Chaos Engineering

Crash test your applications

Mikolaj Pawlikowski

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Chaos Engineering
Crash test your applications
Version 5

For more information on this and other Manning titles go to
manning.com

welcome

Thanks for purchasing the MEAP for *Chaos Engineering in Action*. I'm really glad you did, and I can't wait to hear back from you! You're now part of a crucial group of people who have the power to make this book its best possible version. Kudos for that!

When I first started with chaos engineering, I was leading the design and building of a large scale microservices platform similar to AWS Lambda, but with some extra bells and whistles. At Bloomberg, in the financial sector, reliability is of paramount importance. We decided to adopt Kubernetes and found ourselves with a ton of machines running two tons of moving parts, many of which were very new. What can you do to sleep better at night, instead of worrying that your testing (or one of your dependencies) missed something important?

The answer, of course, is chaos engineering. Inspired by the work at Netflix, but with a slightly different set of constraints, we decided to experiment with the system and detect how it breaks, instead of waiting for it to break on us. Fast forward a few years and chaos engineering has become a part of our culture and has brought us a lot of value. With this book, I'm hoping to generate that value for you and your teams.

This book is different from most other books you've read in two ways. First, it cuts across stacks, technologies, and environments. Chaos engineering can be applied to any system and technology, so you'll find we'll wander around a fair bit. Don't worry if you don't know a particular technology - you can still learn the principles and apply them in other situations. Second, it comes with a VM, and I strongly recommend using it to follow the examples around. This will ensure that we're using the same software in the same environment and will allow us to do braver things inside of the VM, without having to worry about breaking the host system. You can always recreate it in seconds. It also comes with all the source code, examples and tools we're going to use pre-installed, so once you have it, you should be able to follow chapters on the go, without having to download things from the internet.

The book uses Linux as the OS, since it runs a vast majority of servers in the world. Basic Linux skills are assumed, most non-basic stuff will be explained. Chapters talking about a particular technology (Docker, Kubernetes, JVM, Chrome...) should be self-contained, and take you from a vague idea of the technology to knowing how to test its reliability.

Finally, a lot of programming books are rather boring. It is my hope to make this one entertaining. After all, we're doing computer science first and foremost because it's fun, and we should remember that.

Go forth, have fun, and tell me how to make the book better. Thanks again and talk to you soon! If you have any questions, comments, or suggestions, please share them in Manning's Author Online forum for my book.

-- Mikolaj Pawlikowski

brief contents

1

Into the world of chaos engineering

This chapter covers

- What *chaos engineering* is and is not
- Motivations for doing chaos engineering
- Anatomy of a chaos experiment
- A simple example of chaos engineering in practice

What would you do to make absolutely sure the car you're designing is safe? A typical vehicle today is a real wonder of engineering. A plethora of sub-systems operating everything from rain-detecting wipers to life-saving airbags, all coming together to not only go from A to B, but to protect the passengers during an accident. Isn't it moving when your loyal car gives up the ghost to save yours through the strategic use of crumple zones, from which it will never recover?

Passenger safety being the highest priority, all these parts go through rigorous testing. But even assuming they all work as designed, does that guarantee you'll survive in a real world accident? If your business card says "New Car Assessment Programme," you demonstrably don't think so. Presumably that's why every new car making it to the market goes through crash tests.

Picture this: a production car, heading at a controlled speed, closely observed with high speed cameras, in a life-like scenario crashing into an obstacle to test the system as a whole. In many ways, **chaos engineering is to software systems what crash tests are to the car industry**: it's a deliberate practice of experimentation designed to uncover systemic problems. In this book we'll look at the why, when and how to apply chaos engineering to improve your computer systems. And perhaps, who knows, save some lives in the process. What's a better place to start than a nuclear power plant?

1.1 What is chaos engineering?

Imagine you're responsible for designing the software operating a nuclear power plant. Your job description, among other things, is to prevent a radioactive fallout. The stakes are high, a failure of your code can produce a disaster leaving people dead and rendering vast lands uninhabitable. You need to be ready for anything from earthquakes, power cuts, floods, hardware failures to terrorist attacks. What do you do?

You hire the best programmers, set in place a rigorous review process, test coverage targets, and walk around the hall reminding everyone that we're doing serious business here. But "yes, we have 100% test coverage, Mr. President!" will not fly at the next meeting. You need contingency plans, you need to be able to demonstrate that when bad things happen, the system as a whole can withstand it and the name of your power plant stays away from the news headlines. You need to go looking for problems before they find you. That's what this book is about.

Chaos engineering is defined as "the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production" (Principles of Chaos Engineering, http://principlesofchaos.org/). In other words, it's a software testing method focusing on finding evidence of problems before they are experienced by the users.

You want your systems to be reliable (we'll look into that), and that's why you work hard on producing good quality code and good test coverage. Yet, even if your code works as intended, in the real world there are plenty of things that can (and will) go wrong. The list of things that can break is longer than that of possible side-effects of your painkillers. Starting with sinister-sounding events like floods and earthquakes, which can take down entire data centers, through power supply cuts, hardware failure, networking problems, resource starvation, race conditions, unexpected peaks of traffic, complex and unaccounted-for interactions between elements in your system, all the way to the evergreen operator (human) error. And the more sophisticated and complex your system is, the more opportunity there is for problems to appear.

