The Best of Hack and /

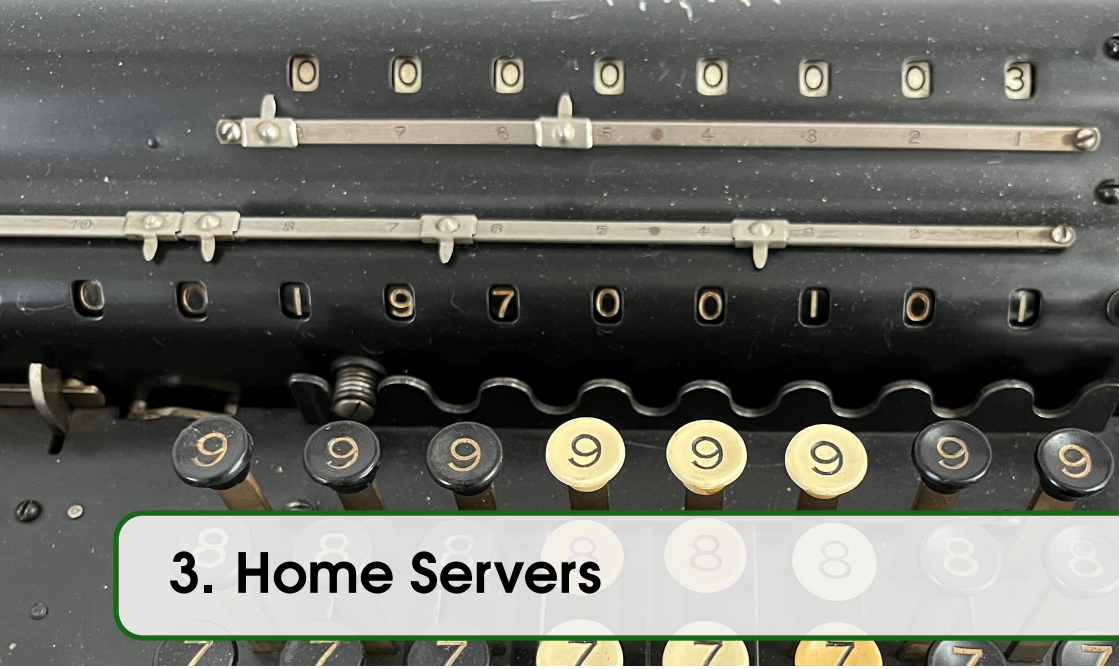
Linux Admin Crash Course

Kyle Rankin

PUBLISHED BY KYLE RANKIN

HTTPS://KYLERANK.IN/WRITING.HTML

First edition, April 2023



3. Home Servers

These days, it seems everyone is talking about the cloud. Now, what exactly someone means by "the cloud" seems to vary, but typically, the cloud refers to some sort of service, such as e-mail, Web, DNS, file storage and so on, that is managed for you by a third party. Many people love how easy it can be to outsource their e-mail service, blog or image site to someone else. Like oil changes, home repair and cooking, server administration is yet another task you can pay (either with money or with marketing data) someone else to manage for you.

Alongside this trend to outsource work is a growing movement that values doing things yourself. Some examples include "Makers" involved in designing their own electronics, do-it-yourself home improvement, gardening, amateur cheese making, baking and even home brewing. The fact is, many of these so-called chores actually are rather rewarding and even fun to do yourself. I think we as Linux users should apply this same idea to server management. It turns out it is quite rewarding, educational and not terribly difficult to manage your own services at home instead of outsourcing them to the cloud. This is on top of the fact that when you manage your own server, you are in full control of your server, what's installed on it and who can see it.

In addition to the other reasons why it's valuable to manage some of your own services yourself, having a vibrant home network is the perfect laboratory for sysadmin projects you don't have time or resources to work on during your day job. I developed many of my sysadmin skills first on my home network, especially when I was starting out. It's a great way for junior sysadmin to make up for any gaps in their experience from their day job and it was this experience that got me one of my first full-time sysadmin jobs. During the interview I was asked how to set up a number of different

services and it had just so happened I had recently set up the same servers at home for myself. Even though I didn't have much professional experience, my personal experience gave me the confidence (and know-how) to answer the interview questions well and I got the job.

In this chapter, I'm going to discuss how to set up various types of services at home and how to make them available to the Internet at large. In this first section, I discuss some things you should consider about your network before you set up your first server at home. Then I will describe how to set up a DNS server, a personal blog, a local mail server, and a redundant web server hosted on a Raspberry Pi cluster. Finally I will describe how I used my own home servers as an emergency backup when my primary colocated server had a major outage.

3.1 Setting Up Your Network

When it comes to hosting servers at home, all ISPs (Internet Service Providers) are not created equal. Before I even discuss bandwidth, first you should look into your ISP's terms of service. It turns out that some ISPs discourage, disallow or sometimes outright block home users from hosting their own services on the Internet. Take a large dose of caffeine and try to read through your ISP's terms of service (or just call and ask them) to see whether they have any sort of restrictions. At the very least it's common for even server-friendly ISPs to block outbound e-mail traffic (SMTP port 25) by default to prevent spam. Although I'll discuss this more in a future column about e-mail, some ISPs will lift this restriction and some won't. The bottom line is that if hosting your own server is important to you, you will want to make sure you use an ISP that allows it. For me, this policy is more important when choosing an ISP than even speed or price.

Static IPs vs. Dynamic IPs

No matter what type of Internet connection you have, ultimately you are assigned at least one publicly routed IP address. If this address changes each time you connect to the Internet (or each time your DSL or cable modem resets), you have a dynamic IP. If this IP stays the same, it's static. Although people historically have run servers on both static and dynamic IPs, with a dynamic IP, you will have to go through the additional trouble of setting up some sort of dynamic DNS service so that each time your IP changes at home, everyone trying to access your service on the Internet will get the new IP. Unfortunately, due to the nature of how DNS works, you can't always guarantee (even with low TTLs) that everyone will see your changed IP in a timely manner, so if you are serious about running servers at home, I recommend you spring for one or more static IPs.

Connection Speed

Typically when you rate the quality of your Internet connection at home, you first look at your download speed. Average home users rely much more on their download bandwidth than their upload bandwidth as a metric of how "fast" their connection is, and many home Internet connections have much higher download bandwidth than upload. Once you start hosting servers at home, however, you'll find that their performance is governed more by your upload bandwidth. If you want to host bandwidth-hungry services at home, like streaming audio or video or image-heavy Web sites, you might want to upgrade or change your Internet connection to get more upload bandwidth. On the other hand, your personal DNS or e-mail server probably is going to be fine even with somewhat low upload bandwidth. Although upload bandwidth can be slower at home than at a data center, most connections at home (at least in the US) are unmetered so you don't have to worry about bandwidth caps.

Modems and Gateways

Most people who would want to host a server at their homes tend to access the Internet through some sort of DSL or cable modem. This device connects to either a phone line or some other cable on one end and provides a network port (or sometimes a USB port) on the other. More sophisticated modems actually can act as a gateway, and even a DHCP server, and hand out internal IPs to computers in the home while the public IP resides on the modem itself.

