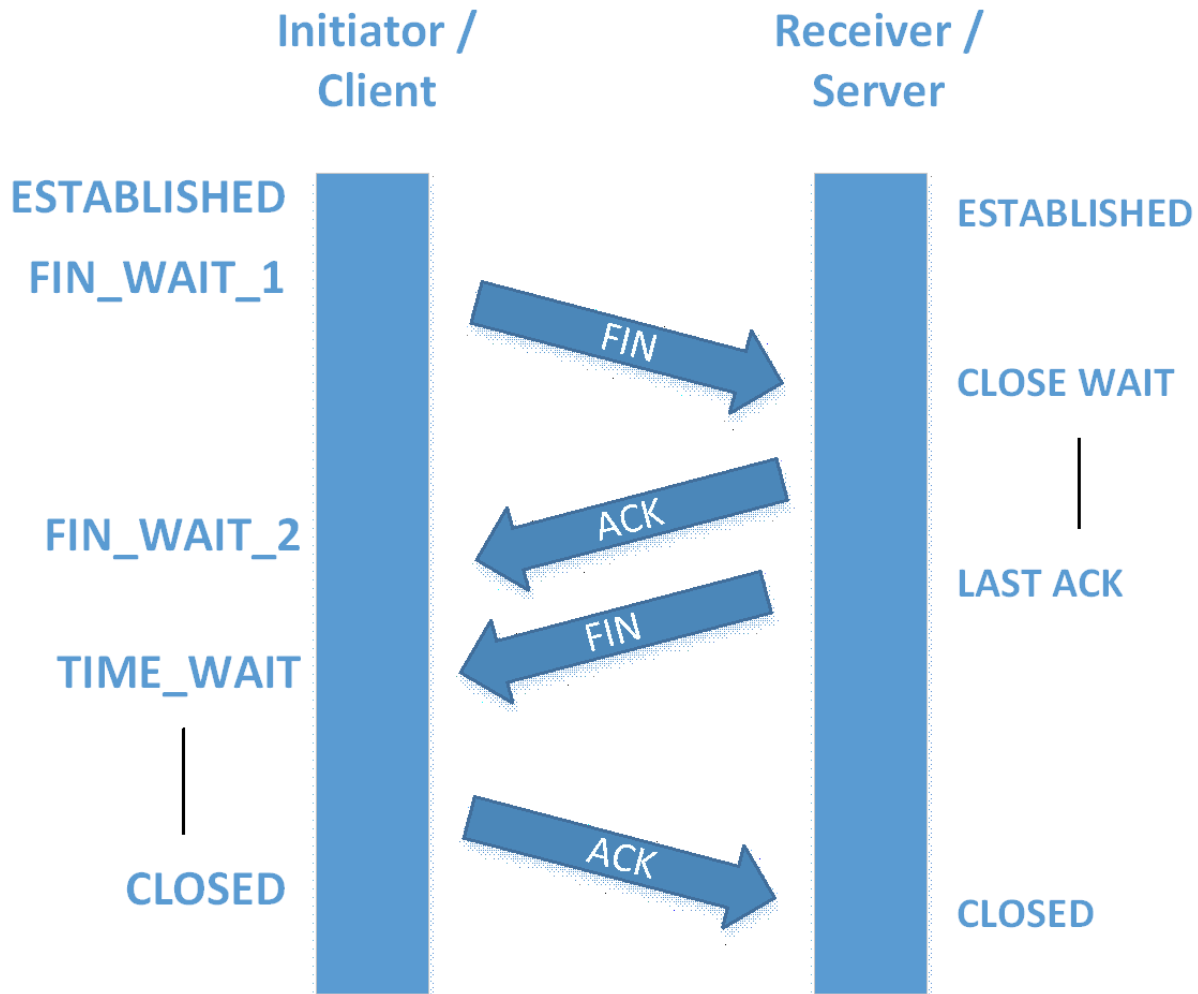


Figure 3.7 – TCP session status at various points as the session is "torn down"

Notice that on the session initiation, we don't have two states that separate the **SYN** and **ACK** back from the server – there are a lot more states involved in shutting down a session than in standing one up. Also note that packets turn around in fractions of a second, so if you see any TCP sessions in a `netstat` display that are anything other than **ESTABLISHED**, **LISTENING**, **TIME-WAIT**, or (less frequently) **CLOSED**, then something is unusual – the other states tend to come and go in a blink! To relate the listening ports back to the services that are behind them, we'll use `l` (for listening) instead of `a`, and add the `p` option for program:

```
$ sudo netstat -tulpn
[sudo] password for robv:
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 127.0.0.53:53          0.0.0.0:*               LISTEN      666/systemd-reso
tcp        0      0 0.0.0.0:22             0.0.0.0:*               LISTEN      811/sshd: /usr/st
tcp        0      0 127.0.0.1:631          0.0.0.0:*               LISTEN      4147/cupsd
tcp6       0      0 :::22                  :::*                    LISTEN      811/sshd: /usr/st
tcp6       0      0 :::1:631               :::*                    LISTEN      4147/cupsd
udp        0      0 127.0.0.53:53          0.0.0.0:*               LISTEN      666/systemd-reso
udp        0      0 0.0.0.0:49345          0.0.0.0:*               LISTEN      715/avahi-daemon
udp        0      0 0.0.0.0:631           0.0.0.0:*               LISTEN      4149/cups-browse
udp        0      0 0.0.0.0:5353           0.0.0.0:*               LISTEN      715/avahi-daemon
udp6       0      0 :::5353                :::*                    LISTEN      715/avahi-daemon
udp6       0      0 :::34878               :::*                    LISTEN      715/avahi-daemon
```

Output of "netstat -tulnp"

Are there alternatives to netstat ? Definitely, there are many. ss for instance has almost the same functions. In the following table, you can see what we've asked for:

- t TCP ports
- u UDP ports
- a Both directions ("all")

Let's add the process information by adding the p option:

```
$ sudo ss -tuap
Netid      State      Recv-Q      Send-Q      Local Address:Port      Peer Address:Port
udp        UNCONN     0            0           127.0.0.53%lo:domain
udp        UNCONN     0            0           0.0.0.0:49345
udp        UNCONN     0            0           0.0.0.0:631
udp        UNCONN     0            0           0.0.0.0:mdns
udp        UNCONN     0            0           [::]:mdns
udp        UNCONN     0            0           [::]:34878
tcp        LISTEN     0            4096        127.0.0.53%lo:domain
tcp        LISTEN     0            128         0.0.0.0:ssh
tcp        LISTEN     0            5          127.0.0.1:ipp
tcp        ESTAB      0            64         192.168.122.22:ssh      192.168.122.201:3310
tcp        ESTAB      0            0          192.168.122.22:42502    44.227.121.122:https
tcp        TIME-WAIT  0            0          192.168.122.22:46292    35.227.121.122:https
tcp        ESTAB      0            0          192.168.122.22:47902    192.168.122.1:ssh
tcp        LISTEN     0            128        [::]:ssh
tcp        LISTEN     0            5          [::1]:ipp
```

Notice how that last column was wrapped onto the next line? Let's use the `cut` command to only select some fields in this text display. Let's ask for columns 1, 2, 4, 5, and 6 (we'll remove the `Recv-Q` and `Send-Q` fields). We'll do this using the concept of *piping* the output of one command to the next command. The `cut` command has only a few options, and normally you'll use the `d` (delimiter) and `f` (field number) options. In our case, our delimiter is a *space* character, and we want fields 1, 2, 5, and 6. Unfortunately, we have multiple spaces between our fields. How can we fix that? Let's use the `tr` (translate) command. Normally `tr` will translate one single character to a single different character, for instance `tr 'a' 'b'` will replace all occurrences of `a` with `b`. In our case though, we'll use the `s` option of `tr`, which will reduce (squeeze) multiple occurrences of the target character down to one. What will our final set of commands look like? Look at the following:

```
sudo ss -tuap | tr -s ' ' | cut -d ' ' -f 1,2,4,5,6 --output-delimiter='${t}'
```

The first command is the same `ss` command we used last time. We send that on to `tr`, which replaces all repeated space characters with a single space. `cut` gets the output of this and executes the following: "using the space character delimiter, give me only fields 1, 2, 5, and 6, using a *tab* character between my resulting columns." Our final result? Let's see:

```
sudo ss -tuap | tr -s ' ' | cut -d ' ' -f 1,2,5,6 --output-delimiter='${t}'
Netid      State      Local Address:Port
udp        UNCONN     127.0.0.53%lo:domain 0.0.0.0:*
udp        UNCONN     0.0.0.0:49345        0.0.0.0:*
udp        UNCONN     0.0.0.0:631          0.0.0.0:*
udp        UNCONN     0.0.0.0:mdns         0.0.0.0:*
udp        UNCONN     [::]:mdns            [::]:*
udp        UNCONN     [::]:34878           [::]:*
tcp        LISTEN     127.0.0.53%lo:domain 0.0.0.0:*
tcp        LISTEN     0.0.0.0:ssh          0.0.0.0:*
tcp        LISTEN     127.0.0.1:ipp        0.0.0.0:*
tcp        ESTAB      192.168.122.22:ssh    192.168.122.201:3310
tcp        ESTAB      192.168.122.22:42502  44.227.121.122:https
tcp        ESTAB      192.168.122.22:47902  192.168.122.1:ssh
tcp        LISTEN     [::]:ssh             [::]:*
tcp        LISTEN     [::1]:ipp            [::]:*
```

Using the tab for a delimiter gives us a better chance of the resulting columns lining up. If this were a larger listing, we might send the whole output to a `.tsv` (short for **tab-separated variables**) file, which can be read directly by most spreadsheet applications. This would be done using a variant of piping, called **redirection**. In this example, we'll send the whole output to a file called `ports.csv` using the `>` operator, then type the file using the `cat` (concatenate) command:

```
$ sudo ss -tuap | tr -s ' ' | cut -d ' ' -f 1,2,5,6 --output-delimiter=$'\t' > ports.tsv
$ cat ports.out
Netid    State    Local    Address:Port
udp      UNCONN  127.0.0.53%lo:domain  0.0.0.0:*
udp      UNCONN  0.0.0.0:49345  0.0.0.0:*
udp      UNCONN  0.0.0.0:631    0.0.0.0:*
udp      UNCONN  0.0.0.0:mdns    0.0.0.0:*
udp      UNCONN  [::]:mdns       [::]:*
udp      UNCONN  [::]:34878      [::]:*
tcp      LISTEN  127.0.0.53%lo:domain  0.0.0.0:*
tcp      LISTEN  0.0.0.0:ssh     0.0.0.0:*
tcp      LISTEN  127.0.0.1:ipp   0.0.0.0:*
tcp      ESTAB   192.168.122.22:ssh  192.168.122.201:3310
tcp      ESTAB   192.168.122.22:42502  44.227.121.122:https
tcp      ESTAB   192.168.122.22:47902  192.168.122.1:ssh
tcp      LISTEN  [::]:ssh        [::]:*
tcp      LISTEN  [::1]:ipp       [::1]:*
```