It's tempting to discard these as rare events, but they just keep happening. So far, 2019 has seen two crash landings on the surface of the Moon: the Indian Chandrayaan-2 mission (https://en.wikipedia.org/wiki/Chandrayaan-2#Loss_of_Vikram) and the Israeli Beresheet (https://en.wikipedia.org/wiki/SpaceIL#Failed_landing) both lost on lunar descent. And remember that even if you did everything right, more often than not you still depend on other systems, and these systems fail. For example, during last summer, Google Cloud[1], Cloudflare, Facebook (WhatsApp) and Apple all had major outages in a space of just over a month (https://blog.thousandeyes.com/looking-back-biggest-internet-outages-2019/). If your software ran on Google Cloud or relied on Cloudflare for routing, you were potentially affected. That's just the reality of this.

It's a common misconception that chaos engineering is only about randomly breaking things in production. It's not. Although running experiments in production is a unique part of chaos engineering (more on that later), it's about much more than that - anything that helps us be confident the system can withstand turbulence. It interfaces with Site Reliability

[1] You can see the official, detailed report at https://status.cloud.google.com/incident/cloud-networking/19009

Engineering (SRE), application and systems performance analysis and other forms of testing. Practicing *chaos engineering* can help you prepare for failure and by doing that learn to build better systems, improve the existing ones and make the world a safer place.

1.2 Motivations for chaos engineering

At the risk of sounding like an infomercial, there are at least three good reasons to implement chaos engineering:

1. Risk, cost and service-level indicators, objectives, and agreements verification
2. Testing a system (often complex and distributed) as a whole
3. Finding emergent properties you were unaware of

Let's take a closer look at these motivations.

1.2.1 Risk, cost and service-level indicators, objectives, and agreements (SL{I,O,A})

You want your computer systems to run well, and the subjective definition of what "well" means depends on the nature of the system and your goals regarding it. Most of the time, the primary motivation for companies is to create profit for the owners and shareholders. The definition of "running well" will therefore be a derivative of the business model objectives.

Let's say you're working on a planet-scale website for sharing photos of cats and toddlers and checking on your high-school ex, called Bookface. Your business model might be to serve your users targeted ads in which case you will want to balance the total cost of running the system with the amount of money you can sell these ads for. From an engineering perspective, one of the main risks is that the entire site is down and you can't serve ads and bring home the revenue. Conversely, not being able to display a single cat picture in the rare event of a problem with the cat pictures server is probably not a deal breaker, and will only affect your bottom line in a small way.

With both of these risks (users can't use the website and users can't access a cat photo momentarily) you can estimate the associated cost, expressed in dollars per unit of time. That cost includes the direct loss of business as well as various other, less tangible things like public image damage, that might be equally important. As a real-life example, Forbes once estimated[2] Amazon to lose $66,240 per minute of their website being down.

Now, in order to quantify these risks, the industry uses the idea of service level indicators (SLI). In our example, the percentage of the time that your users can access the website could be an SLI. And so could the ratio of requests that are successfully served by the cat photos service within some time window. The SLIs are there to put a number next to an event, and picking the right SLI is important.

Two parties agreeing on a certain range of an SLI can form a service level objective (SLO), that's a tangible target that the engineering team can work toward. SLOs, in turn, can be legally enforced as a service level agreement (SLA), in which one party agrees to guarantee a certain SLO or otherwise pay some form of penalties if they fail to do so.

[2] https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/

Going back to our cats and toddlers photo-sharing website, one possible way to work out the risk, SLI and SLO could look like this:

- The main risk is "people can't access the website", or simply the *downtime*
- A corresponding SLI could be "the ratio of success responses to errors from our servers"
- An SLO for the engineering team to work toward: "the ratio of success responses to errors from our servers > 99.95% on average monthly"

To give you a different example, imagine a financial trading platform, where people hit an API when their algorithms want to buy or sell commodities on the global markets. The speed is critical. We could imagine a different set of constraints, set on the trading API:

- SLI: "99th percentile response time"
- SLO: "99th percentile response time < 25ms, 99.999% of the time"

From the engineering team perspective, that sounds like mission impossible: we allow ourselves about only 5 minutes in a year, where the top 1% of the slowest requests average over 25ms response time. Building a system like that might be very difficult and expensive.

Number of nines

In the context of SLOs, we often talk about the number of nines to mean specific percentages. For example, 99% is "two nines", 99.9% is three nines, 99.999% is five nines and so on. Sometimes, we also use phrases like "three nines five" or "three and a half nines" to mean 99.95%, although the latter is not technically correct (going from 99.9% to 99.95% is a factor of 2, but going from 99.9% to 99.99% is a factor of 5). Note a few of the most common values and their corresponding downtimes per year and per day are the following:

- 90% ("one nine") - 36.53 days per year, or 2.4 hours per day
- 99% ("two nines") - 3.65 days per year, or 14.40 minutes per day
- 99.95% ("three and a half nines") - 4.38 hours per year, or 43.20 seconds per day
- 99.999% ("five nines") - 5.26 minutes per year, or 840 milliseconds per day

How does chaos engineering help with these? In order to satisfy the SLOs, you are going to engineer the system in a certain way. You will need to take into account the various sinister scenarios, and the best way to see if the system works fine in these conditions is to go and create them - which is exactly what chaos engineering is about! We're effectively working backward from the business goals, to an engineering-friendly defined SLO, that we can in turn continuously test against using *chaos engineering*. Notice that in all of the examples above, we were talking in terms of entire systems.