If you plan on having multiple computers inside your home network, I recommend getting the modem configured so it acts more like a bridge, so that the publicly routed IP address is assigned to a device that is under your control, whether it's a home router or a computer on your network. Most home routers (including DSL and cable modems, if your ISP gives you the ability to configure them) have the ability to do port forwarding so that incoming traffic intended for your Web server (ports 80 and 443) can be redirected to the internal IP address. The more control you have over your gateway, the more flexibility you will have in how you set up your servers and your network. If you do opt to use a consumer router instead of turning a home computer into the gateway, you might want to choose a router that can be reflashed with custom Linux firmware (like OpenWRT or DD-WRT), so you can have some of the same flexibility you would have if a Linux server acted as the gateway.

Security, Firewalls and Virtual IPs

Of course, any time you open up a service to the Internet, you are opening yourself up to attack. It doesn't matter if you just have a lone server on the Internet; attacks are automated, so your obscurity doesn't ensure security. Be sure that any service and server you make available on the Internet is kept up to date with the latest security patches. If you have the ability to configure a firewall on your gateway router, block all incoming ports by default and allow in only ports you know need to be open. If

you are going to open up an SSH server to the public Internet, be sure to audit your passwords, and make sure they are difficult to guess (or better, disable passwords altogether and use key-based authentication). Today, more home (and enterprise) Linux servers are hacked due to bad passwords than just about anything else.

While I'm on the subject of firewalls, here's a quick tip if you happen to use a Linux device as your router with iptables. Even if you are granted multiple public IPs, you may find you prefer to have all Internet traffic come through a central router so it's easier to monitor and secure. To accomplish that, you likely will need to have your gateway device configured to answer on all of the public IPs and assign private IPs to the computers inside your home. Let's assume I have a few static IPs, including 66.123.123.63 and 66.123.123.64, and a gateway router that is configured to answer to both of those IPs on eth0. I have an internal server on my network with an IP address of 192.168.0.7. Because it has an internal IP, I want to forward traffic on my gateway destined for 66.123.123.64 to 192.168.0.7. The first way I could do it is to forward traffic only on specific ports to this host. For instance, if this were a Web server, I might want to forward only ports 80 and 443 to this server. I could use these iptables commands on my gateway router for the port forwarding:

```
iptables -t nat -A PREROUTING -d 66.123.123.64 -i eth0 -p
 ↪ tcp -m tcp -dport 80 -j DNAT -to-destination 192.168.0.7:80
iptables -t nat -A PREROUTING -d 66.123.123.64 -i eth0 -p
 ↪ tcp -m tcp -dport 443 -j DNAT -to-destination 192.168.0.7:443
```

This is also a common solution if you have only one public IP but multiple servers in your network, so you can forward Web ports to an internal Web server and e-mail ports to a different e-mail server. This method works; however, I'll have to be sure to add new firewall rules each time I want to forward another port. If I simply want to have the router forward all traffic destined for 66.123.123.64 to 192.168.0.7, I could use these two commands:

```
iptables -t nat -A PREROUTING -d 66.123.123.64 -i eth0 -j
 ↪ DNAT -to-destination 192.168.0.7
iptables -t nat -A POSTROUTING -s 192.168.0.7 -o eth0 -j
 ↪ SNAT -to-source 66.123.123.64
```

Note that because these commands forward all traffic to that internal host, regardless of port, I will want to make sure to lock down the firewall rules on that internal server.

This should be enough information to get you started on your network setup at home so that, you'll be ready to set up your first service. In the next section, I'll focus on DNS, including how to register a domain and how to set up your own home DNS server.

3.2 Setting Up A Home DNS Server

I honestly think most people simply are unaware of how much personal data they leak on a daily basis as they use their computers. Even if they have some inkling along those lines, I still imagine many think of the data they leak only in terms of individual

facts, such as their name or where they ate lunch. What many people don't realize is how revealing all of those individual, innocent facts are when they are combined, filtered and analyzed.

Cell-phone metadata (who you called, who called you, the length of the call and what time the call happened) falls under this category, as do all of the search queries you enter on the Internet.

For this section, I discuss a common but often overlooked source of data that is far too revealing: your DNS data. You see, although you may give an awful lot of personal marketing data to Google with every search query you type, that still doesn't capture all of the sites you visit outside Google searches either directly, via RSS readers or via links your friends send you. That's why the implementation of Google's free DNS service on 8.8.8.8 and 8.8.4.4 is so genius–search queries are revealing, but when you capture all of someone's DNS traffic, you get the complete picture of every site they visit on the Internet and beyond that, even every non-Web service (e-mail, FTP, P2P traffic and VoIP), provided that the service uses hostnames instead of IP addresses.

Let me back up a bit. DNS is one of the core services that runs on the Internet, and its job is to convert a hostname, like www.linuxjournal.com, into an IP address, such as 76.74.252.198. Without DNS, the Internet as we know it today would cease to function, because basically every site we visit in a Web browser, and indeed, just about every service we use on the Internet, we get to via its hostname and not its IP. That said, the only way we actually can reach a host on the Internet is via its IP address, so when you decide to visit a site, its hostname is converted into an IP address to which your browser then opens up a connection. Note that via DNS caching and TTL (Time To Live) settings, you may not have to send out a DNS query every time you visit a site. All the same, lately TTLs are short enough (often ranging between one minute to an hour or two–www.linuxjournal.com's TTL is 30 minutes) that if I captured all your DNS traffic for a day, I'd be able to tell you every Web site you visited along with the first time that day you visited it. If the TTL is short enough, I probably could tell you every time you went there.

Most people tend to use whatever DNS servers they have been provided. On a corporate network, you are likely to get a set of DNS servers over DHCP when you connect to the network. This is important because many corporate networks have internal resources and internal hostnames that you would be able to resolve only if you talked to an internal name server.

Although many people assume very little privacy at work, home is a different matter. At home, you are most likely to use the DNS servers your ISP provided you, while others use Google's DNS servers because the IPs are easy to remember. This means even if others can't intercept your traffic (maybe you are sending it through a VPN, or maybe that kind of line tapping simply requires more legal standing), if they can get access to your DNS logs (I could see some arguing that this qualifies as metadata), they would have a fairly complete view of all the sites you visit without

your ever knowing.

This is not just valuable data from a surveillance standpoint, or a privacy standpoint, but also from a marketing standpoint. Even if you may be fine with the government knowing what porn sites you browse, where you shop, where you get your news and what e-mail provider you use, you may not want a marketing firm to have that data.

Recursive DNS vs. DNS Caching

The key to owning your DNS data and keeping it private is to run your own DNS server and use it for all of your outbound DNS queries. Although many people already run some sort of DNS caching programs, such as dnsmasq to speed up DNS queries, what you want isn't simply a DNS cache, but something that can function as a recursive DNS resolver. In the case of dnsmasq, it is configured to use upstream recursive DNS servers to do all of the DNS heavy lifting (the documentation recommends you use whatever DNS servers you currently have in /etc/resolv.conf). Thus, all of your DNS queries for www.linuxjournal.com go to your DNS caching software and then are directed to, for instance, your ISP's DNS servers before they do the traditional recursive DNS procedure of starting at root name servers, then going to com, then finally to the name servers for linuxjournal.com. So, all of your queries still get logged at the external recursive DNS server.

What you want is a local DNS service that can do the complete recursive DNS query for you. In the case of a request for www.linuxjournal.com, it would communicate with the root, com and linuxjournal.com name servers directly without an intermediary and ultimately cache the results like any other DNS caching server. For outside parties to capture all of your DNS logs, they either would have to compromise your local, personal DNS server on your home network, set up a tap to collect all of your Internet traffic or set up a tap at all the root name servers. All three of these options are either illegal or require substantial court oversight.