Finally, there's a special command called `tee` that will send the output to two different locations. In this case, we'll send it to the `ports.out` file, and the special file `STDOUT` (standard output), which essentially means "type it back into my terminal session". For fun, let's use the `grep` command to select only established sessions:

```
$ sudo ss -tuap | tr -s ' ' | cut -d ' ' -f 1,2,5,6 --output-delimiter=$'\t' | grep "EST" | tee ports.out
tcp      ESTAB   192.168.122.22:ssh  192.168.122.201:3310
tcp      ESTAB   192.168.122.22:42502  44.227.121.122:https
tcp      ESTAB   192.168.122.22:47902  192.168.122.1:ssh
```

Want to see some more detailed statistics on the TCP conversations? Use `t` for TCP, and `o` for options:

```
$ sudo ss -to
State    Recv-Q    Send-Q    Local Address:Port    Peer Address:Port    Process
ESTAB    0          64        192.168.122.22:ssh    192.168.122.201:3310  timer:(on,24)
ESTAB    0          0         192.168.122.22:42502  44.227.121.122:https  timer:(keep,24)
ESTAB    0          0         192.168.122.22:47902  192.168.122.1:ssh    timer:(keep,24)
```

This TCP options display can be useful in troubleshooting long-lived TCP sessions that might run through a firewall. Because of memory constraints, firewalls will periodically clear TCP sessions that have not terminated correctly. Since they haven't terminated, in most cases the firewall will look for sessions that have run longer than `x` minutes (where `x` is some number that has a default value and can be configured). A classic way that this can go sideways is if a client is running a backup or transferring a large file through the firewall, perhaps backing up to a cloud service or transferring a large file in or out of the network. If these sessions exceed that timeout value, they'll of course just get closed at the firewall. It's then up to the application if it just starts over (bad idea as you might imagine) or has some way to pick up where it left off. In cases like this, it's important to see how long individual TCP sessions might last in a long transfer. A backup or a file transfer might be composed using several shorter sessions, running in parallel and in sequence to maximize performance. Or they might be a single transfer that runs as long as the process. This set of `ss` options can help you gauge how your process behaves under the hood, without having to resort to a packet capture (though never fear, we will get to packet captures later in this book). Let's take one more crack at this, looking at the listening ports and relating the display back to the listening service on the host:

```
$ sudo netstat -tulpn
[sudo] password for robv:
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name
tcp    0      0 127.0.0.53:53 0.0.0.0:* LISTEN 666/systemd-reso
tcp    0      0 0.0.0.0:22 0.0.0.0:* LISTEN 811/sshd: /usr/st
tcp    0      0 127.0.0.1:631 0.0.0.0:* LISTEN 4147/cupsd
tcp6   0      0 :::22 :::* LISTEN 811/sshd: /usr/st
```

Protocol	Local Address	Local Port	Foreign Address	Foreign Port	State	Process
tcp6	0	0 :::1:631	:::*		LISTEN	4147/cupsd
udp	0	0 127.0.0.53:53	0.0.0.0:*			666/systemd-reso
udp	0	0 0.0.0.0:49345	0.0.0.0:*			715/avahi-daemon
udp	0	0 0.0.0.0:631	0.0.0.0:*			4149/cups-browsec
udp	0	0 0.0.0.0:5353	0.0.0.0:*			715/avahi-daemon
udp6	0	0 :::5353	:::*			715/avahi-daemon
udp6	0	0 :::34878	:::*			715/avahi-daemon

Another classic way to collect this information is to use the `lsof` (list of open files) command. Wait a minute though, we want to get network information, not a list of who has what file open! The missing information behind this question is that in Linux, **everything** is represented as a file, including network information – remember the `proc` directory from earlier in this chapter? Let's use `lsof` to enumerate connections on TCP ports `80` and `22`:

```
$ lsof -i :443
COMMAND PID USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
firefox 4627 robv 162u IPv4  93018      0t0  TCP ubuntu:42502->ec2-44-227-121-122.us-west-2.com
$ lsof -i :22
COMMAND PID USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
ssh      5832 robv   3u  IPv4 103832      0t0  TCP ubuntu:47902->_gateway:ssh (ESTABLISHED)
```

You can see the same information, represented in slightly different ways. This is also handy in that the `lsof` command explicitly shows the direction of each conversation, which it gets from the initial **SYN** packet in the conversation (whoever sent the first **SYN** packet is the client in any TCP conversation). Why are we so focused on listening ports and processes? One answer was actually touched on earlier in this chapter – you can only have one service listening on a particular port. The classic example of this is trying to start a new website on TCP port `80`, not being aware that there's already a service listening on that port. In that case, the second service or process will usually fail to start. Now that we've explored local listening ports, along with their associated processes, let's turn our attention to remote listening ports – services listening on other hosts.

Remote port enumeration using native tools

So now we know how to work out our local services and some traffic diagnostics, how can we enumerate listening ports and services on remote hosts? The easy way is to use native tools – for instance **scp** for SFTP servers, or **ftp** for FTP servers. But what if it's some different service that we don't have an installed client for. Simple enough, the `telnet` command can be used in a pinch for this – for instance, we can telnet to a printer's admin port, running `http (tcp/80)`, and make a `GET` request for the header of the first page. Notice the garbage characters at the bottom of the listing – that's how graphics are represented on this page:

```
$ telnet 192.168.122.241 80
Trying 192.168.122.241...
Connected to 192.168.122.241.
Escape character is '^]'.
GET / HTTP/1.1
HTTP/1.1 200 OK
Server: HP HTTP Server; HP PageWide 377dw MFP - J9V80A; Serial Number: CN74TGJ0H7; Built: Thu Oct
Content-Encoding: gzip
Content-Type: text/html
Last-Modified: Thu, 15 Oct 2020 13:32:45 GMT
Cache-Control: max-age=0
Set-Cookie: sid=se2b8d8b3-e51eab77388ba2a8f2612c2106b7764a;path=/;HttpOnly;
Content-Security-Policy: default-src 'self' 'unsafe-eval' 'unsafe-inline'; style-src * 'unsafe-ir
X-Frame-Options: SAMEORIGIN
X-UA-Compatible: IE=edge
X-XSS-Protection: 1
X-Content-Type-Options: nosniff
Content-Language: en
Content-Length: 667
00W
Hs<M'M
q.[[lNj+""]s[szr}?[<|B{3v=esn"1vR?X9oI
2?i ] ) ^uF
```

Even if you don't know what to type - usually if you can connect at all with telnet, that's a good indicator that the port you are testing is open. There are a few problems with this method though – if you don't know what to type,

this isn't a foolproof way of determining whether that port is open or not – you'll get a blank page either way. Also, quite often exiting out of that session can be a problem – often `BYE`, `QUIT`, or `EXIT` will work, sometimes pressing `^c` (*Ctrl* + *c*) or `^z` (*Ctrl* + *z*) will work, but neither of those methods is 100% guaranteed. Finally, it's likely you are either looking at multiple hosts or multiple ports, or this might be just the first step in your troubleshooting. All of these factors combined make this method both clumsy and time consuming. In answer to this, we have dedicated *port scanner* tools that are purpose-built for this – `Nmap` (which we'll cover in the next section) is the most popular of these. However, if you don't happen to have one of those installed, the `nc` (*netcat*) command is your friend! Let's scan our example HP printer with `netcat`:

```
$ nc -zv 192.168.122.241 80
Connection to 192.168.122.241 80 port [tcp/http] succeeded!
$ nc -zv 192.168.122.241 443
Connection to 192.168.122.241 443 port [tcp/https] succeeded!
```

Or how about testing the first 1024 ports? If we use the following command:

```
$ nc -zv 192.168.122.241 1-1024
```

We get pages and pages of errors such as the following:

```
nc: connect to 192.168.122.241 port 1013 (tcp) failed: Connection refused
```

OK, let's try to filter those down with our friend `grep`:

```
$ nc -zv 192.168.122.241 1-1024 | grep -v refused
```

That still doesn't work – why not? The key is the word "error". `Netcat` sends errors to the special file **STDERR** (standard error), which is normal in Linux (we'll see why successful connections count as errors for this tool later in this section). That file echoes to the console, but it's not **STDOUT**, so our **grep** filter misses it entirely. How do we fix this? A bit of background on the three **STD** files or *streams* – they each have a file number associated with them:

```
STDIN 0
STDOUT 1
STDERR 2
```

By playing some games with these file numbers, we can redirect **STDERR** to **STDOUT** (so `grep` will now work for us):

```
$ nc -zv 192.168.122.241 1-1024 2>&1 | grep -v refused
Connection to 192.168.122.241 80 port [tcp/http] succeeded!
Connection to 192.168.122.241 443 port [tcp/https] succeeded!
Connection to 192.168.122.241 515 port [tcp/printer] succeeded!
Connection to 192.168.122.241 631 port [tcp/ipp] succeeded!
```

That's **exactly** what we wanted!! Not only that, we found a couple of extra ports we didn't know about! Expanding that to *all* ports, we find even more services running. Note that in our first attempt, we try to include port `0` (which is definitely seen on real networks), but `netcat` fails on that:

```
$ nc -zv 192.168.122.241 0-65535 2>&1 | grep -v refused
nc: port number too small: 0
$ nc -zv 192.168.122.241 1-65535 2>&1 | grep -v refused
Connection to 192.168.122.241 80 port [tcp/http] succeeded!
Connection to 192.168.122.241 443 port [tcp/https] succeeded!
Connection to 192.168.122.241 515 port [tcp/printer] succeeded!
Connection to 192.168.122.241 631 port [tcp/ipp] succeeded!
Connection to 192.168.122.241 3910 port [tcp/*] succeeded!
Connection to 192.168.122.241 3911 port [tcp/*] succeeded!
Connection to 192.168.122.241 8080 port [tcp/http-alt] succeeded!
Connection to 192.168.122.241 9100 port [tcp/*] succeeded!
```

We can duplicate this for UDP as well:

```
$ nc -u -zv 192.168.122.1 53
Connection to 192.168.122.1 53 port [udp/domain] succeeded!
```

However, if we scan a UDP range this can take a **very** long time – we'll also find that the UDP scan isn't very reliable. It depends on the target host replying with an ICMP port unreachable error when the test fails, which isn't always supported if you have any firewalls in the path. Let's see how long that "first 1024" scan takes when targeting a UDP port (note how we're stringing commands together using the semicolon):

```
$ date ; nc -u -zv 192.168.122.241 1-1024 2>&1 | grep succeed ; date
Thu 07 Jan 2021 09:28:17 AM PST
Connection to 192.168.122.241 68 port [udp/bootpc] succeeded!
Connection to 192.168.122.241 137 port [udp/netbios-ns] succeeded!
Connection to 192.168.122.241 138 port [udp/netbios-dgm] succeeded!
Connection to 192.168.122.241 161 port [udp/snmp] succeeded!
Connection to 192.168.122.241 427 port [udp/svrloc] succeeded!
Thu 07 Jan 2021 09:45:32 AM PST
```

Yes, a solid 18 minutes – this method is not a speed demon! Using netcat, you can also interact directly with a service, the same as in our telnet example, but without the "terminal/cursor control" type overhead that telnet brings. For instance, to connect to a web server, the syntax would be as follows:

```
# nc 192.168.122.241 80
```

Of more interest though, we can stand up a fake service, telling netcat to *listen* on a given port. This can be extremely handy if you need to test connectivity, in particular if you want to test a firewall rule, but don't have the destination host or service built yet. This syntax tells the host to listen on port 80. Using the `-l` parameter tells netcat to listen, but when your remote tester or scanner connects and disconnects, the netcat listener exits. Using the `-L` parameter is the "*listen harder*" option, which properly handles TCP connections and disconnections, leaving the listener in place. Unfortunately, the `-L` parameter and the `-e` (execute) parameter are both missing in the Ubuntu implementation of netcat. We can fake this out though – read on! Expanding on this, let's stand up a simple website using netcat!