1.2.2 Testing a system as a whole

Various testing techniques approach software at different levels. Unit tests typically cover single functions or smaller modules in isolation. End to end (e2e) tests and integration tests work on a higher level, where whole components are put together to mimic a real system, and some verification is done that assures that the system does what it should.

Benchmarking is yet another form of testing, where we are interested in the performance of a piece of code, which can be lower level (for example, micro-benchmarking a single function) or a whole system (e.g. simulating client calls).

I like to think of chaos engineering as the next logical step - a little bit like e2e testing, but during which we rig the conditions to introduce the type of failure we expect to see, and measure that we still get the correct answer within the expected time frame. It's also worth noting, as you'll see in part 2, that even a single-process system can be tested using chaos engineering techniques and sometimes that comes in really handy.

1.2.3 Emergent properties

Our complex systems often exhibit *emergent properties* that we didn't initially intend for. An example of an emergent property in the real world would be a human heart: the single cells don't have the property of pumping blood, but the right configuration of cells produces a heart that keeps us alive. The same way our neurons don't "think" but their interconnected collection that we call "brain" does, as you're illustrating by reading these lines.

In computer systems properties often emerge from the interactions between the moving parts the system is built of. Let's consider an example. Imagine that you run a system with many services, all using a DNS server to find each other. Each service is designed to handle DNS errors by retrying up to 10 times. Similarly, the external users of the systems are told to retry if their requests ever fail. Now, imagine that for whatever reason the DNS server fails and restarts. When it comes back up, it sees an amount of traffic amplified by the layers of retries, an amount that it wasn't set up to handle. So it might fail again, and get stuck in an infinite loop restarting, while the system as a whole is down. None of the components of the system have the property of creating infinite downtime, but together, with the right timing of events, the system as a whole might go into that state.

Although certainly less exciting than the example of consciousness I mentioned before, this property emerging from the interactions between the parts of the system is a real problem to deal with. This kind of unexpected behavior can have very serious consequences on any systems, especially the large ones. The good news is that chaos engineering excels at finding issues like that. By experimenting on real systems, oftentimes we can discover how simple, predictable failures can cascade into large problems. And once we know about them, we can fix them.

Chaos engineering and randomness

When doing chaos engineering we can often use the element of randomness and borrow from the practice of fuzzing - feeding pseudo-random payloads to a piece of software in order to try to come up with an error that our purposely written tests might be missing. The randomness definitely can be helpful, but once again, I would like to stress that controlling the experiments is necessary to be able to understand the results - it's not about randomly breaking things.

Hopefully I've had your curiosity and now I've got your attention. Let's see how to do chaos engineering!

1.3 Four steps to chaos engineering

Chaos engineering experiments (*chaos experiments* for short) are the basic unit of *chaos engineering*. We do *chaos engineering* through a series of *chaos experiments*. Given a computer system and a certain number of characteristics we are interested in, we design experiments to see how the system fares when bad things happen. In each experiment, we focus on proving or refuting our assumptions about how the system will be affected by a certain condition.

For example, imagine you are running a popular website and you own an entire datacenter. You need your website to survive power cuts, so you make sure two independent power sources are installed in the datacenter. In theory, you are covered, but in practice, a lot can still go wrong. Perhaps the automatic switching between power sources doesn't work? Or maybe your website has grown since the launch of the datacenter, and a single power source no longer provides enough electricity for all the servers? Did you remember to pay an electrician for the regular checkup of the appliances every 3 months?

If you feel worried, you should. Fortunately, *chaos engineering* can help you sleep better. We can design a very simple *chaos experiment* which will scientifically tell us what happens when one of the power supplies goes down (for more dramatic effect always pick the newest intern to run these steps):

1. Repeat for all POWER SOURCES, one at a time:
    a) Check that The Website is up
    b) Open the electric cupboard and turn POWER SOURCE off
    c) Check that The Website is still up
    d) Turn POWER SOURCE back on

It is crude, it sounds obvious, but let's go through what we just did. Given a computer system (a datacenter) and a characteristic (survives a single power source failure), we designed an experiment (switch a power source off and eyeball whether The Website is still up) that increases our confidence in the system withstanding a power problem. We used science for the good, and it only took a minute to set up. *That's one small step for man, one giant leap for mankind.*

Before we pat ourselves on the back, though, it's worth asking what would happen if the experiment failed and the datacenter went down? In this overly-crude-for-demonstration-purposes case, we would create an outage of our own. A big part of our job will be about minimizing the risks coming from our experiments and choosing the right environment to execute them. More on that later.