Install and Configure A Recursive DNS Server

So, even when you rule out pure DNS caching software, there still are a number of different DNS servers you can choose from, including BIND, djbdns and unbound, among others. I personally have the most experience with BIND, so that's what I prefer, but any of those would do the job. The nice thing about BIND, particularly in the case of the Debian and Ubuntu packages, is that all you need to do is run:

```
sudo apt install bind9
```

and after the software installs, BIND automatically is configured to act as a local recursive DNS server for your internal network. The procedure also would be the same if you were to set this up on a spare Raspberry Pi running the Raspbian distribution. On other Linux distributions, the package may just be called bind.

If BIND isn't automatically configured as a local recursive DNS server on your particular Linux distribution and doesn't appear to work out of the box, just locate the options section of your BIND config (often in /etc/bind/named.conf, /etc/bind/-named.conf.options or /etc/named/named.conf, depending on the distribution), and if you can't seem to perform recursive queries, add the following line under the options section:

```
options {
  allow-recursion { 10/8; 172.16/12; 192.168/16; 127.0.0.1; };
  . . .
}
```

This change allows any hosts on those networks (internal RFC1918 IP addresses) to perform recursive queries on your name server without allowing the world to do so.

Once you have BIND installed, you'll want to test it. If you installed BIND on your local machine, you could test this out with the dig command:

```
dig @localhost www.linuxjournal.com
; <<>> DiG 9.8.1-P1 <<>> @localhost www.linuxjournal.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 17485
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 0

;; QUESTION SECTION:
;www.linuxjournal.com.          IN   A

;; ANSWER SECTION:
www.linuxjournal.com.   1800   IN   A   76.74.252.198

;; AUTHORITY SECTION:
linuxjournal.com.       30479  IN   NS  ns66.domaincontrol.com.
linuxjournal.com.       30479  IN   NS  ns65.domaincontrol.com.

;; Query time: 31 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Wed Dec 18 09:37:13 2013
;; MSG SIZE  rcvd: 106
```

Otherwise, replace localhost with the IP address of your Raspberry Pi or whatever machine on which you installed BIND. To use this name server for all of your requests, update your /etc/resolv.conf file so that it contains:

```
nameserver 127.0.0.1
```

as its only nameserver line. Replace 127.0.0.1 with the IP address of the machine you installed BIND on if it isn't on the same machine. On some modern distributions, there are external tools that tweak /etc/resolv.conf for you, so in those cases, you may have to edit your dhclient.conf or other network configuration files so that you can override the provided list of name servers. Once you do that though, really that's all there is to it. Now you can use DNS knowing that all of your DNS search data sits on a machine under your control.

Install and Configure an Authoritative DNS Server

Recursive DNS servers allow you to resolve other people's domains but what about your own domains? Every server (including your own) that has a presence on the Internet should have a public IP address. In an earlier section I discussed how to set up your home network for a server and to do this you need at least one public IP (hopefully static) you can use. It's true that all you really need to host many services on the Internet is an IP address; however, in practice, there are only so many IP addresses (like phone numbers) that the average person is going to commit to memory. As IPv6 becomes commonplace, this will be even more true. DNS allows you to register a domain name and associate individual host names (like www.example.com and mail.example.com) to IP addresses.

For instance, how many of you (besides you, Katherine) have memorized the IP address for www.linuxjournal.com? If you did want to know the IP address, all you would need to do is perform a simple nslookup command:

```
nslookup www.linuxjournal.com
Server:         192.168.0.1
Address:        192.168.0.1#53

Non-authoritative answer:
Name:   www.linuxjournal.com
Address: 76.74.252.198
```

In this example, the first bit of output tells me that I'm getting this answer from a DNS server at 192.168.0.1 (my own personal DNS server) and that the IP address for www.linuxjournal.com is currently 76.75.252.198. There isn't enough space in this section to describe everything that happened to allow me to get that IP address, but essentially, my DNS server asked other DNS servers on the Internet for this IP address and was subsequently redirected to more and more DNS servers until it finally found the one that knew the answer. If you are interested in more detail on how this works, books like DNS and BIND do a good job of explaining it, or from the command line, you could run `dig www.linuxjournal.com +trace`.

Like with the recursive name server example I'm going to use BIND for my DNS server software. When it comes to authoritative name servers, though, you can just rely on the out-of-the-box configuration. Unfortunately, there are slight differences in how each distribution packages BIND. For instance, under Red Hat, you install the bind package, but under Debian-based systems (like Ubuntu), you install bind9. Red Hat stores its core BIND configuration file at /etc/named.conf and all its zone files (files that contain name→IP address mappings for a domain, such as example.org, a subdomain, such as ny.example.org, or possibly both) under /var/named, while Debian-based systems put named.conf and any zone files under the /etc/bind/ directory. Once you get past the differences, however, the syntax inside the files should be similar. Just to simplify things, I'm going to base the rest of this article off a standard Debian server, so we have some sort of baseline. If you use a different distribution, however, it shouldn't be too difficult to adapt these instructions

to the different file paths.

Primary DNS Configuration

> **Note:** BIND still uses the outdated terms "master" and "slave" in its configuration files to describe its primary and secondary DNS servers. I will use the terms "primary" and "secondary" instead in this section, except for in the actual configuration files where I will fall back to BIND's terminology.

A DNS primary contains its own zone files that have name→IP address mappings, and it doesn't have to consult any other source to answer queries for those names. By contrast, a DNS secondary is configured to load all of its zone configurations from a DNS primary. Any future changes are made on the primary and propagate to each of the secondary. Any individual BIND instance acts as a DNS primary, a DNS secondary or a caching name server, or all three at the same time (although it can be a primary or a secondary only to any individual zone, not both).

For this example, let's set up a DNS primary for example.org, and this primary will have the following records:

- ns1.example.org, which points to 123.12.34.56 (the public IP of the name server itself).
- example.org, which points to 123.12.34.57.
- www.example.org also points to 123.12.34.57.

To start, I create the zone file at /etc/bind/db.example.org (remember Red Hat stores these zones in a different places) and put the following information in it:

```
;
; BIND data file for example.org
;
$TTL 4h
@  IN  SOA ns1.example.org. root.example.org. (
  2             ; Serial
  604800        ; Refresh
  86400         ; Retry
  2419200       ; Expire
604800 )        ; Negative Cache TTL
;
@       IN NS   ns1.example.org.
@       IN A    123.12.34.57
www     IN A    123.12.34.57
ns1     IN A    123.12.34.56
```

Make sure this file has similar permissions to the other zone files you find in the /etc/bind directory. The first non-comment line in the file sets the TTL or Time To Live, the default time in which a remote DNS server will cache any answers it gets from your DNS server before it will ask it again. The value you put here will help determine how fast changes you make will propagate. BIND accepts seconds in this field, or you can use shorthand values like 1d for one day, 4h for four hours or 20m for 20 minutes. I set the TTL to four hours here; however, if you make frequent changes to your records (or know you are going to soon), you may want to make the TTL

shorter. On the other hand, if you find you hardly ever change these values, you might want to bump up the TTL to a day to reduce load on your DNS server.

Something to note is that zone files use semicolons not hashes at the beginning of a line for comments. A common mistake is to put hashes in a zone file to make a comment, reload BIND and then wonder why your changes didn't take. When BIND sees a mistake like that, it just skips that particular zone.