1. First, create a simple text file. We'll make our `index.html` something like the following:

```
HTTP/1.1 200 OKThis is my simple websiteText-only club - no graphics allowed!
```

1. Now, to stand up the website, let's add a timeout of 1 second to our netcat statement, and put the whole thing into a loop so that when we exit a connection, netcat gets restarted:

```
$ while true; do cat index.html | nc -l -p 80 -q 1; done
nc: Permission denied
nc: Permission denied
nc: Permission denied
... (and so on) ...
```

1. Note how listening on port 80 fails – we had to hit `Ctrl + C` to exit from the loop. Why is that? (Hint: go back to how ports are defined in Linux, earlier in this chapter.) Let's try again with port 1500:

```
$ while true; do cat index.html | nc -l -p 1500 -q 1 ; done
```

Browsing to our new website (note that it's HTTP, and note the `:1500` used to set the destination port), we now see the following:

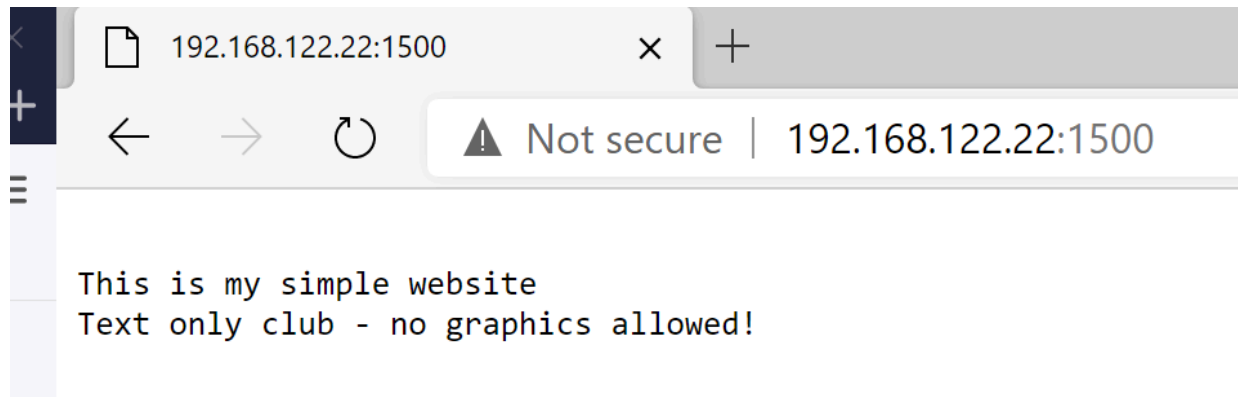


Figure 3.8 – A Netcat simple website

Back on the Linux console, you'll see that netcat echoes the client GET request and the browser's User-Agent string. You'll see the entire HTTP exchange (from the server's perspective):

```
GET / HTTP/1.1
Host: 192.168.122.22:1500
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.182 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
```

1. Making this a bit more active, let's make this a website that tells us the date and time:

```
while true; do echo -e "HTTP/1.1 200 OK\n\n $(date)" | nc -l -p 1500 -q 1; done
```

Browsing to that site now gives us the current date and time:

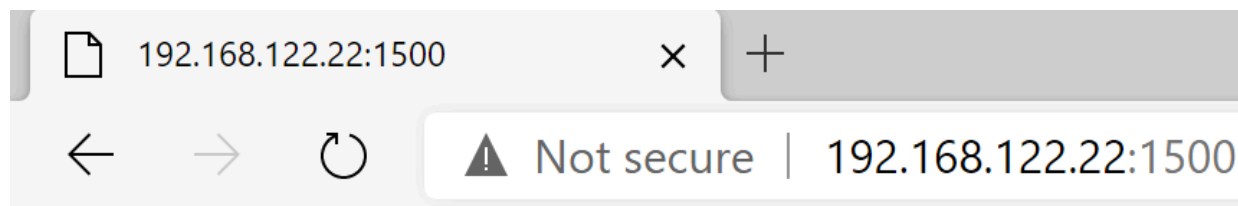


Figure 3.9 – A more complex Netcat website – Adding time and date

1. Or, using apt-get to install the fortune package, we can now add a proverb to give us some *timely* wisdom:



Figure 3.10 – Adding a Fortune to the Netcat website

2. We can also transfer a file using netcat. At the receiving end, we'll listen on port 1234 , and send our output to out.file , again using redirection:

```
nc -l -p 1234 > received.txt
```

1. At the sending end, we'll connect to that service for 3 seconds, and send it sent-file.txt . We'll get our input by using redirection in the opposite direction, using the < operator:

```
nc -w 3 [destination ip address] 1234 < sent-file.txt
```

1. Now, back at the receiver, we can cat the resulting file:

```
$ cat received.txt
Mr. Watson, come here, I want to see you.
```

This illustrates that netcat can be a valuable troubleshooting tool, but it can be complex to use depending on what you are trying to accomplish. We can use netcat to be a simple proxy, as a simple chat application, or to present a complete Linux shell – all kinds of things that are handy to a network administrator (or a penetration tester for that matter). That wraps up the basics of netcat. We've used netcat to enumerate local ports, connect to and interact with remote ports, stand up some fairly complex local services, and even transfer files. Now let's look at Nmap, a much faster and more elegant method of enumerating remote ports and services.

Remote port and service enumeration – Nmap

The tool that is most widely used to scan network resources is **Nmap** (short for **Network Mapper**). Nmap started as a simple port scanner tool, but is well past that set of simple functions now, with a long list of functions. First of all, Nmap is not installed by default on a basic Ubuntu workstation (though it is included by default in many other distros). To install it, run `sudo apt-get install nmap`. As we go forward working with nmap, please try the various commands we're using in our examples. You'll likely see similar results, and will learn this valuable tool along the way. You may learn lots about your network along the way too!

Important note

One very important caveat on the advice of "try this out yourself." Nmap is a pretty innocuous tool, it almost never causes network problems. However, if you are running this against a production network, you will want to get a feel for that network first. There are several classes of gear that have particularly "rickety" network stacks – older medical devices for instance, as well as older **Industrial Control Systems (ICS)** or **Supervisory Control and Data Acquisition (SCADA)** gear.

In other words, if you are in a hospital, a factory, or a utility, take care! Running any network mapping against your production networks can cause issues with production equipment.

You likely do still want to do this, but test against known "idle" gear first so you know that when you scan the "real" network, you have some assurance that you aren't going to cause a problem. And please (**please**), if you are on a healthcare network, **don't ever** scan or test anything that's attached to a person!

A second (legal) caveat – don't scan things without permission. If you are on your home or lab network, that's a great place to play with assessment tools such as nmap or more aggressive security assessment tools. However, if you are at work, even if you're sure that you're not going to cause problems, you'll want to get permission in writing first.

Scanning internet hosts that you don't own or don't have written permission to scan is very much illegal. Many would consider it pretty innocuous, and in most cases scanning from the internet is simply considered "white noise" by most companies (most organizations are scanned dozens or hundreds of times per hour). However, as soon as liability is involved the story can become very different very quickly for someone scanning "just for fun". If you interrupt service, trip a SIEM alert or if you find an application vulnerability that exposes that application owner to legal action, it's not uncommon for lawyers to get involved, then see the lawyers pointed at the poor person who found the problem, not the ones who caused it.

Always keep in mind that the proverb "the difference between a criminal and an information security professional is a signed contract" is repeated so often because it is 100% true.

With all that behind us, let's get more familiar with this great tool! Try running `man nmap` (remember the `man` command?) – there's lots of good information in the man pages for nmap, in fact that's the full/official documentation. Once we get more familiar with the tool though, you may find the help text quicker to use. Normally you know (more or less) what you are looking for, so you can search for it using the `grep` command, for instance: `nmap - -help | grep <my_search_string>`. In the case of nmap, you can dispense with the standard `- - help` option, as the default output of nmap with no arguments is the help page. So, to find out how to do a ping scan – that is, to ping everything in the range (which I always forget the syntax for) – you would search as follows:

```
$ nmap | grep -i ping
-sn: Ping Scan - disable port scan
-PO[protocol list]: IP Protocol Ping
$
```

How do we proceed? Nmap wants to know what you want mapped – in this case I'll map the `192.168.122.0/24` subnet:

```
$ nmap -sn 192.168.122.0/24
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-05 13:53 PST
Nmap scan report for _gateway (192.168.122.1)
Host is up (0.0021s latency).
Nmap scan report for ubuntu (192.168.122.21)
Host is up (0.00014s latency).
Nmap scan report for 192.168.122.51
Host is up (0.0022s latency).
Nmap scan report for 192.168.122.128
Host is up (0.0027s latency).
Nmap scan report for 192.168.122.241
Host is up (0.0066s latency).
Nmap done: 256 IP addresses (5 hosts up) scanned in 2.49 seconds
```

So that's a quick scan that tells us every IP that is currently active on our subnet. Now let's look for services. Let's start by looking for anything running `tcp/443` (which you may recognize as HTTPS). We'll use the command `nmap -p 443 --open 192.168.122.0/24`. There are two things to note in this command. First of all, we specified the port with the `-p` option. By default Nmap scans for TCP ports using a **SYN** scan. `nmap` sends a **SYN** packet, and waits to get back a **SYN-ACK** packet. If it sees that, the port is open. If it gets a port unreachable response, then the port is considered closed. If we wanted a full connect scan (where the entire three-way handshake completes), we could have specified `-sT`. Next, we see a `--open` option. This indicates "only show me open ports". Without this we would see closed ports as well as "filtered" ports (which typically means nothing at all came back from the initial packet). If we wanted more detail on why a port might be considered open, closed, or filtered, we would remove the `--open` option, and add the `--reason` option.

```
$ nmap -p 443 --open 192.168.122.0/24
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-05 13:55 PST
```

```

Nmap scan report for _gateway (192.168.122.1)
Host is up (0.0013s latency).
PORT      STATE SERVICE
443/tcp    open  https
Nmap scan report for 192.168.122.51
Host is up (0.0016s latency).
PORT      STATE SERVICE
443/tcp    open  https
Nmap scan report for 192.168.122.241
Host is up (0.00099s latency).
PORT      STATE SERVICE
443/tcp    open  https
Nmap done: 256 IP addresses (5 hosts up) scanned in 2.33 seconds

```

To scan UDP ports, we would use the same syntax, but add the `-sU` option.

```

$ nmap -sU -p 53 --open 192.168.122.0/24
You requested a scan type which requires root privileges.
QUITTING!