Take a look at figure 1.1 which summarizes the process we just went through. When you're back, let me anticipate your first question: what if we are dealing with more complex problems?

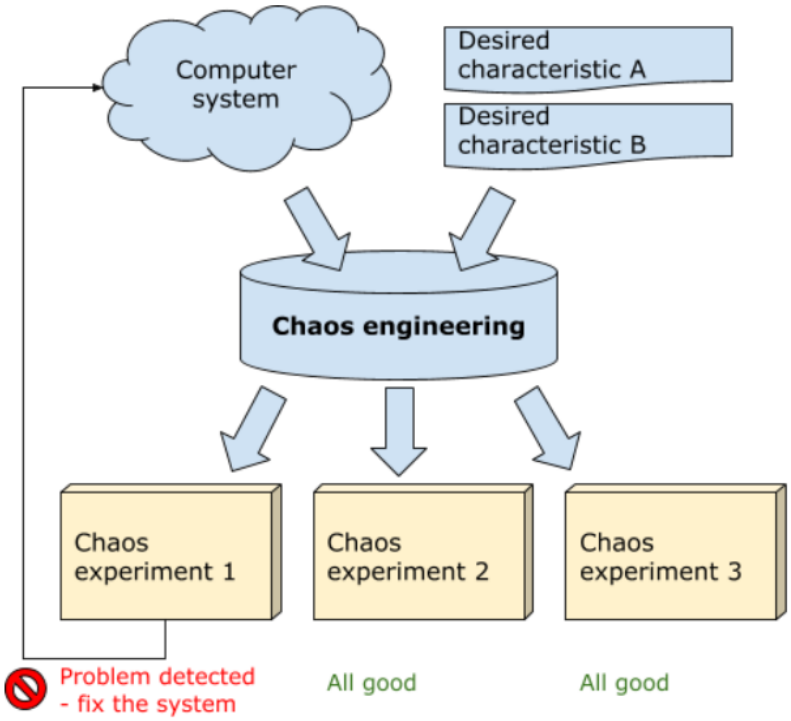


Figure 1.1 The process of doing *chaos engineering* through a series of *chaos experiments*

Like with any experiment, we start with an idea of a hypothesis that we want to prove or disprove and we design the entire experience around that idea. When Gregor Mendel had an intuition about the laws of heredity, he designed a series of experiments on yellow and green peas proving the existence of dominant and recessive traits. His results didn't follow the expectations, and that's perfectly fine; in fact, that's how his breakthrough in genetics was made.[3] We will be drawing inspiration from his experiments throughout the book, but before we get into the details of good craftsmanship in designing our experiments, let's plant a seed of an idea about what we're looking for.

Let's zoom in on one of these *chaos experiment* boxes from figure 1.1, and see what that's made of. Let me guide you through figure 1.2 that describes the simple, four-step process to design an experiment like that.

First, we need to be able to observe our results. Whether it's the color of the resulting peas, the crash test dummy having all limbs in place, our website being up, the CPU load, the number of requests per second or the latency of successful requests, the first step is to ensure that we can accurately read the value for these variables. We're lucky dealing with

[3] He did have to wait a couple of decades for anyone to reproduce his findings and for mainstream science to appreciate it and mark it 'a breakthrough'. But let's ignore that for now.

computers in the sense that we can often produce very accurate and very detailed data easily. We will call this "*observability*."

Second, using the data we observe, we need to define what's normal. This is so that we can understand when things are out of the expected range. For instance, we might expect the CPU load on 15 minute average to be below 20% for our application servers during the working week. Or 500 to 700 requests per second per instance of our application server running with four cores on our reference hardware specification. This normal range is often referred to as *steady state*.

Third, we shape our intuition into a hypothesis that can be proved or refuted, using the data we can reliably gather (observability). A simple example could be 'killing one of the machines doesn't affect the average service latency'.

Finally, we execute the experiment, make our measurements to conclude whether we were right. And funnily enough, we like being wrong, because that's what we learn more from. Rinse and repeat.