To keep things simple, I'm going to skip the Retry, Refresh and other values here–just keep them with these defaults unless you know what you are doing. The Serial line is for DNS secondaries, which I discuss later. Below those values, however, you'll see the syntax I used to define the different records:

```
@       IN NS   ns1.example.org.
@       IN A    123.12.34.57
www     IN A    123.12.34.57
ns1     IN A    123.12.34.56
```

The first record starts with @, which means it is a record for example.org itself. In this case, it is an NS record that defines the hostname I'm going to use for my name server. You can use any hostname you control here (including hostnames on a different domain, actually), but one popular convention is to use hostnames like ns1 and ns2 for the first and second name servers. The second record begins with an @ as well, only in this case, it's an A record. An A record is a fundamental DNS record that maps a hostname (like www) to an IP address (like 123.12.34.57). In this case, because the record starts with @, I am setting the IP address for example.org itself. The next two lines define two more A records, one for www.example.org and one for ns1.example.org. It's important if you used a name within this same domain for your name server (like ns1.example.org) that you be sure to add an A record so that it has an IP address.

Now that I have created my zone, next I need to modify the /etc/bind/named.conf file and add a new section at the end of the file to point to the /etc/bind/db.example.org file I just created:

```
zone "example.org" {
  type master;
  file "/etc/bind/db.example.org";
};
```

After the file is changed, I reload BIND, and I should be able to send DNS requests to my new DNS server:

```
sudo /etc/init.d/bind9 reload
* Reloading domain name server... bind [OK]

nslookup www.example.org localhost
Server:         localhost
Address:        127.0.0.1#53

Name:   www.example.org
Address: 123.12.34.56
```

If there is a problem with the BIND reload, it should tell you on the command line. Otherwise, if it still doesn't work, you may have to look in your syslog file (/var/log/syslog on Debian-based systems and /var/log/messages on Red Hat) for clues.

Secondary DNS Configuration

Many registrars on the Internet require that any domain you register have at least two DNS servers configured with it. It's a good practice to have, because if you have a single DNS server and it goes down, it effectively will make all your servers under that domain inaccessible. This means you need to set up a second DNS server on a different IP, ideally on a different network, or have a friend with a DNS server act as a secondary to your primary DNS server. In either case, it's a relatively simple process. Let's say that my second DNS server is going to be at the IP address 98.76.54.32. First, I would log in to my Master DNS server and add the new NS and A records to my zone file:

```
;
; BIND data file for example.org
;
$TTL 4h
@  IN  SOA ns1.example.org. root.example.org. (
  2             ; Serial
  604800        ; Refresh
  86400         ; Retry
  2419200       ; Expire
604800 )        ; Negative Cache TTL
;
@       IN NS   ns1.example.org.
@       IN NS   ns2.example.org.
@       IN A    123.12.34.57
www     IN A    123.12.34.57
ns1     IN A    123.12.34.56
ns2     IN A    98.76.54.32
```

Next, I edit named.conf and add a line to the configuration of example.org so that it will allow zone transfers from my DNS secondary:

```
zone "example.org" {
  type master;
  file "/etc/bind/db.example.org";
  allow-transfer { 98.76.54.32; };
};
```

Finally, I would install BIND on the second server, or if it already exists, all I would have to do is add a new entry at the end of the named.conf file to define the example.org zone and tell this server the IP address of the primary:

```
zone "example.org" {
  type slave;
  file "/var/cache/bind/db.example.org";
  masters { 123.12.34.56; };
};
```

Note that in this case the secondary zone is being stored under /var/cache/bind. That's the default location for secondary zone files under Debian-based systems. Under Red Hat, you would store them under /var/named/. Once I reload BIND on the secondary server, it will pull the new zone information from the primary, and I should be able to perform DNS queries against it.

Once you have set up a secondary, keep in mind that any time you make a change to the primary, you will need to increment the Serial field in the Master's zone file (in my example, it is set to 2, but a lot of administrators like to set it to the current date plus two extra number fields, such as 2010120500). When the secondary needs to know whether its zone information is up to date, it compares its serial number with the serial number on the primary. If the primary's serial number for a zone is higher, it copies down the new zone information; otherwise, it sticks with what it has cached.

Domain Registration

Once you have a functioning DNS server, all that's left is to tell the world to use it. If you haven't already registered your domain with a registrar, find a domain registration service on the Internet (there are too many for me to list here, but a search for domain name registration should turn up plenty). When you register the domain, most registrars will let you use their own DNS servers for your domain, but you don't need them! When you get to the point in the registration process where it asks you about your DNS servers, just give them the public IP address for your own DNS server (in my case, it would be ns1.example.org or 123.12.34.56). Note that many registrars require you to have two DNS servers defined for a domain, so in that case, set up a secondary DNS server and add its IP address as well. Once you complete the registration process and allow the new domain information time to propagate around the Internet, you will have the ability make IP changes for your Web, mail and other servers all from your own machines.

3.3 A Local Mail Server

I've written about the mutt email client a lot over the years. For me, in this day and age of large graphical mail programs and Web-based mail applications, you still can't beat the speed, power and customization of mutt. Let's also not forget the vi-style keybindings–I love those.

One thing that was true for the longest time about mutt, was that it is strictly a MUA (Mail User Agent) and not an MTA (Mail Transfer Agent). This means mutt was concerned only with acting as an e-mail client and didn't actually contain any code to communicate with remote mail servers. That job is done by an MTA. Although many mail clients also include code so they can relay mail through an MTA (including mutt), originally mutt opted to use the system's own local mail server. Traditionally, this hasn't been an issue on Linux, as most Linux servers have had some mail server installed and set up. Now, however, you might not have a fully

configured mail server on your desktop install. That's okay though, because in this section, you'll see how simple it is to set up your own local mail server, thanks to Postfix.

Even if you don't use mutt, there are many advantages to having your own local mail server, if only to relay mail for you. For one, it can handle spooling all of your e-mail and will retry delivery automatically if it fails for some reason or another (such as if your wireless connection drops or you close your laptop) without having to leave your mail program open. For another, once you have your mail server set up how you want it, any other mail client on your computer can take advantage of it: simply point your client to localhost.

The Mail Server Holy War

A number of different mail servers are available for Linux, each with its own set of advantages and disadvantages. Many holy wars have been fought over Sendmail vs. Postfix vs. Exim vs. using Telnet to connect directly to port 25 on a mail server and type in raw SMTP commands. I've tried them all over the years (yes, even Telnet), and for me, Postfix has the best balance between stable performance, security and most important, simple configuration files. So for this column, I discuss the specific steps for setting up Postfix as a mail relay.

The first step is to install the Postfix server itself. On most distributions, you'll find this package is split up into a main Postfix package plus a few extra packages that provide specific features, such as MySQL or LDAP integration. Because we are just setting up a basic mail relay here, all we really need is the main Postfix package. Now, if you install this package on a Debian-based system, you will be prompted by the post-install script that acts as a wizard to set up Postfix for you. If you want, you simply can walk through the wizard and pick "Internet Site" to send e-mail out directly to the rest of the Internet or choose "Internet with smarthost" to relay all of your mail through a second mail server (perhaps provided by your ISP) first. Either way, you will be asked a few simple questions, and at the end, you'll have a basic Postfix configuration ready to use.