```

Oops – because we're scanning for UDP ports, Nmap needs to run with root privileges (using `sudo`). This is because it needs to put the sending interface into *promiscuous mode* so that it can capture any packets that might be returned. This is because there is no Layer 5 concept of a *session* in UDP like we have in TCP, so there is no layer 5 connection between the sent and received packets. Depending on what command-line arguments are used (not just for UDP scans), Nmap may need elevated rights. In most cases if you are using Nmap or a similar tool, you'll find yourself using `sudo` a fair bit. Also notice at this point that we're starting to see MAC addresses of the hosts that are up. This is because the scanned hosts are in the same subnet as the scanner, so that information is available. Nmap uses the MAC addresses' OUI section to identify the vendor of each network card:

```

$ sudo nmap -sU -p 53 --open 192.168.122.0/24
[sudo] password for robv:
Sorry, try again.
[sudo] password for robv:
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-05 14:04 PST
Nmap scan report for _gateway (192.168.122.1)
Host is up (0.00100s latency).
PORT      STATE SERVICE
53/udp     open  domain
MAC Address: 00:0C:29:3B:73:CB (VMware)
Nmap scan report for 192.168.122.21
Host is up (0.0011s latency).
PORT      STATE SERVICE
53/udp     open|filtered domain
MAC Address: 00:0C:29:E4:0C:31 (VMware)
Nmap scan report for 192.168.122.51
Host is up (0.00090s latency).
PORT      STATE SERVICE
53/udp     open|filtered domain
MAC Address: 00:25:90:CB:00:18 (Super Micro Computer)
Nmap scan report for 192.168.122.128
Host is up (0.00078s latency).
PORT      STATE SERVICE
53/udp     open|filtered domain
MAC Address: 98:AF:65:74:DF:6F (Unknown)
Nmap done: 256 IP addresses (23 hosts up) scanned in 1.79 seconds

```

A few more things to note about this scan:

- The initial scan attempt failed – note that you need root rights to do many scans within Nmap, not just the ones that are against UDP ports. To get the results it does, in many cases the tool crafts packets itself rather than using the standard OS services to do that, and it also usually needs rights to capture packets that are returned by your target hosts – so nmap needs elevated privileges for both of those operations.
- We see lots more statuses indicating `open|filtered` ports. UDP is particularly prone to this – since there's no **SYN/SYN-ACK** type of handshake, you send a **UDP** packet, and you may not get anything back – this doesn't necessarily mean the port is down, it might mean that your packet was processed by the remote service, and no acknowledgment was sent (some protocols are like that). Or in many cases it might mean that

the port is not up, and the host does not properly return an ICMP Port Unreachable error message (ICMP Type 1, Code 3).

To get more detail, let's use the `sv` option, which will probe the ports in question and get more information on the service itself. In this case, we'll see that `192.168.122.1` is identified positively as open, running the `domain` service, with a service version listed as `generic dns response: NOTIMP` (this indicates that the server does not support the DNS UPDATE function, described in *RFC 2136*). The *service fingerprint* signature following the service information can be helpful in further identifying the service if the Nmap identification isn't conclusive. Notice also that for other hosts, the `REASON` is listed as `no-response`. If you know the protocol, you can usually make good inferences in those situations. In the case of scanning for DNS, `no-response` means that there's no DNS server there or the port is closed. (or possibly it's open with some oddball service other than DNS running on it, which is highly unlikely). Also note that this scan took a solid 100 seconds, roughly 50 times slower than our original scan:

```
$ sudo nmap -sU -p 53 --open -sV --reason 192.168.122.0/24
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-05 14:17 PST
Nmap scan report for _gateway (192.168.122.1)
Host is up, received arp-response (0.0011s latency).
PORT      STATE SERVICE REASON          VERSION
53/udp open  domain  udp-response ttl 64 (generic dns response: NOTIMP)
1 service unrecognized despite returning data. If you know the service/version, please submit the
SF-Port53-UDP:V=7.80%I=7%D=1/5%Time=5FF4E58A%P=x86_64-pc-linux-gnu%r(DNSVe
SF:rsionBindReq,1E,"\0\06\81\85\0\01\0\0\0\0\0\0\0\07version\04bind\0\
SF:0\10\0\03")%r(DNSStatusRequest,C,"\0\0\90\04\0\0\0\0\0\0\0")%r(NB
SF:TStat,32,"\x80\xf0\x80\x95\0\01\0\0\0\0\0\0\0x20CAAAAAAAAAAAAAAAAAAAAA
SF:AAAAAAAA\0\0!\0\01");
MAC Address: 00:0C:29:3B:73:CB (VMware)
Nmap scan report for 192.168.122.51
Host is up, received arp-response (0.00095s latency).
PORT      STATE SERVICE REASON          VERSION
53/udp open|filtered domain  no-response
MAC Address: 00:25:90:CB:00:18 (Super Micro Computer)
Nmap scan report for 192.168.122.128
Host is up, received arp-response (0.00072s latency).
PORT      STATE SERVICE REASON          VERSION
53/udp open|filtered domain  no-response
MAC Address: 98:AF:65:74:DF:6F (Unknown)
Nmap scan report for 192.168.122.171
Host is up, received arp-response (0.0013s latency).
PORT      STATE SERVICE REASON          VERSION
53/udp open|filtered domain  no-response
MAC Address: E4:E1:30:16:76:C5 (TCT mobile)
Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 256 IP addresses (24 hosts up) scanned in 100.78 seconds
```

Let's try a `sv` verbose service scan of just `192.168.122.1`, port `tcp/443` – we'll see that Nmap does a pretty good job of identifying the web server running on that host:

```
root@ubuntu:~# nmap -p 443 -sV 192.168.122.1
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-06 09:02 PST
Nmap scan report for _gateway (192.168.122.1)
Host is up (0.0013s latency).
PORT      STATE SERVICE VERSION
443/tcp open  ssl/http nginx
MAC Address: 00:0C:29:3B:73:CB (VMware)
Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 12.60 seconds
```

Trying the same against `192.168.122.51`, we see that the service is properly identified as the VMware ESXi 7.0 management interface:

```
root@ubuntu:~# nmap -p 443 -sV 192.168.122.51
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-06 09:09 PST
Nmap scan report for 192.168.122.51
Host is up (0.0013s latency).
PORT      STATE SERVICE VERSION
```

```
443/tcp open  ssl/https VMware ESXi SOAP API 7.0.0
MAC Address: 00:25:90:CB:00:18 (Super Micro Computer)
Service Info: CPE: cpe:/o:vmware:ESXi:7.0.0
Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 140.48 seconds
```

Now that we're experts at scanning ports with various options, let's expand on this. Nmap allows us to run scripts against any open ports it finds – this can be a great time saver!

Nmap scripts

So far we've just looked at port scanning – Nmap is much more than that though. A fully featured scripting engine is available to process the packets or the output of Nmap, based on LUA (a text-based interpreted language). We won't dig into LUA in this book to any great extent, but Nmap does come with several pre-written scripts, some of which are invaluable to a network administrator. For instance, consider the SMB version information. Microsoft has been strongly recommending that SMBv1 be retired for years now, peaking just before the EternalBlue and EternalRomance vulnerabilities in SMBv1 were used by the WannaCry/Petya/NotPetya families of malware in 2017. While SMBv1 has been effectively retired by making it hard to even enable in newer Windows versions, we still see SMBv1 in corporate networks – whether on older server platforms or on older Linux-based appliances that implement SMBv1 in their SAMBA service. Scanning for this couldn't be easier using the `smb-protocols` script. Before you use any script, it's handy to open the script to see exactly what it does, and how it needs to be called by Nmap (what ports or arguments it might need). In this case, the `smb-protocols` text gives us the usage, as well as what to expect in the output:

```
-- @usage nmap -p445 --script smb-protocols <target>
-- @usage nmap -p139 --script smb-protocols <target>
--
-- @output
-- | smb-protocols:
-- |   dialects:
-- |     NT LM 0.12 (SMBv1) [dangerous, but default]
-- |     2.02
-- |     2.10
-- |     3.00
-- |     3.02
-- |     3.11
--
-- @xmloutput
-- <table key="dialects">
-- <elem>NT LM 0.12 (SMBv1) [dangerous, but default]</elem>
-- <elem>2.02</elem>
-- <elem>2.10</elem>
-- <elem>3.00</elem>
-- <elem>3.02</elem>
-- <elem>3.11</elem>
-- </table>
```

Let's scan some specific hosts in a target network to see more. We'll just show the output from one example host that has the SMBv1 protocol running. Note that from the hostname it seems to be a **Network Attached Storage (NAS)** device, so is likely Linux- or BSD-based under the hood. From the OUI we can see the brand name of the NIC, which gives us even more specific information:

```
nmap -p139,445 --open --script=smb-protocols 192.168.123.0/24
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-06 12:27 Eastern Standard Time
Nmap scan report for test-nas.defaultroute.ca (192.168.123.1)
Host is up (0.00s latency).
PORT      STATE SERVICE
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
MAC Address: 00:D0:B8:21:89:F8 (Iomega)
Host script results:
| smb-protocols:
|   dialects:
|     NT LM 0.12 (SMBv1) [dangerous, but default]
|     2.02
```

```
| 2.10
| 3.00
| 3.02
|_ 3.11
```