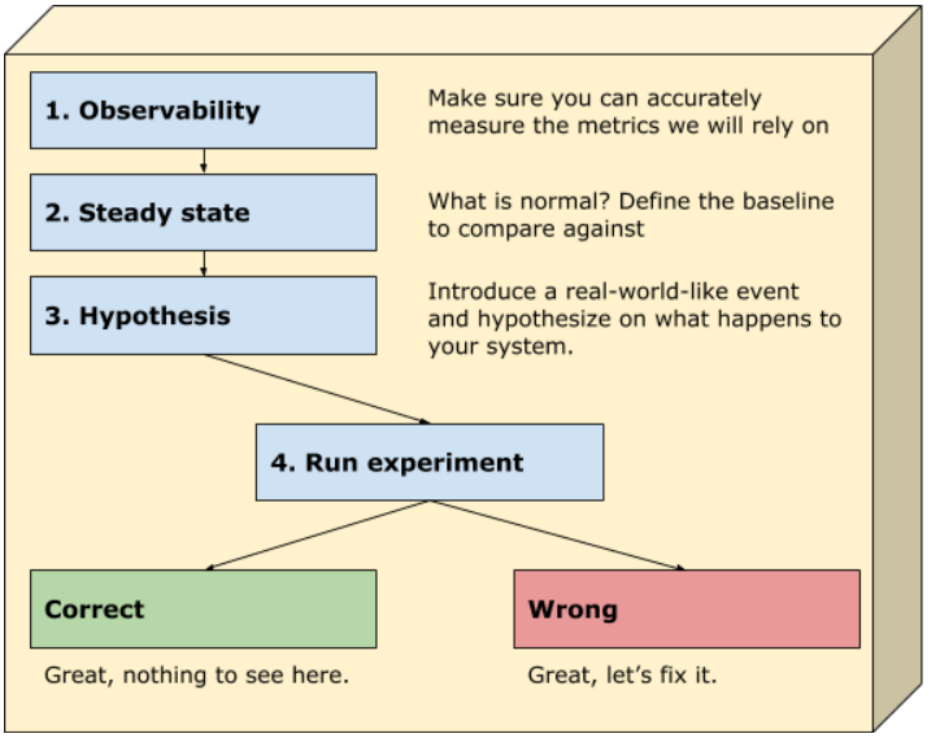


Figure 1.2 The four steps of a *chaos experiment*

The simpler your experiment, usually the better. There are no bonus points for elaborate designs, unless that's the best way of proving the hypothesis. Look at the figure 1.2 again, and let's dive just a little bit deeper, starting with *observability*.

1.3.1 Observability

I quite like the word *observability* because it's straight to the point. It means being able to see whatever metric we are interested in reliably. The keyword here is *reliably*. Working with computers, we are often spoiled - the hardware producer, or the OS already provides mechanisms for reading various metrics, from the temperature of CPUs, fan's RPM to memory usage and hooks to use for various kernel events. But at the same time, it's often easy to forget that they are subject to bugs and caveats that the end user needs to take into account. If the process you're using to measure CPU load ends up using more CPU than your application, that's probably a problem.

If you've ever seen a crash test on television, you will know it's both frightening and mesmerizing at the same time. Watching a three-thousand-pound machine accelerate to a carefully controlled speed and then fold like an origami swan on impact with a massive block of concrete is... humbling.

But the high-definition, slow-motion footage of shattered glass flying around, and seemingly unharmed (and unfazed) dummies sitting in what used to be a car just seconds before is not just for entertainment. Like any scientist who earned their white coat (and hair), both crash test specialists and chaos engineering practitioners alike need reliable data to conclude whether an experiment worked or not. That's why observability, or reliably harvesting data about a live system, is paramount.

In this book we're going to focus on Linux and the system metrics that it offers to us (CPU load, RAM usage, IO speeds) as well as go through examples of higher-level metrics from the applications we'll be experimenting on.

Observability in the quantum realm

If your youth was as filled with wild parties as mine, you might be familiar with the double-slit experiment.[4] It's one of my favorite experiments in physics, and one that displays the probabilistic nature of quantum mechanics. It's also one that has been perfected over the last 200 years by generations of physicists. The experiment in its modern form consists of shooting photons (or matter particles like electrons) at a barrier that has two parallel slits, and then observing what landed on the screen on the other side. The fun part is that, if you don't observe which slit they go through, then they behave like a wave and interfere with each other forming a pattern on the screen. But if you try to detect (observe) which slit each particle went through, then it will not behave like a wave. So much for reliable observability in quantum mechanics!

1.3.2 Steady state

Armed with the reliable data from the previous step ('observability') we need to define what's normal, so that we can measure abnormalities. A fancier way of saying that is '*to define a steady state*', and works much better at dinner parties.

What we measure will depend on the system and our goals about it. It could be '*undamaged car going straight at 60 mph*' or perhaps '*99% of our users can access our API in under 200ms*'. Often, this will be driven directly by the business strategy.

[4] https://en.wikipedia.org/wiki/Double-slit_experiment

It's important to mention that on a modern Linux server, there is going to be a lot of things going on, and we're going to try our best to isolate as many variables as possible. Let's take the example of CPU usage of our process. It sounds simple, but in practice, a lot of things can affect our reading. Is our process getting enough CPU or is it being stolen by other processes (perhaps it's a shared machine, or maybe a CRON job updating the system kicked in during our experiment)? Did the kernel schedule allocate cycles to another process with higher priority? Are we in a virtual machine, and perhaps the hypervisor decided something else needed the CPU more?

You can go deep down the rabbit hole. The good news is that often we are going to repeat our experiments many times and some of the other variables will be brought to light, but remembering that all these other factors can affect our experiments is just something we should keep at the back of our heads.

1.3.3 Hypothesis for our experiment

Now, for the real fun part. Step three is where we shape our intuitions into a testable hypothesis - essentially an educated guess of what will happen to our system in the presence of a well defined problem. Will it carry on working? Will it slow down? By how much?