On other systems (or if you choose "No configuration" on a Debian-based system), you might end up with an empty or very heavily commented Postfix configuration file at /etc/postfix/main.cf. What you'll find is that for a basic mail server, you really need only a few lines in your config. Postfix picks pretty sane and secure defaults, so if you want it to deliver mail on your behalf, you need only a few lines:

```
mynetworks = 127.0.0.0/8
inet_interfaces = loopback-only
```

Yes, that's basically it. Now, restart Postfix with /etc/init.d/postfix restart, and your mail server will be up and running. With the sane defaults in Postfix, you just need to hard-code those two settings to ensure that Postfix accepts mail only on localhost. The `inet_interfaces` line tells Postfix to listen only on the localhost address for

e-mail so no clients can connect to your server from the outside. The `mynetworks` line adds to that security and tells Postfix to allow only mail from localhost to be relayed through the server.

The Pesky Port 25 Problem

It used to be that the above was all you needed for a functioning mail server on the Internet. With the rise of spam measures and countermeasures, however, fewer and fewer ISPs are willing to allow port 25 traffic from clients through to the outside world. Even if they do, many mail servers on the Net won't accept traffic from hosts inside ISP networks. If you find yourself on such a network, you likely will need to add a relay host to your main.cf. The relay host is a mail server usually provided by your ISP through which your mail server can send e-mail. If you were setting up a client like Thunderbird, for instance, this would be the SMTP server you would configure for it.

To set up a generic relay host in Postfix, just add:

```
relayhost = mail.somedomain.net
```

to your /etc/postfix/main.cf. Replace mail.somedomain.net with the hostname of your ISP's relay host. Once you modify the file, type `sudo postfix reload` to enable the new settings.

SMTP AUTH

Of course, some mail servers won't just let anyone on their network relay through them (and rightly so). In that case, usually they require that everyone authenticate with them first. This takes a few extra steps with Postfix, but like with everything else, it's still not very difficult. First, add the following lines to the /etc/postfix/main.cf:

```
smtp_sasl_auth_enable = yes
smtp_sasl_password_maps = hash:/etc/postfix/sasl_passwd
smtp_sasl_security_options = noanonymous
```

This tells postfix to enable SMTP authentication and tells it to look in /etc/postfix/sasl_passwd for logins and passwords to use for hosts. The next step is to create the /etc/postfix/sasl_passwd file. If I wanted to log in to mail.somedomain.net with the user name kyle and the password muttrules, I would put the following line in the file:

```
mail.somedomain.net kyle:muttrules
```

There is a downside to this in that the password for the account is now in clear text. That's less than ideal, but you can at least make sure that only root can read the file. As the root user, type:

```
chown root:root /etc/postfix/sasl_password
chmod 600 /etc/postfix/sasl_passwd
```

Postfix actually doesn't read this file directly; instead, it reads a hash database created from this file. To create the file, run:

```
postmap /etc/postfix/sasl_passwd
```

And, you will see that a new file, /etc/postfix/sasl_passwd.db, has been created. You'll need to run the postmap command any time you modify the /etc/postfix/sasl_passwd file. Now, reload Postfix one final time, and mutt should be able to relay mail through your local host. If you want to perform a quick test without mutt, you can type:

```
echo test | mail -s "test" user@remotehost
```

and it will send an e-mail message with a subject and body of "test" to the user you specify.

Postfix's logfile might vary a bit, depending on your system, but you should be able to find it in /var/log/mail.log or /var/log/maillog. That's the first place you should look if you find that some mail is not being delivered. The second place to look is the mailq command. That command will give you a quick status of all e-mail that is currently in the local spool along with its status. If all of your mail has been delivered successfully to other hosts, the output will look something like this:

```
mailq
Mail queue is empty
```

It's truly that simple. Of course, mail server administration definitely can become more complex than this when you want to do more than relay your own personal e-mail. But, it's good to know that simple configurations like the above are possible. If you are like me, saving time on the Postfix configuration just gives you extra time to tweak your mutt config.

3.4 Clustering with Raspberry Pis

Although many people are excited about the hardware-hacking possibilities with the Raspberry Pi, one of the things that interests me most is the fact that it is essentially a small low-power Linux server I can use to replace other Linux servers I already have around the house. In past Linux Journal columns, I've talked about using the Raspberry Pi to replace the server that controls my beer fridge[1] and colocating a Raspberry Pi in Austria[2]. After I colocated a Raspberry Pi in Austria, I started thinking about the advantages and disadvantages of using something with so many single points of failure as a server I relied on, so I started thinking about ways to handle that single point of failure.

So, in this section, I'm building the foundation for setting up redundant services with a pair of Raspberry Pis. I start with setting up a basic clustered network filesystem

[1] https://www.linuxjournal.com/content/temper-pi

[2] https://www.linuxjournal.com/content/raspberry-strudel-my-raspberry-pi-austria

using GlusterFS. In later articles, I'll follow up with how to take advantage of this shared storage to set up other redundant services. Of course, although I'm using a Raspberry Pi for this article, these same steps should work with other hardware as well.

Configure the Raspberry Pis

To begin, I got two SD cards and loaded them with the latest version of the default Raspberry Pi distribution from the official Raspberry Pi downloads page, the Debian-based Raspbian. I followed the documentation to set up the image and then booted in to both Raspberry Pis while they were connected to a TV to make sure that the OS booted and that SSH was set to start by default (it should be). You probably also will want to use the raspi-config tool to expand the root partition to fill the SD card, since you will want all that extra space for your redundant storage. After I confirmed I could access the Raspberry Pis remotely, I moved them away from the TV and over to a switch and rebooted them without a display connected.

By default, Raspbian will get its network information via DHCP; however, if you want to set up redundant services, you will want your Raspberry Pis to keep the same IP every time they boot. In my case, I updated my DHCP server so that it handed out the same IP to my Raspberry Pis every time they booted, but you also could edit the /etc/network/interfaces file on your Raspberry Pi and change:

```
iface eth0 inet dhcp
```

to:

```
auto eth0
iface eth0 inet static
  address 192.168.0.121
  netmask 255.255.255.0
  gateway 192.168.0.1
```

Of course, modify the networking information to match your personal network, and make sure that each Raspberry Pi uses a different IP. I also changed the hostnames of each Raspberry Pi, so I could tell them apart when I logged in. To do this, just edit /etc/hostname as root and change the hostname to what you want. Then, reboot to make sure that each Raspberry Pi comes up with the proper network settings and hostname.

Configure the GlusterFS Server

GlusterFS is a userspace clustered filesystem that I chose for this project because of how simple it makes configuring shared network filesystems. To start, choose a Raspberry Pi that will act as your primary. What little initial setup you need to do will be done from the primary node, even though once things are set up, nodes should fail over automatically. Here is the information about my environment:

```
Primary hostname: pi1
```

```
Primary IP: 192.168.0.121
Primary brick path: /srv/gv0
Secondary hostname: pi2
Secondary IP: 192.168.0.122
Secondary brick path: /srv/gv0
```

Before you do anything else, log in to each Raspberry Pi, and install the glusterfs-server package:

```
sudo apt install glusterfs-server
```

GlusterFS stores its files in what it calls bricks. A brick is a directory path on the server that you set aside for gluster to use. GlusterFS then combines bricks to create volumes that are accessible to clients. GlusterFS potentially can stripe data for a volume across bricks, so although a brick may look like a standard directory full of files, once you start using it with GlusterFS, you will want to modify it only via clients, not directly on the filesystem itself. In the case of the Raspberry Pi, I decided just to create a new directory called /srv/gv0 for my first brick on both Raspberry Pis:

```
sudo mkdir /srv/gv0
```

In this case, I will be sharing my standard SD card root filesystem, but in your case, you may want more storage. In that situation, connect a USB hard drive to each Raspberry Pi, make sure the disks are formatted, and then mount them under /srv/gv0. Just make sure that you update /etc/fstab so that it mounts your external drive at boot time. It's not required that the bricks are on the same directory path or have the same name, but the consistency doesn't hurt.