Or you can scan for the Eternal* vulnerabilities directly using the `smb-vuln-ms17-010.nse` script (showing just one host as an example). Scanning that same host, we see that even though SMBv1 is enabled, that specific vulnerability is not in play. It's still strongly recommended that SMBv1 be disabled though, as there's a whole list of vulnerabilities that SMBv1 is susceptible to, not just `ms17-010`. Scrolling down a bit further in the list, our second example host does have that vulnerability. From the hostname, we see that this is likely a business-critical host (running BAAN), so we'd much rather have this server fixed than ransomware. Looking at the production application on that host, there's really no reason for SMB to be exposed at all to most users – really only system or application administrators should be mapping a drive to this host, and users would connect to it through its application port. The recommendation for this is to clearly to patch the vulnerability (this host likely hasn't been patched in several years), but also to firewall that SMB service away from most users (or disable that service if it isn't used by the administrators):

```
Starting Nmap 7.80 ( https://nmap.org ) at 2021-01-06 12:32 Eastern Standard Time
Nmap scan report for nas.defaultroute.ca (192.168.123.11)
Host is up (0.00s latency).
PORT      STATE SERVICE
139/tcp    open  netbios-ssn
445/tcp    open  microsoft-ds
MAC Address: 00:D0:B8:21:89:F8 (Iomega)
Nmap scan report for baan02.defaultroute.ca (192.168.123.77)
Host is up (0.00s latency).
PORT      STATE SERVICE
139/tcp    open  netbios-ssn
445/tcp    open  microsoft-ds
MAC Address: 18:A9:05:3B:ED:EC (Hewlett Packard)
Host script results:
| smb-vuln-ms17-010:
|   VULNERABLE:
|     Remote Code Execution vulnerability in Microsoft SMBv1 servers (ms17-010)
|       State: VULNERABLE
|       IDs: CVE:CVE-2017-0143
|       Risk factor: HIGH
|         A critical remote code execution vulnerability exists in Microsoft SMBv1
|           servers (ms17-010).
|
|     Disclosure date: 2017-03-14
|     References:
|       https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-0143
|       https://technet.microsoft.com/en-us/library/security/ms17-010.aspx
|_      https://blogs.technet.microsoft.com/msrc/2017/05/12/customer-guidance-for-wannacrypt-att
```

Nmap installs with hundreds of scripts. If you are looking for something specific, especially if you can't determine it from just a port scan, then using one or more nmap scripts is often the easiest way to go. Just keep in mind that if you are looking for a "rogue" host, say a DHCP server, you'll find your production host as well as any unwanted instances. Note that many of these rely on you to include the right port numbers in the scan. The "broadcast" style scripts will usually only scan the subnet that your scanner is on, so scanning a remote subnet likely means "borrowing" or placing a host on that subnet. Many of the core network services discussed in this list are covered in later chapters in this book, including DNS, DHCP, and the like. Keep in mind (again) that scanning without authorization is never in your best interest – get written permission first! There are hundreds of scripts that come with nmap, and hundreds more available with a quick internet search. In Linux these scripts can be found at `/usr/share/nmap/scripts`. Some of the pre-packaged nmap scripts I find handiest on a production network include the following:

What are we looking for?

The **MTU (Message Transfer Unit)** size over a communications path.

Scripts to use

`Path-mtu`

In other words, the largest payload that can be encapsulated into a frame and sent over the link. This is often used in troubleshooting communications over a **Wide Area Network (WAN)** or especially **Virtual Private Network (VPN)** links.

For Unexpected, malicious, or misconfigured network infrastructure:

Un-inventoried or malicious routers.

broadcast-eigrp-discovery

A malicious router can be used in **MiTM (Machine in the Middle)** attacks to redirect traffic – for instance to a fake bank website to steal users' credentials.

broadcast-igmp-discovery

broadcast-ospf2-discover

broadcast-rip-discover

broadcast-ripng-discover

broadcast-wpad-discover

Unwanted or malicious proxy servers.

The *WPAD attack* is often used in credential theft attacks – a malicious proxy server can be used even more easily than a malicious router to steal credentials.

snmp-info

SNMP services that are misconfigured.

Hint: look especially for the SNMPv2 strings of `public` or `private`.

For Server problems and malicious services:

DNS servers that you didn't know were there.

broadcast-dns-service-discovery

dns-srv-enum

dns-recursion

DNS servers with recursion enabled.

This is likely desired for an internal DNS server, but usually not so much for an internet-facing DNS server (we'll discuss this in depth in a later chapter).

Rogue DHCP servers.

dhcp-discover

Often these will be "*I brought it from home*" wireless routers or switches, but a rogue DHCP server can definitely be used in combination with an internet-based malicious proxy server for credential theft.

broadcast-dhcp-discover

broadcast-dhcp6-discover

For Pirated software, "shadow IT", malicious, or otherwise unexpected servers:

Unlicensed or "shadow" database servers.	<code>broadcast-ms-sql-discover</code>
Often just an <code>sv</code> scan of the default listening port for the target database is sufficient, but these scripts will give you more information if you need it.	<code>broadcast-sybase-asa-discover</code> <code>oracle-tns-version</code>
These scripts are typically used to find database servers that people thought were retired, or developers "pirating" a database instance for testing.	<code>broadcast-db2-discover</code> <code>couchdb-databases</code>
Finding a MongoDB or CouchDB server outside of the IT environment isn't entirely unusual – these are favorites of marketing departments. The risk there is that they are often chock-full of confidential client information and are often not well secured (finding these with no admin password applied isn't uncommon).	<code>mongodb-info</code>

Personal Jenkins servers (used for source code control and DevOps pipelines).	<code>broadcast-jenkins-discover</code>
---	---

Finding an unexpected Jenkins server, for instance running on a developer's machine, can mean real trouble in a larger development team.

For Workstation issues:

Hosts that have Universal Plug and Play (UPnP) enabled.	<code>broadcast-upnp-info</code>
Hosts with Local Link Multicast Name Resolution (LLMNR) enabled. This service (switched on by default in many Windows versions) is a layer 2 replacement for DNS. As it's also used in credential theft attacks involving Windows clients and SMB, you likely want to disable it wherever you find it.	<code>llmnr-resolve</code>

For Network perimeter problems:

Unwanted VPN hosts at your internet edge, or VPN hosts that support IKEv1. (See also <code>dns-recursion</code>).	<code>ike-version</code> <code>http-cisco-anyconnect</code>
--	--

For Miscellaneous server or workstation issues:

Previously uninventoried certificates or expiring certificates. (Hint: don't limit your search to just port <code>tcp/443</code> , as certificates are used for all sorts of services besides HTTPS web servers.) This is a good scan to run regularly, as having a critical certificate expire unexpectedly can definitely ruin your day!	<code>ssl-cert</code> <code>ssl-date</code>
SSL or TLS services with retired or unwanted versions. SSL and TLS version 1 should be well and truly retired.	<code>ssl-dh-param</code> <code>ssl-enum-ciphers</code>
RDP servers with low encryption values set.	<code>rdp-enum-encryption</code>
SSH servers with low encryption values set. SSH version 1 in particular is easily attacked, so should be updated if found.	<code>ssh2-enum-algo</code> <code>sshv1</code>
Other typically unwanted services on an organization's network.	<code>bitcoin-info</code>

This summarizes the various uses of Nmap. Where Nmap doesn't do so well is in larger networks – for instance, in `/8` or `/16` networks, or some of the really large IPv6 networks. For these networks, a faster tool is needed. Let's explore the MASSCAN tool for these uses.

Are there limits to Nmap?

The primary limit to Nmap is performance. As the network size grows, Nmap will (of course) take longer and longer to complete any scan you are running. This is often not an issue, but in a production network, if your scan starts at 8 A.M. and ends sometime the next day, there's likely a sizable amount of time where devices are mostly powered off or are disconnected, so the scan's usefulness will suffer. This gets especially pronounced when you are on very large networks – for instance, as your subnet mask shrinks or your subnet count grows, scan times for Nmap can grow to hours, days, or weeks. Equally, on IPv6 networks it's common to see thousands, hundreds of thousands, or even millions of addresses, which can translate to Nmap scan times of years or decades. There are two ways to help resolve this.

- First, if you read the Nmap `man` page, there are some parameters to speed things up – you can adjust parallelism (how many operations can run at once), host timeouts, round trip timeouts, and the delay wait between operations. These are fully explained on the `man` page, and are discussed in more depth here: <https://nmap.org/book/man-performance.html>.
- Or, you can look at a different tool. Robert Graham maintains the **MASSCAN** tool, which is specifically built for high-performance scanning. With enough bandwidth and horsepower, it can scan the entire IPv4 internet in under 10 minutes. Version 1.3 of this tool adds IPv6 support. The MASSCAN syntax is similar to Nmap's, but there are some things to watch out for when using this faster tool. The tool, as well as its documentation and "gotchas" are posted here: <https://github.com/robertdavidgraham/masscan>.

For very large networks, a common approach would be to use MASSCAN (or Nmap tuned for faster scans) for an initial set of scans. The output from that cursory scan can then be used to "feed" the next tool, whether that is Nmap or possibly some other tool, perhaps a security scanner such as Nessus or OpenVAS. "Chaining" tools together like this maximizes the strengths of each to deliver the best outcomes in the shortest time. All tools have their limits though, and IPv6 networks remain a challenge for scanning tools. Unless you can limit the scope somehow, IPv6 will quickly reach the limits of network bandwidth, time, and memory on the scanning host. Tools such as DNS harvesting, harvesting the ARP cache of routers or MAC Address tables of switches, or other neighbor discovery tools can help here – if you can identify which hosts are actually active before scanning for services, that of course reduces the target addresses significantly, hopefully back down to manageable volumes. With port scanning behind us, let's leave the wired world and explore troubleshooting with Linux on wireless networks.