In real life, these questions will often be prompted by incidents (unprompted problems we discover when things stop working), but the better we are at this game, the more we can (and should) preempt. Earlier in the chapter I listed a few examples of what tends to go wrong. These events can be broadly categorized as the following:

- External events (earthquakes, floods, fires, power cuts, and so on)
- Hardware failures (disks, CPUs, switches, cables, power supplies, and so on)
- Resource starvation (CPU, RAM, swap, disk, network)
- Software bugs (infinite loops, crashes, hacks)
- Unsupervised bottlenecks
- Unpredicted emergent properties of the system
- Virtual Machine (JVM, V8, others)
- Hardware bugs
- Human error (pushing the wrong button, sending the wrong config, pulling the wrong cable, ...)

We will look into how we can simulate these problems as we go through the concrete examples in part 2 of the book. Some of them are easy (switch off a machine to simulate machine failure or take out ethernet cable to simulate network issues), while others will be much more advanced (add latency to a system call). The choice of failures to take into the account requires a good understanding of the system we are working on.

Here are a few examples of what a hypothesis could look like:

- 'On frontal collision at 60 mph, no dummies will be squashed'
- 'If both parent peas are yellow, all the offspring will be yellow'
- 'If we take 30% of our servers down, the API continues to serve 99th percentile of requests in under 200ms'
- 'If one of our DB servers goes down, we continue meeting our SLO'

Now, it's time to run the experiment.

1.3.4 Run the experiment and prove (or refute) your hypothesis

Finally, we run the experiment, measure the results, and conclude whether we were right. Remember, being wrong is fine and much more exciting at this stage!

Everybody gets a medal in the following conditions:

- If you were right, congratulations! You just gained more confidence in your system withstanding a stormy day
- If you were wrong, congratulations! You just found a problem in our system before our clients did, and we can still fix it before anyone gets hurt!

We'll spend some time on the good craftsmanship rules in the following chapters, including automation, managing the blast radius, and testing in production. For now, just remember that as long as this is good science, we learn something from each experiment.

1.4 What chaos engineering is not

If you're just skimming it in a store, hopefully you've already gotten some value out of it. There is more coming, so don't put it away! As it's often the case, the devil is in the details, and in the coming chapters we're going to see more in depth how to execute the above four steps. I hope that by now you can clearly see the benefits of what chaos engineering has to offer, and roughly what's involved in getting to it.

But before we proceed, I'd like to make sure that you also understand what **not** to expect from these pages. Chaos engineering is not a silver bullet, doesn't automatically fix your system, cure cancer, or guarantee weight loss. In fact, it might not even be applicable to your use case or project.

A common misconception is that chaos engineering is about randomly destroying stuff. I guess the name kind of hints at it, and Chaos Monkey (https://netflix.github.io/chaosmonkey/), the first tool to get internet fame in the domain, relies on randomness quite a lot. But although randomness can be a powerful tool, and sometimes there is an overlap with fuzzing, we want to control the variables we are interacting with as closely as possible. More often than not, adding failure is the easy part: the hard part is to know where to inject it and why.

Chaos engineering is not just Chaos Monkey, Chaos Toolkit (https://chaostoolkit.org/), PowerfulSeal (https://github.com/bloomberg/powerfulseal) or any of the number of projects available on Github - these are tools making it easier to implement certain types of experiments, but the real difficulty is in learning how to look critically at systems and predict where the fragile points might be.

It's important to understand that chaos engineering doesn't replace other testing methods, like unit or integration tests. Instead, it complements them: just like airbags are tested in isolation, and then again with the rest of the car during a crash test, chaos experiments operate on a different level and test the system as a whole.

This book will not give you ready made answers on how to fix your systems. Instead it will teach you how to find problems by yourself and where to look for them. Every system is

different, and although there will be common scenarios and gotchas that we'll look at together, a deep understanding of your system's weak spots is required to come up with useful chaos experiments. In other words, the value you get out of it is going to depend on your system, how well you understand it, how deep you want to go testing it, and how well you set up your observability shop.

Although chaos engineering is unique in that it can be applied to production systems, it's not the only scenario that it caters for. A lot of content on the internet appears to be centered around "breaking things in production", quite possibly because it's the most radical thing you can do, but that's not all chaos engineering is about -- or even its main focus. A lot of value can be derived from applying the chaos engineering principles and running experiments in other environments too.

Finally, although there is some overlap, chaos engineering doesn't stem from the Chaos Theory in mathematics and physics. I know: bummer. Might be an awkward question to answer at a family reunion, so better be prepared.

With these caveats out of the way, let's get a taste of what chaos engineering is like with a small case study.

1.5 A taste of chaos engineering

Before things get technical, let's close our eyes and take a quick detour to Glanden, a fictional island country in northern Europe. Life is enjoyable for Glandeners. The geographical position provides a mild climate and the hard-working people a prosperous economy. At the heart of Glanden is Donlon, the capital with a large population of about 8 million people, all with a rich heritage from all over the world - a true cultural melting pot. It's in Donlon, that our fictitious startup FizzBuzzAAS tries really hard to *make the world a better place*.