After the brick directory is available on each Raspberry Pi and the glusterfs-server package has been installed, make sure both Raspberry Pis are powered on. Then, log in to whatever node you consider the primary, and use the gluster peer probe command to tell the primary to trust the IP or hostname that you pass it as a member of the cluster. In this case, I will use the IP of my secondary node, but if you are fancy and have DNS set up you also could use its hostname instead:

```
pi@pi1 ~ $ sudo gluster peer probe 192.168.0.122
Probe successful
```

Now that my pi1 server (192.168.0.121) trusts pi2 (192.168.0.122), I can create my first volume, which I will call gv0. To do this, I run the gluster volume create command from the primary node:

```
pi@pi1 ~ $ sudo gluster volume create gv0 replica 2
 ↪ 192.168.0.121:/srv/gv0 192.168.0.122:/srv/gv0
Creation of volume gv0 has been successful. Please start
the volume to access data.
```

Let's break this command down a bit. The first part, gluster volume create, tells the gluster command I'm going to create a new volume. The next argument, gv0 is the name I want to assign the volume. That name is what clients will use to refer to the volume later on. After that, the replica 2 argument configures this volume to use

replication instead of striping data between bricks. In this case, it will make sure any data is replicated across two bricks. Finally, I define the two individual bricks I want to use for this volume: the /srv/gv0 directory on 192.168.0.121 and the /srv/gv0 directory on 192.168.0.122.

Now that the volume has been created, I just need to start it:

```
pi@pi1 ~ $ sudo gluster volume start gv0
Starting volume gv0 has been successful
```

Once the volume has been started, I can use the volume info command on either node to see its status:

```
sudo gluster volume info

Volume Name: gv0
Type: Replicate
Status: Started
Number of Bricks: 2
Transport-type: tcp
Bricks:
Brick1: 192.168.0.121:/srv/gv0
Brick2: 192.168.0.122:/srv/gv0
```

Configure the GlusterFS Client

Now that the volume is started, I can mount it as a GlusterFS type filesystem from any client that has GlusterFS support. First though, I will want to mount it from my two Raspberry Pis as I want them to be able to write to the volume themselves. To do this, I will create a new mountpoint on my filesystem on each Raspberry Pi and use the mount command to mount the volume on it:

```
sudo mkdir -p /mnt/gluster1
sudo mount -t glusterfs 192.168.0.121:/gv0 /mnt/gluster1
df
Filesystem         1K-blocks    Used Available Use% Mounted on
rootfs               1804128 1496464    216016  88% /
/dev/root            1804128 1496464    216016  88% /
devtmpfs               86184       0     86184   0% /dev
tmpfs                  18888     216     18672   2% /run
tmpfs                   5120       0      5120   0% /run/lock
tmpfs                  37760       0     37760   0% /run/shm
/dev/mmcblk0p1         57288   18960     38328  34% /boot
192.168.0.121:/gv0   1804032 1496448    215936  88% /mnt/gluster1
```

The more pedantic readers among you may be saying to yourselves, "Wait a minute, if I am specifying a specific IP address here, what happens when 192.168.0.121 goes down?" It turns out that this IP address is used only to pull down the complete list of bricks used in the volume, and from that point on, the redundant list of bricks is what will be used when accessing the volume.

Once you mount the filesystem, play around with creating files and then looking into /srv/gv0. You should be able to see (but again, don't touch) files that you've created from /mnt/gluster1 on the /srv/gv0 bricks on both nodes in your cluster:

```
pi@pi1 ~ $ sudo touch /mnt/gluster1/test1
pi@pi1 ~ $ ls /mnt/gluster1/test1
/mnt/gluster1/test1
pi@pi1 ~ $ ls /srv/gv0
test1
pi@pi2 ~ $ ls /srv/gv0
test1
```

After you are satisfied that you can mount the volume, make it permanent by adding an entry like the following to the /etc/fstab file on your Raspberry Pis:

```
192.168.0.121:/gv0  /mnt/gluster1  glusterfs  defaults,_netdev  0  0
```

Note that if you also want to access this GlusterFS volume from other clients on your network, just install the GlusterFS client package for your distribution (for Debian-based distributions, it's called glusterfs-client), and then create a mountpoint and perform the same mount command as I listed above.

Test Redundancy

Now that I have a redundant filesystem in place, let's test it. Since I want to make sure that I could take down either of the two nodes and still have access to the files, I configured a separate client to mount this GlusterFS volume. Then I created a simple script called glustertest inside the volume:

```
#!/bin/bash

while [ 1 ]
do
  date > /mnt/gluster1/test1
  cat /mnt/gluster1/test1
  sleep 1
done
```

This script runs in an infinite loop and just copies the current date into a file inside the GlusterFS volume and then cats it back to the screen. Once I make the file executable and run it, I should see a new date pop up about every second:

```
chmod a+x /mnt/gluster1/glustertest
/mnt/gluster1/glustertest
Sat Mar  9 13:19:02 PST 2013
Sat Mar  9 13:19:04 PST 2013
Sat Mar  9 13:19:05 PST 2013
Sat Mar  9 13:19:06 PST 2013
Sat Mar  9 13:19:07 PST 2013
Sat Mar  9 13:19:08 PST 2013
```

I noticed every now and then that the output would skip a second, but in this case, I think it was just a function of the date command not being executed exactly one second apart every time, so every now and then that extra sub-second it would take to run a loop would add up.

After I started the script, I then logged in to the first Raspberry Pi and typed sudo reboot to reboot it. The script kept on running just fine, and if there were any hiccups

along the way, I couldn't tell it apart from the occasional skipping of a second that I saw beforehand. Once the first Raspberry Pi came back up, I repeated the reboot on the second one, just to confirm that I could lose either node and still be fine. This kind of redundancy is not bad considering this took only a couple commands.

There you have it. Now you have the foundation set with a redundant file store across two Raspberry Pis. Next, I will build on top of the foundation by adding a new redundant service that takes advantage of the shared storage.

Web Servers Across a Raspberry Pi Cluster

Now that I have the storage up and tested, I'd like to set up these Raspberry Pis as a fault-tolerant Web cluster. Granted, Raspberry Pis don't have speedy processors or a lot of RAM, but they still have more than enough resources to act as a Web server for static files. Although the example I'm going to give is very simplistic, that's intentional–the idea is that once you have validated that a simple static site can be hosted on redundant Raspberry Pis, you can expand that with some more sophisticated content yourself.

Install Nginx

Although I like Apache just fine, for a limited-resource Web server serving static files, something like nginx has the right blend of features, speed and low resource consumption that make it ideal for this site. Nginx is available in the default Raspbian package repository, so I log in to the first Raspberry Pi in the cluster and run:

```
sudo apt update
sudo apt install nginx
```

Once nginx installed, I created a new basic nginx configuration at /mnt/gluster1/cluster that contains the following config:

```
server {
  root /mnt/gluster1/www;
  index index.html index.htm;
  server_name twopir twopir.example.com;

  location / {
    try_files $uri $uri/ /index.html;
  }
}
```

> **Note:** I decided to name the service twopir, but you would change this to whatever hostname you want to use for the site. Also notice that I set the document root to /mnt/gluster1/www. This way, I can put all of my static files onto shared storage so they are available from either host.