Wireless diagnostic operations

Diagnostic tools in wireless networks are generally concerned with finding areas of low signal strength and interference – things that cause problems for the folks using your wireless network. There are a few excellent wireless tools that are Linux-based, but we'll discuss Kismet, Wavemon, and LinSSID. All three tools are free, and all can be installed with the standard `apt-get install <package name>` command. If you expand your tool search to include attack-type tools or commercial products, that list obviously grows much bigger. Before starting these installations, be aware that most wireless tools of this type require an optional feature on wireless adapters called monitor mode. This is similar to promiscuous mode in wired adapters, except that monitor mode is optional. So in the interest of shaving off a few pennies on manufacturing and development, many NIC manufacturers have not implemented this feature. To see the current status of what is supported or not, this link can be helpful. Or just google your NIC's model and the term "monitor mode". Without this mode, Kismet in particular simply won't work. Kismet is one of the older wireless tools available for Linux. My first exposure to it was as an information security tool, highlighting the fact that "hidden" wireless SSIDs were in fact not hidden at all! You will find that kismet cannot be installed immediately using apt, as it has been recently removed from the default ubuntu repositories. Full instructions for installation on various Linux distros can be found here: <https://www.kismetwireless.net/packages/> For our Ubuntu station, let's add the kismet repo to the list for your workstation. Since this application manipulates your wireless card, these commands in particular will be version and distro specific, and do change as kismet gets updated. Be sure to reference the kismetwireless link above to get the correct links for your distro at time you are installing. These commands are fairly complex, be sure to reference the kismet link mentioned earlier and use copy/paste, a simple typo at this step will mean that your installation will fail:

```
wget -O - https://www.kismetwireless.net/repos/kismet-release.gpg.key --quiet | gpg --dearmor | :
echo 'deb [signed-by=/usr/share/keyrings/kismet-archive-keyring.gpg] https://www.kismetwireless.r
```

Next, run the usual installation commands:

```
sudo apt update
sudo apt install kismet
```

The install line will ask if you want Kismet to install with suid-root helpers. This allows kismet to run without root privileges, as certain executables (those that do packet captures and manipulate the wireless interface for instance) are granted root rights. This configuration is generally more secure, so select the default "Yes" at this prompt. Now, add your Linux account to the kismet group to allow setuid rights above to work:

```
sudo usermod -a -G kismet <yourusername>
```

Group membership is collected during login, so you'll need to logout / login for this to take affect. To run the tool, use the following command:

```
$ kismet -c <your wireless card name>
```

If you don't know your wireless card name, run:

```
robv@ubuntu:~$ ip ad | grep mtu
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group default c
3: wlx90f6520f6b4f: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group c
```

This command just gives us the first line of output for each interface. We see that interface 3 (wlx90f6520f6b4f) is new, and starts with the tell-tale "wl" that indicates a wireless card. Now that kismet is running, open your browser and point it to <http://localhost:2501> to access the Kismet client. Follow the prompts to create your application userid and password – this is used to access the kismet application, and can be different than your Linux login credentials. In the display that appears, you'll see the various SSIDs, and the BSSIDs of the access points that are transmitting them. As you scroll through this list, you'll see the channel and encryption types used for each SSID, the speed that your laptop understands that it can negotiate on that SSID, and also all of the client stations on that SSID. Each client will have its MAC address, frequency, and packet count shown. This information is all sent in cleartext as part of each client's association process and continued connection "handshaking".

SSID	Length	Last Seen	Encryption	# Probing	# Responding	# Advertising
WPA2 WPA2-PSK AES-CCM	10	Apr 05 2024 16:21:55	WPA2 WPA2-PSK AES-CCM	0	3	5
WPA2 WPA2-PSK AES-CCM	29	Apr 05 2024 16:21:29	WPA2 WPA2-PSK AES-CCM	0	1	1
WPA2 WPA2-PSK AES-CCM	10	Apr 05 2024 16:20:15	WPA2 WPA2-PSK AES-CCM	0	1	1
WPA2 WPA2-PSK AES-CCM	13	Apr 05 2024 16:21:53	WPA2 WPA2-PSK AES-CCM	1	1	2
WPA2 WPA2-PSK AES-CCM	6	Apr 05 2024 16:21:53	WPA2 WPA2-PSK AES-CCM	0	1	3
WPA2 WPA2-PSK AES-CCM	8	Apr 05 2024 16:20:40	WPA2 WPA2-PSK AES-CCM	0	0	1
None / Open	32	Apr 05 2024 16:21:53	None / Open	2	0	0
WPA2 WPA2-PSK AES-CCM	8	Apr 05 2024 16:16:48	WPA2 WPA2-PSK AES-CCM	0	0	1
WPA2 WPA2-PSK AES-CCM	7	Apr 05 2024 16:21:53	WPA2 WPA2-PSK AES-CCM	3	2	2
WPA2 WPA2-PSK AES-CCM	4	Apr 05 2024 16:21:54	WPA2 WPA2-PSK AES-CCM	1	4	4
None / Open	10	Apr 05 2024 16:20:39	None / Open	1	0	0
WPA2 WPA2-PSK AES-CCM	13	Apr 05 2024 16:06:42	WPA2 WPA2-PSK AES-CCM	0	0	1
WPA2 WPA2-PSK AES-CCM	7	Apr 05 2024 16:10:29	WPA2 WPA2-PSK AES-CCM	1	0	1
wlg01	5	Apr 05 2024 16:18:15	WPA2 WPA2-PSK AES-CCM	1	4	4

Showing 1 to 14 of 14 entries

Previous 1 Next

Figure 3.10: Kismet Display showing local clients, with their SSIDs, BSSIDs and encryption algorithms.

Since your wireless adapter can only "see" one wireless channel at a time, the information presented is collected by "hopping" between channels. Collecting this information has become more and more challenging over time. In the early days of wireless, most signals were broadcast "omnidirectionally", but newer access points can use multiple antennas to direct signal towards clients in a process called "**beamforming**". What this means for tools of this type is that you could be right beside a radio, but if all the clients are on the other side of the radio, you may not see the traffic at all. Back to the SSID screen, let's look more closely at the SSID "**WLG01**". Looking deeper at this SSID, we see 2 APs advertising this SSID – they both show up twice because they're both advertising on both the 2.4GHz and the 5GHz bands.

Wi-Fi (802.11) SSIDs	
SSID ?	wlg01 (5 characters)
First Seen	Apr 05 2024 16:06:27
Last Seen	Apr 05 2024 16:18:15
Encryption ?	WPA2 WPA2-PSK AES-CCM
Advertising APs ?	
▶ B0:26:80:ED:2E:41 - B0:26:80:ED:2E:41 - WPA2 WPA2-PSK AES-CCM	
▶ B0:26:80:ED:2E:4E - B0:26:80:ED:2E:4E - WPA2 WPA2-PSK AES-CCM	
▶ CC:16:7E:80:79:31 - CC:16:7E:80:79:31 - WPA2 WPA2-PSK AES-CCM	
▶ CC:16:7E:80:79:3E - CC:16:7E:80:79:3E - WPA2 WPA2-PSK AES-CCM	
Responding APs ?	
▶ B0:26:80:ED:2E:41 - B0:26:80:ED:2E:41 - WPA2 WPA2-PSK AES-CCM	
▶ B0:26:80:ED:2E:4E - B0:26:80:ED:2E:4E - WPA2 WPA2-PSK AES-CCM	
▶ CC:16:7E:80:79:31 - CC:16:7E:80:79:31 - WPA2 WPA2-PSK AES-CCM	
▶ CC:16:7E:80:79:3E - CC:16:7E:80:79:3E - WPA2 WPA2-PSK AES-CCM	
Probing devices ?	
▼ F0:EF:86:0F:5D:70 - F0:EF:86:0F:5D:70	
Probing Device	View Device Details
MAC	F0:EF:86:0F:5D:70 (Google, Inc.)
Name	F0:EF:86:0F:5D:70
Type	Wi-Fi Client

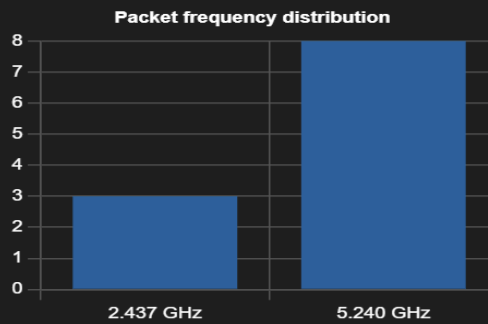
Figure 3.11: An Example SSID show in Kismet, with advertising and responding APs.

We see one device attached, with an OUI that indicates that Google is the manufacturer (it happens to be a Google Nest thermostat). Looking at details on this device, we see a myriad of other information, including signal strength, how much of it's data is being transported over 2.4 or 5 GHz, what wireless channel it's associated on and how much data it has moved.

Device: F0:EF:86:0F:5D:70

▼ Device Info

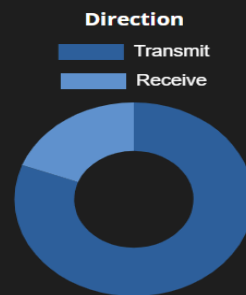
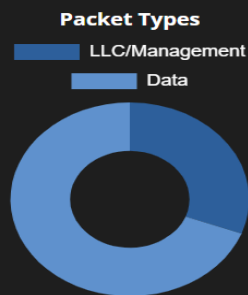
Name ?	F0:EF:86:0F:5D:70 ?
Notes ?	Empty
MAC Address ?	F0:EF:86:0F:5D:70 ?
Manufacturer ?	Google, Inc.
Type	Wi-Fi Client
First Seen	Fri Apr 05 2024 16:06:58 GMT-0400 (Eastern Daylight Time)
Last Seen	Fri Apr 05 2024 16:18:25 GMT-0400 (Eastern Daylight Time)
Frequencies	
Channel ?	1
Main Frequency ?	2.437 GHz



Signal

Monitor Signal	Monitor
Latest Signal ?	-72 dbm
Min. Signal ?	-82 dbm
Max. Signal ?	-70 dbm

Packets



Total Packets ?	26
Rx Packets ?	5
Tx Packets ?	21
LLC/Management ?	8
Error/Invalid ?	0
Data ?	18
Encrypted ?	0
Filtered ?	0
Data Transferred ?	2.34 KB

► Wi-Fi (802.11)

► Packet Graphs

► Seen By

Figure 3.12: A wireless device displayed in Kismet (in the case a client)

Even historic traffic activity is available:

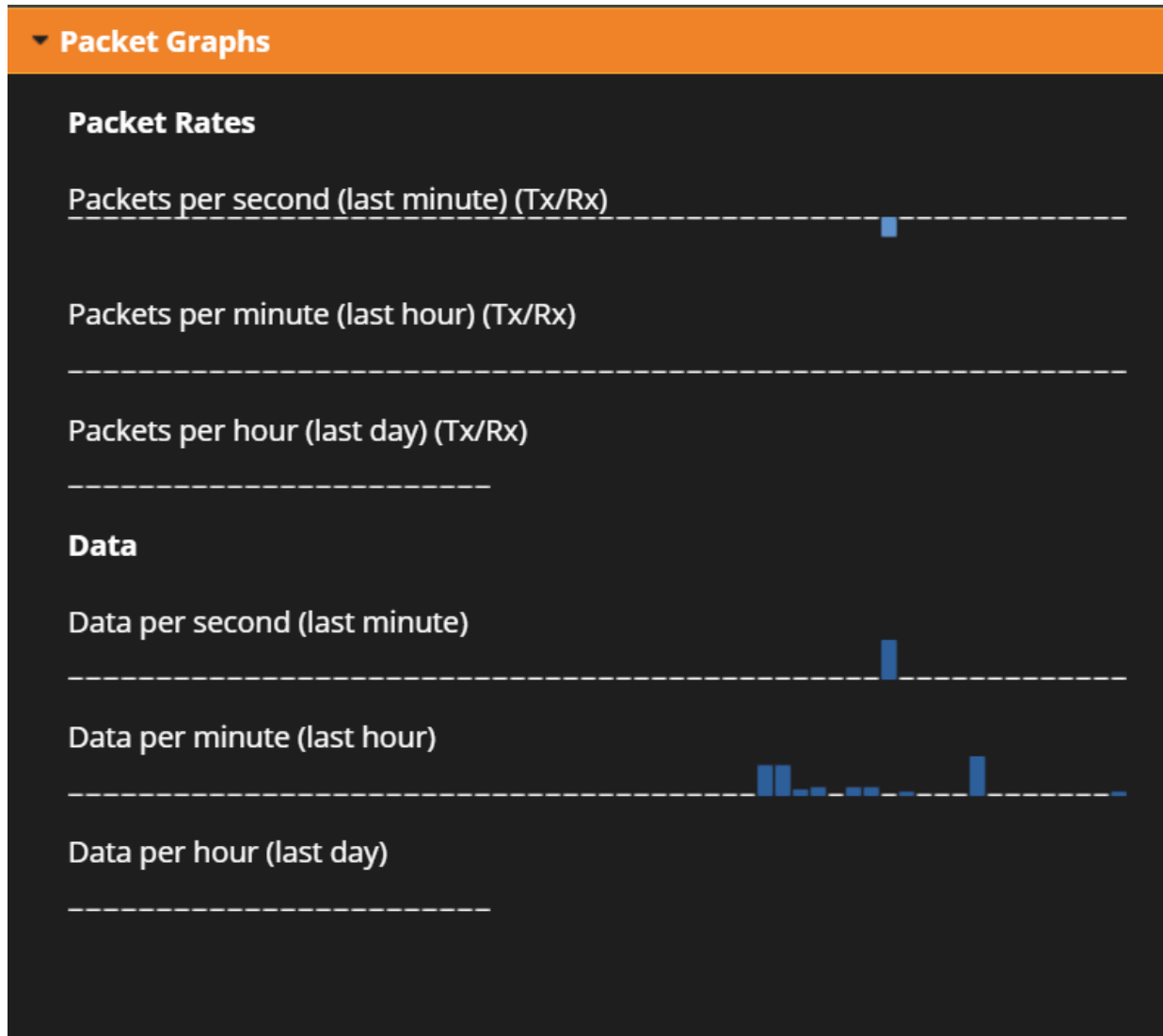


Figure 3.13: Packet rates shown in Kismet

The "security gotcha" that is inherent in modern Wireless / WiFi communication is that it's easy for Kismet to harvest all of this information from the wireless management frames that establish and maintain the wireless connection for this device. These management frames are not encrypted, all of this information is free for the taking, all you have to do is look for it. Finally, this SSID is "hidden" or "cloaked". This protection only means that the Access Point doesn't advertise it for all to see, but the information is all there as soon as one device connects. Kismet never connects to any of the SSIDs or participates in any wireless negotiations, it's just a "good listener". We can see that this SSID is cloaked in the "Wi-Fi" screen for this device, where the "Last Beaconsed SSID" shows as "None", and in the "Client Behaviour" lines, we see "Cloaked / Empty".

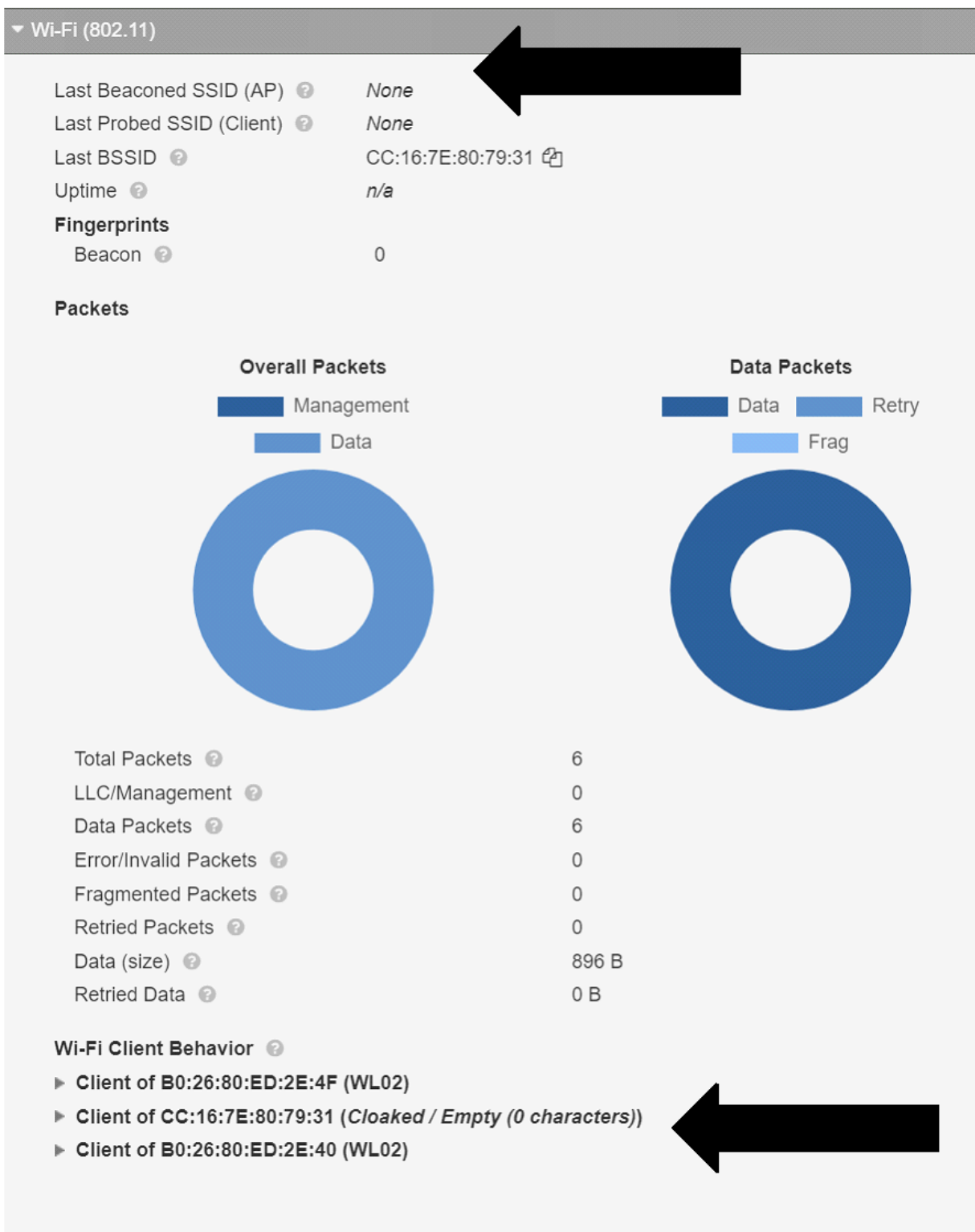


Figure 3.14: Kismet showing a "cloaked" SSID, this network administrator likely thinks this makes that SSID more secure.

Kismet has several other features that we won't dive into here – for instance historically this tool has been used by "war-drivers", who configure Kismet to collect GPS information. This information can then be shared – in the past this was used to share "open" SSIDs that you could use, depending on where in the world you happened to be. Now-a-days this is more useful as a location aid, in other words, without the use of a GPS you can use a free service (such as wgle.net) to use visible and hidden SSID information in your area to determine your location. While Kismet is a great tool for reconnaissance and demonstrations, the menu is fairly easy to get lost in, and it isn't easy to focus on tracking the things we really care about when troubleshooting signal strength. Wavemon is a very different tool. It monitors your connection only, so you have to associate it with an SSID. It'll give you your current access point, speed, channel, and so on, as shown in the following screenshot. In short, all the nuts and bolts information for your current wireless connection is shown on one screen. This can be useful, but it's a narrow view of the information that's normally required for troubleshooting – note in the following screenshot that the values reported are mostly about data throughput and signal as seen from the network that the adapter is associated to. For this reason, the Wavemon tool is mostly useful for troubleshooting uplink issues, and isn't used so much in troubleshooting, assessing, or viewing information on an overall wireless infrastructure:

```

Interface
wlx90f6520f6b4f (IEEE 802.11), phy 0, reg: CA (DFS-FCC), SSID: wlg01
Levels
link quality: 97% (68/70)
=====

signal level: -42 dBm (0.06 uW)
=====

Statistics
RX: 856 (150.07 KiB), drop: 32 (3.7%)
TX: 97 (13.79 KiB), retries: 25 (25.8%), failed: 4
Info
mode: Managed, connected to: B0:26:80:ED:2E:4E, time: 28 sec, inactive: 10.6s
freq: 5540 MHz [survey freq: 5825 MHz], ctr1: 5550 MHz, channel: 108 (width: 40
rx rate: 300.0 Mbit/s MCS 15 40MHz short GI (exp: 57.3 MB/s), tx rate: 300.0 Mb
beacons: 258, avg sig: -42 dBm, interval: 0.1s, DTIM: 1
power mgt: on, tx-power: 18 dBm (63.10 mW)
retry: short long limit 2, rts/cts: off, frag: off
Network
wlx90f6520f6b4f (UP RUNNING BROADCAST MULTICAST)
mac: 90:F6:52:0F:6B:4F, qlen: 1000
ip: 192.168.122.211/24

```

Figure 3.15 – Wavemon display

Of more use for network troubleshooting is **LinSSID**, a fairly close port of InSSIDer, the Windows application from MetaGeek. On running the application, the screen is fairly empty. Select the wireless adapter that you want to use to "sniff" the local wireless networks, and press the **Run** button. The display shows the channels available on both spectrums (2.4 and 5 GHz), with each SSID represented in the top window. Each SSID/BSSID combination that is checked in the list is shown in the bottom window. This makes it very easy to see the signal strength of each AP in the list, along with the relative strengths in the graphical display. SSIDs that interfere with each other are obvious in their overlapping graphical displays. The following screenshot of the display shows the 5GHz spectrum situation – notice how the APs seem to all be clustered around two channels. Any one of them could improve their performance by changing channels, and in our display there's plenty of channels to spare – in fact, this is what's pushing the migration to 5 GHz. Yes, that band is faster, but more importantly it's much easier to solve any

interference problems from neighboring access points. Note also that each channel is shown on the graph as taking roughly 20 GHz (more on this later):