1.5.1 FizzBuzz as a service

FizzBuzzAAS ltd is a rising star in the Donlon's booming tech scene. Started just a year ago, it has already established itself as a clear leader in the market of FizzBuzz as a Service. Recently supported by serious venture capital dollars, they are looking to expand their market reach and scale their operations. The competition, exemplified by FizzBuzzEnterpriseEdition[5] is fierce and unforgiving. Their business model is straightforward: the clients pay a flat monthly subscription fee to access the cutting-edge APIs.

Betty, their head of sales is a natural. She's about to land a big contract that could make or break the ambitious startup. Everyone's been talking about that contract at the water cooler for weeks. The tension is sky high.

Suddenly, the phone rings, and everyone goes silent. It's the Big Company calling. Betty picks up. "Mhm... Yes. I understand. '' It's so quiet you can hear the birds chirping outside. "Yes ma'am. Yes, I'll call you back. Thank you."

Betty stands up, realizing everyone's holding their breath. "Our biggest client can't access the API".

[5] https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

1.5.2 A long, dark night

It was the first time in the history of the company that the entire engineering team (Alice and Bob) pulled an all-nighter. Initially nothing made sense. They could successfully connect to each of the servers, the servers were reporting as healthy, the expected processes were running and responding, so where did the errors come from?

Moreover, their architecture really wasn't that sophisticated. An external request would hit a load balancer, which would route to one of the two instances of the API server, which would consult a cache to either server a precomputed response, if it's fresh enough, or compute a new one and store it in cache. You can see this simple architecture in figure 1.3.

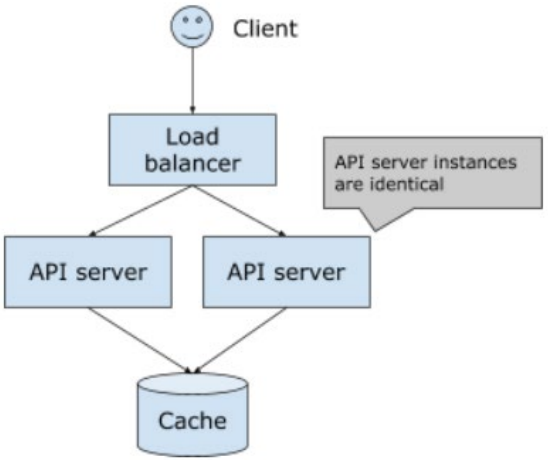


Figure 1.3 FizzBuzz as a Service technical architecture

Finally, a couple gallons of coffee into the night, Alice found the first piece of the puzzle. "It's kinda weird,'' she said as she was browsing through the logs of one of the API server instances, "I don't see any errors, but for all of these requests, they seem to stop at the cache lookup". Eureka! It wasn't long from that moment on until she found the problem: while their code handled gracefully the cache being down (connection refused, no host, and so on), but didn't have any timeouts in case of no response. It was downhill from there - a quick session of pair programming, a rapid build and deploy, and it was time for a nap.

The order of the world was restored; people could continue requesting FizzBuzz as a service, and the VC dollars were being well spent. The Big Company acknowledged the fix and didn't even mention cancelling their contract. The sun shone again. Later, it turned out that the API server's inability to connect to the cache was a result of badly rolled out firewall policy, in which someone forgot to whitelist the cache. Human error.

1.5.3 Post-mortem

"How can we make sure that we're immune to this kind of issue the next time?" Alice started, in what was destined to be a crucial meeting for the company's future. Silence. "Well, I guess we could preemptively set some of our servers on fire once in a while" answered Bob to lift up the mood just a little bit. Everyone started laughing. Everyone, apart from Alice, that is. "Bob, you're a genius!" Alice acclaimed and then took a moment to appreciate the size of everyone's eyeballs. "Let's do exactly that! If we could *simulate* a broken firewall rule like this, then we can add this to our integration tests".

"You're right!" Bob jumped out of his chair. "It's easy! I do it all the time to block my teenager's Counter Strike servers on the router at home! All you need to do is this," he said and proceeded to write on the whiteboard.

```
iptables -A ${CACHE_SERVER_IP} -j DROP
```

"And then after the test, we can undo that with this" he carried on, sensing the growing respect his colleagues were about to kindle in themselves.

```
iptables -D ${CACHE_SERVER_IP} -j DROP
```

They implemented these as part of the setup and teardown of their integration tests, and then confirmed that the older version wasn't working, but the newer one including the fix worked like a charm. Both Alice and Bob changed their job titles to Site Reliability Engineer (SRE) on LinkedIn the same night, and made a pact to never tell anyone they hot-fixed the issue in production.

1.5.4 Chaos engineering in a nutshell

If you've ever worked for a startup, long, coffee-fueled nights like this are probably no stranger to you. Raise your hand if you can relate! Although simplistic, this scenario shows all four steps we covered earlier on in action:

- The *observability* metric is whether or not we can successfully call the API
- The *steady state* is that the API responds a successful response
- The *hypothesis* is that if we drop connectivity to the cache, we continue getting a successful response
- After *running the experiment*, we can confirm that the old version breaks, and the new one works

Well done team, you've just increased confidence in the system surviving difficult conditions! In this scenario, the team was reactive - they only came up with this new test to account for an error their users have already noticed. That made for a more dramatic effect on the plot. In real life, and in the chapters of this book, we're going to do our best to predict and proactively detect this kind of issue without the external stimulus of becoming jobless overnight!