Now that I have an nginx config, I need to move the default nginx config out of the way and set up this config to be the default. Under Debian, nginx organizes its files a lot like Apache with sites-available and sites-enabled directories. Virtual host configs

are stored in sites-available, and sites-enabled contains symlinks to those configs that you want to enable. Here are the steps I performed on the first Raspberry Pi:

```
cd /etc/nginx/sites-available
sudo ln -s /mnt/gluster1/cluster .
cd /etc/nginx/sites-enabled
sudo rm default
sudo ln -s /etc/nginx/sites-available/cluster .
```

Now I have a configuration in place but no document root to serve. The next step is to create a /mnt/gluster1/www directory and copy over the default nginx index.html file to it. Of course, you probably would want to create your own custom index.html file here instead, but copying a file is a good start:

```
sudo mkdir /mnt/gluster1/www
cp /usr/share/nginx/www/index.html /mnt/gluster1/www
```

With the document root in place, I can restart the nginx service:

```
sudo /etc/init.d/nginx restart
```

Now I can go to my DNS server and make sure I have an A record for twopir that points to my first Raspberry Pi at 192.168.0.121. In your case, of course, you would update your DNS server with your hostname and IP. Now I would open up http://twopir/ in a browser and confirm that I see the default nginx page. If I look at the /var/log/nginx/access.log file, I should see evidence that I hit the page.

Once I've validated that the Web server works on the first Raspberry Pi, it's time to duplicate some of the work on the second Raspberry Pi. Because I'm storing configurations on the shared GlusterFS storage, really all I need to do is install nginx, create the proper symlinks to enable my custom nginx config and restart nginx:

```
sudo apt-get update
sudo apt-get install nginx
cd /etc/nginx/sites-available
sudo ln -s /mnt/gluster1/cluster .
cd /etc/nginx/sites-enabled
sudo rm default
sudo ln -s /etc/nginx/sites-available/cluster .
sudo /etc/init.d/nginx restart
```

Two DNS A Records

So, now I have two Web hosts that can host the same content, but the next step in this process is an important part of what makes this setup redundant. Although you definitely could set up a service like heartbeat with some sort of floating IP address that changed from one Raspberry Pi to the next depending on what was up, an even better approach is to use two DNS A records for the same hostname that point to each of the Raspberry Pi IPs. Some people refer to this as DNS load balancing, because by default, DNS lookups for a hostname that has multiple A records will return the results in random order each time you make the request:

```
dig twopir.example.com A +short
192.168.0.121
192.168.0.122
dig twopir.example.com A +short
192.168.0.122
192.168.0.121
```

Because the results are returned in random order, clients should get sent evenly between the different hosts, and in effect, multiple A records do result in a form of load balancing. What interests me about a host having multiple A records though isn't as much the load balancing as how a Web browser handles failure. When a browser gets two A records for a Web host, and the first host is unavailable, the browser almost immediately will fail over to the next A record in the list. This failover is fast enough that in many cases it's imperceptible to the user and definitely is much faster than the kind of failover you might see in a traditional heartbeat cluster.

So, go to the same DNS server you used to add the first A record and add a second record that references the same hostname but a different IP address–the IP address of the second host in the cluster. Once you save your changes, perform a dig query like I performed above and you should get two IP addresses back.

Once you have two A records set up, the cluster is basically ready for use and is fault-tolerant. Open two terminals and log in to each Raspberry Pi, and run `tail -f /var/log/nginx/access.log` so you can watch the Web server access then load your page in a Web browser. You should see activity on the access logs on one of the servers but not the other. Now refresh a few times, and you'll notice that your browser should be sticking to a single Web server. After you feel satisfied that your requests are going to that server successfully, reboot it while refreshing the Web page multiple times. If you see a blip at all, it should be a short one, because the moment the Web server drops, you should be redirected to the second Raspberry Pi and be able to see the same index page. You also should see activity in the access logs. Once the first Raspberry Pi comes back from the reboot, you probably will not even be able to notice from the perspective of the Web browser.

Experiment with rebooting one Raspberry Pi at a time, and you should see that as long as you have one server available, the site stays up. Although this is a simplistic example, all you have to do now is copy over any other static Web content you want to serve into /mnt/gluster1/www, and enjoy your new low-cost fault-tolerant Web cluster.

3.5 Home As Your Backup Data Center

New Linux users often ask me "what is the best way to learn about Linux?" My advice always comes down to this: install and use Linux (any distribution will do but something stable works better), and play around with it. Inevitably, you will break something, and then instead of re-installing, force yourself to fix what you broke. That's my advice, because I've personally learned more about Linux by fixing my

own problems than just about any other way. After years of doing this, you start to build confidence in your Linux troubleshooting skills, so that no matter what problem comes your way, you figure if you work at it long enough, you can solve it.

That confidence was put to the test recently when I had a problem with a KVM host. After a power outage, it refused to boot a virtual machine that was my primary personal server for just about everything. In this section, I walk through a problem that almost had me stumped and show how I was able to find a solution in an unorthodox place (at least for me).

The Setup

Before I dive too deep into my problem, it would help to understand my setup. Although I do have servers at home, my primary server is colocated in a data center. I share the server with a friend, so the physical server simply acts as a secured KVM host, and I split the server's RAM and CPU across two virtual machines 50/50. All of my most important services from my primary DNS server and e-mail for me and my immediate family, a number of different Web sites and blogs, and even my main Irssi session sits on one of those two VMs. I end up hosting secondary DNS and e-mail from a server on my home connection, but due to a one-megabit upstream connection, I don't host much else at home for the outside world.

One day (while a relative happened to be visiting from out of town), I noticed that both my main server and the physical server that was hosting it were unavailable. I notified my contact at the data center, and it ended up being an accidental power outage that affected my cabinet. I was taking my relative out to the coast for the day, far away from decent cell-phone reception. So, since there wasn't much I could do, I assumed that long before I got back into town that afternoon, power would be restored, and other than losing over a year's uptime, I would be back up and running.

Everything but the Sync

The first time I knew there was a real problem was when I got back into town and my main server still was down. I could log in to the physical host, however; so at first I wasn't too worried. After all, I had seen KVM instances not recover from a physical host reboot before. In the past, it was either from not setting a VM to start at boot or sometimes even a wayward libvirt apparmor profile that got in the way. Usually once I logged in to the physical host, I could change any bad settings, disable any troublesome apparmor module, then manually launch my VM with virsh. This time was different.

When my VM wouldn't boot manually, I was ready to blame AppArmor. It had blocked VMs from booting in the past, but this time, neither setting the libvirtd AppArmor module to complain mode, disabling all AppArmor modules nor even forcefully stopping AppArmor seemed to help. I even resorted to rebooting the physical host to heed AppArmor's warning that forcibly stopping it after it was

running may cause some modules to misbehave. Nothing helped. When I connected a console to the VM as it booted, I started seeing initial kernel errors as though it was having trouble mounting the root filesystem. Great. Did the power outage corrupt my data?