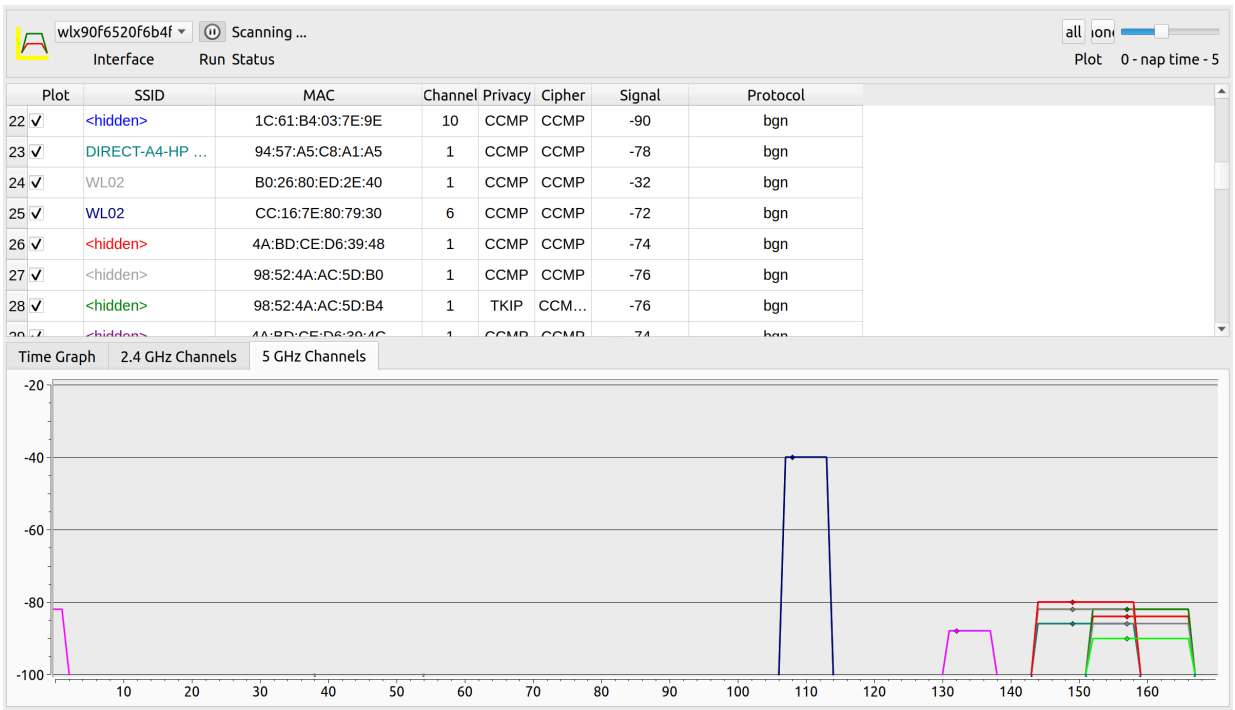


Figure 3.16 – LinSSID Output – The main screen showing channel allocation and strength, both in text and graphically, 5GHz Channels shown

The 2.4Ghz channel is no better. Since there are only 11 channels available in North America, you normally see people select channels 1, 6 or 11 - the 3 channels that do not interfere with each other. In almost any environment that isn't rural, you'll see several neighbors using those same 3 channels that you thought were free! In the following screenshot, we see that everyone picked channel 11 for some reason:

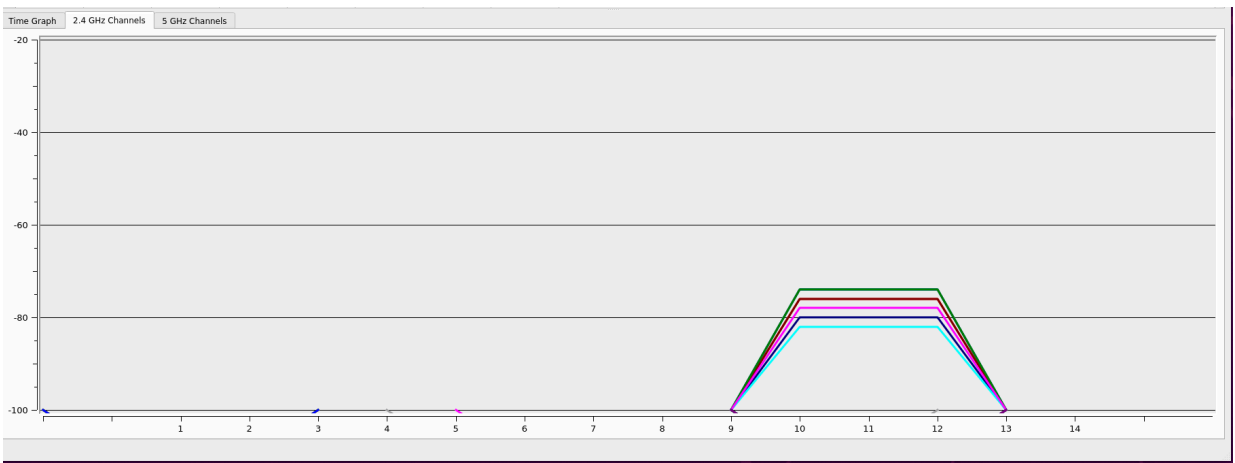


Figure 3.17 – Interference from wireless neighbors - Multiple wireless BSSIDs using the same channel

In this second example (also from the 2.4 GHz spectrum), we see the result of people choosing a wider "footprint" for their signal. In 802.11 wireless, you have the option of expanding your default 20 GHz channel to 40 or 80 GHz. The benefit to this is that – in the absence of any neighbors – this will certainly improve throughput,

especially for a lightly-utilized channel (1 or 2 clients for instance). However, in an environment where adjacent access points have overlapping signals, you can see that increasing channel width (on the 2.4 GHz band) leaves everyone with more interference – neighboring access points can find themselves with no good channel choices. This situation will typically impact the signal quality (and throughput) for everyone, including the one "bad neighbor" that chose to increase their channel width. In the 5 GHz band, there are significantly more channels, so increasing your channel width can usually be done more safely. It's always wise to see what's happening in your spectrum first though, before either selecting or widening the channels on your access points:

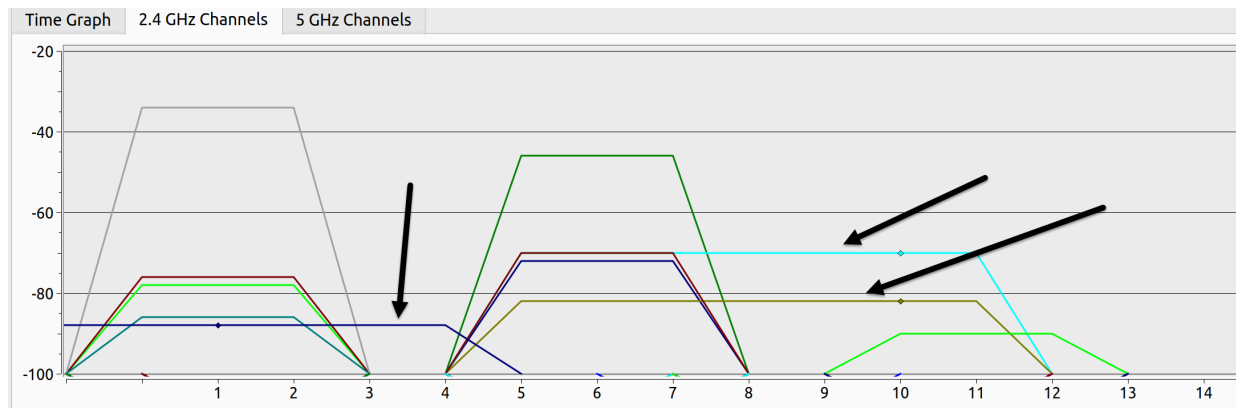


Figure 3.18 – Using wider channel widths in the 2.4GHz spectrum, with resulting interference

Of the tools we've discussed, LinSSID in particular is very useful in doing wireless site surveys where you need to see which channels are available, and more importantly, track signal strength and find "dead spots" to maximize wireless coverage throughout a building or area. LinSSID is also the most helpful of the tools we've discussed in finding situations of channel interference, or troubleshooting situations where a poor choice was made in channel width. With what we've discussed and the tools we've explored, you should now be well equipped to troubleshoot issues around wireless signal strength and interference on both the 2.4 GHz and 5 GHz bands. You should be able to use tools such as Kismet to find hidden SSIDs, tools such as Wavemon to troubleshoot networks you are associated with, and tools including LinSSIDer to view the wireless spectrum holistically, looking for interference and signal strength, as well as issues with channel width and channel overlap.

Summary

With this chapter behind us, you should have a good understanding of the hierarchical organization of the various networking and application protocols, as described in the OSI model. You should have a solid understanding of TCP and UDP, in particular how both protocols use ports, and how TCP sessions are set up and torn down. Using `netstat` or `ss` to see how your host is connecting to various remote services, or what services your host is listening for, is a skill you can use going forward. Expanding on this, using port scanners to see what hosts and network services are running in your organization should be a skill that you should find useful. Finally, our discussion of Linux wireless tools should help in troubleshooting, configuration, and wireless site surveys. All of these skills will be things we build on as we move forward in our journey in this book, but more importantly, they'll be useful in troubleshooting application and networking problems in your organization. This wraps up our discussion of network troubleshooting using Linux. We'll revisit troubleshooting in most chapters though – as we move forward and build each part of our infrastructure, we'll find new potential problems and troubleshooting approaches. In this section, we discussed in detail how communications occur from a network and host point of view. In the next chapter, we'll discuss Linux firewalls, a good way to limit and control those communications.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. When you assess your local ports using `netstat` , `ss` , or another command, will you ever see a UDP session in the `ESTABLISHED` state?
2. Why is it important to be able to determine which processes listen on which ports?
3. Why is it important to determine which remote ports you connect to from any particular application?
4. Why would you scan for expired or soon-to-expire certificates on ports other than `tcp/443` ?
5. Why would netcat need sudo rights in order to start a listener on port `80` ?
6. In the 2.4 GHz band, which three channels make the best selection to reduce interference?
7. When would you use a Wi-Fi channel width other than 20 MHz?

Further reading

- The OSI model (*ISO/IEC 7498-1*):
[https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994\(E\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip)
- Nmap: <https://nmap.org/>
- The Nmap reference guide:
 - <https://nmap.org/book/man.html>
 - <https://www.amazon.com/Nmap-Network-Scanning-Official-Discovery/dp/0979958717>
- MASSCAN: <https://github.com/robertdavidgraham/masscan>



Your gateway to knowledge and culture. Accessible for everyone.



z-library.sk

z-lib.gs

z-lib.fm

go-to-library.sk



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>