The next step in the troubleshooting process was to attempt to boot from a rescue disk. With KVM, it's relatively easy to add a local ISO image as though it were a CD-ROM. So after not much effort, I discovered I could, in fact, boot a rescue disk and confirmed from the rescue disk I could mount my VM's drives, and the data did not seem corrupted. So then why wouldn't it boot? After I ran a manual fsck from the rescue disk, I attempted to reload GRUB, and that was when I got my first strange clue about the nature of the problem–even from the rescue disk, I wasn't able to write to the filesystem reliably. I would get virtual ATA resets, even though I seemed to be able to read fine.

So, I assumed I had some level of corruption with that particular VM, but because my data wasn't affected, I figured in the worst case, I could spawn a fresh VM and migrate the data over. So, that's what I tried next using the ubuntu-vm-builder wrapper script I used previously to build my VM. The VM seemed to spawn fine; however, once again, even this brand-new VM refused to boot properly and had the same strange disk errors.

It was at this point that my troubleshooting steps start to get a bit hazy, because I started trying more desperate things. I booted different kernel versions in GRUB (after all, the kernel had been updated a few times in the year the server had been up). I audited all of the filesystem permissions on my VM disk images, and I tried to launch the VMs as root just in case. I even tried converting one VM's disks from qcow2 to raw with no results. Even Web searches came up empty. This server had been down longer than it ever had before, and I was starting to run out of options.

The Sync

My first break came when I decided to copy the VM I had just spawned over to almost identical hardware I had at home with the same distribution and see if I could reproduce the problem there. I picked the new host simply because since qcow2 filesystems grow on demand, it happened to have the smallest disks and was the fastest to sync over. The process was pretty straightforward. First I exported that KVM instance's configuration XML file with virsh on the colocated host:

```
virsh dumpxml test1.example.net > test1.example.net.xml
```

Then I copied that XML file to my home server, created a local directory named after this VM to store its disk images and synced them over from the physical host:

```
mkdir test1.example.net
rsync -avx -progress remotehost:/var/lib/libvirt/
 ↪ images/test1.example.net/test1.example.net/
```

Once the disk images were copied, I had to edit the test1.example.net.xml file, because the disk images now were stored in a new location. After I did that, I used virsh again to import this XML configuration file and start the VM:

```
virsh define test1.example.net.xml
virsh start test1.example.net
```

The VM actually started! Although I still had no idea what the problem was on the colocated server, I felt pretty confident that if I could sync over my main server, it would run on this home machine. Of course, with a 12Mb-down, 1Mb-up connection at home, it was going to take a bit longer to copy the 45GB disk images for this VM. Other than the time it took, the process was essentially the same as with the test machine, except once the host booted, I had to change its network configuration to reflect its new public IP.

With my server back up and running, I just had to change a number of DNS entries and firewall rules to reflect the new IP, and even with my slower upstream connection at home, I at least had some breathing room to troubleshoot the problem on the colocated server.

The Last Resort

Now that my VM and its data were safe and services were restored (if a bit slow), I felt free to perform more drastic steps on my colocated server. The first step was trying to figure out what was so different about it compared to my home server. They had the same Ubuntu 10.04 server install and most of the same packages. Luckily, I had a number of old cached libvirt and KVM packages on my home server, so at first I iterated through all of those packages to see if the problem was due to some upgrade. Once I exhausted that, I tried different kernel versions on the physical host and still no results.

Believe me when I tell you that during that week I tried every troubleshooting measure I could think of before I finally went to the second-to-last resort. The fact that I was even considering this should tell you how desperate I was getting. The last resort would be to do a complete re-install from scratchsomething I wasn't ready to do yet. I was desperate enough though that I went with the second-to-last resort: an in-place distribution upgrade from 10.04 to 12.04. Once the dust settled, I tried my small test image, and it actually worked. We were back in business.

The Sync Back

Well, we were almost back in business. See, I had been using that server at home for a number of days now, and between the e-mail, blogs and other services, it had a lot of new data on it. This meant I couldn't just start up the image that was already on the colocated server. I had to sync up the changes from my home server.

The real trick to this was that I couldn't just sync the server hot. For one, the disk would be changing all the time, and two, I didn't want to risk having the same server

running in weird states on two different physical hosts. This meant syncing the actual disk images. The problem was that while the 45GB disk images synced to my house relatively quickly over my 12Mb-downstream (plus the server was already down at the time, so downtime wasn't a consideration), syncing the same data up with my 1Mb upstream was going to take a long time–too long for a pure cold sync to be a solution, as I just couldn't have that much downtime.

The solution here was going to be two-fold, and it was based on a few assumptions I could make:

- Although a fair number of files had changed on my local VM instance, the actual size of the change was relatively small compared to the size of the disk images.
- rsync has an excellent mechanism for syncing over only the parts of large files that have changed.
- A lot of the changes in my qcow2 files were likely going to be at the end of those files anyway.
- If I use rsync with the `--inplace` option, it will modify the existing disk image on the remote machine directly and save disk space and time.

So my plan for phase 1 was to run rsync from physical host to physical host and sync over the qcow2 disk images hot while the VM was running and tell rsync to sync the disk images in place. Because I could assume the remote images would be somewhat corrupted anyway (that's the downside of syncing a disk image while the disk is being used), I didn't have to care about `--inplace` leaving behind a potentially corrupted file if it were stopped midway through the sync. I could clean it up later.

The advantage of doing the phase 1 rsync hot was that I could get all of the main differences between the home and colocated images sorted out while the server was still running at home. I even could potentially run that rsync multiple times leading up to phase 2 just to make sure it was as up to date as it could be. Here are the rsync commands I used to perform the phase 1 hot sync:

```
rsync -avz -progress -inplace disk0.qcow2
 ↪ remotehost:/var/lib/libvirt/images/www.example.net/disk0.qcow2
rsync -avz -progress -inplace disk1.qcow2
 ↪ remotehost:/var/lib/libvirt/images/www.example.net/disk1.qcow2
```

Between rsync's syncing only the bits that changed and the fact that I used -z to compress the data before it was transferred, I was able to sync these files way faster than you would think possible on a 1Mb connection. Of course, these commands ended up saturating my bandwidth at home, so since I wasn't under time pressure for the hot sync to complete, I ended up setting a bandwidth limit of 10 kilobytes per second for the larger disk1.qcow2 image:

```
rsync -avz -progress -inplace -bwlimit=10 disk1.qcow2
 ↪ remotehost:/var/lib/libvirt/images/www.example.net/disk1.qcow2
```

Once phase 1 was complete, I could start with phase 2. I needed the phase 2 rsync to run while the VM was powered off so I could make sure the disk wasn't being written

to during the sync. Otherwise, I would risk corruption on the filesystem. Because this required downtime, I picked a proper maintenance window for my server when it would be less busy, finished a final phase 1 hot sync a few hours before, then halted the VM cleanly before I performed the final syncs:

```
rsync -avz -progress -inplace disk0.qcow2
 ↪ remotehost:/var/lib/libvirt/images/www.example.net/disk0.qcow2
rsync -avz -progress -inplace disk1.qcow2
 ↪ remotehost:/var/lib/libvirt/images/www.example.net/disk1.qcow2
```

Because of the previous work of syncing up the disk images, the final cold sync took only an hour or two with most of the time being spent with rsync seeking between the local and remote image to confirm they were in sync. Once the commands completed, I was able to power up the server again on my colocated host, change its IPs back, and I was back in business.