1.6 Summary

- Chaos engineering is a discipline of experimenting on a computer system in order to

uncover problems, often undetected by other testing techniques

- Much like the crash tests done in the automotive industry try to ensure that the car as a whole behaves in a certain way during a well defined, real-life-like event, chaos engineering experiments aim to confirm or refute your hypotheses about the behavior of the system during a real-life-like problem
- Chaos engineering doesn't automatically solve your issues, and coming up with meaningful hypotheses requires a certain level of expertise in how your system actually works
- Chaos engineering isn't about randomly breaking things (although it has its place too), but adding a controlled amount of failure we understand
- Chaos engineering doesn't need to be complicated. The four steps we just covered, along with some good craftsmanship should take you far before things get any more complex. As you will see, computer systems of any size and shape can benefit from chaos engineering

2

First cup of chaos & blast radius

This chapter covers

- **Setting up the VM to run through accompanying code**
- **Basics of Linux forensics - why did my process die?**
- **Your first chaos experiment with a simple bash script**
- **Blast radius**

In the previous chapter you covered what chaos engineering is and what a chaos experiment template looks like. It is now time to get your hands dirty and implement some from scratch! I'm going to take you step by step through building your first chaos experiment, using nothing more than a few lines of bash. I'll also use the occasion to introduce and illustrate new concepts like *blast radius*.

Just one last pit-stop before we're off to our journey - let's set up the workspace.

> **DEFINITIONS** I'll bet you're wondering what a *blast radius* is. Let me explain. Much like an explosive, a software component can go wrong and break other things it connects to. We often speak of a blast radius to describe the maximum of things which can be affected by something going wrong. I'll be teaching you more about it as you read this chapter.

2.1 Setup - working with the code in this book

I care about your learning process. To make sure that all the relevant resources and tools are available to you immediately, I'm providing a virtual machine image that you can download, import and run on any host capable of running VirtualBox (https://www.virtualbox.org/wiki/Downloads). Throughout this book, I'm going to assume you are executing the code provided in the VM. This way you won't have to worry about installing the various tools on your pc. It will also allow us to be more playful inside of the VM than if it was your host OS.

Before you get started, you need to import the virtual machine image into VirtualBox. To do that, you'll complete the following steps:

1. Download the VM image
    a) Go to https://github.com/seeker89/chaos-engineering-book
    b) Click on "releases"
    c) Find the latest release
    d) Follow the release notes to download, verify and decompress the VM archive
2. Install VirtualBox by following instructions at https://www.virtualbox.org/wiki/Downloads
3. Import the VM image into VirtualBox
    a) In VirtualBox, click File->Import Appliance
    b) Pick the VM image file you downloaded and unarchived
    c) Follow the wizard until completion
4. Configure the VM to your taste (and resources)
    a) In VirtualBox, Right click your new VM and choose Settings
    b) In General -> Advanced -> Shared Clipboard check Bidirectional
    c) In System -> Motherboard choose 4096MB of Base Memory
    d) In Display -> Video Memory choose at least 64MB
    e) In Display -> Remote Display uncheck Enable Server
    f) In Display -> Graphics Controller choose what VirtualBox recommends
5. Start the VM and log in
    a) The username and password are both 'chaos'

That's it! The VM comes with all the source code needed and all the tools pre-installed. The versions of the tools will also match the ones I used in the text of this book. All of the source code, including the code used to prebuild the VM, can be found at https://github.com/seeker89/chaos-engineering-book. Once you've completed these steps, you should be able to follow everything in this book. If you find any issues, feel free to create an issue on that GitHub repo. Let's get to the meat of it by introducing an ironically realistic scenario!

2.2 Scenario

Remember our friends from Glanden from the previous chapter? They have just reached out for help. They are having trouble with their latest product: the early clients are complaining it sometimes doesn't work, but when the engineers are testing, it all seems fine. As a growing authority in the chaos engineering community you agree to help them track and fix the issue they are facing. Challenge accepted.

This is a more common scenario than any chaos engineer would like to admit. Something's not working, the existing testing methods don't find anything and the clock is ticking. In the ideal world, we would proactively think about and prevent situations like this,

but in the real world, we'll often face problems when they're already there. To give you the right tools to cope, I wanted to start you off with a scenario of the latter category.

In this kind of situation, we'll typically have at least two pieces of information to work with: the overall architecture and application logs. Let's start by taking a look at the architecture of the FizzBuzz service. As the figure 2.1 illustrates, it consists of a load balancer (`nginx`) and two identical copies of an API server (implemented in Python). When a client makes a request through their internet browser (1) the request is received by the load balancer. The load balancer is configured to route incoming traffic to any instance that's up and running (2). If the instance the load balancer chooses becomes unavailable (3), the load balancer is configured to re-transmit the request to the other instance (4). Finally, the load balancer returns the response provided by the instance of API server to the client (5) and the internal failure is transparent to the user. Please take a look at the figure 2.1.

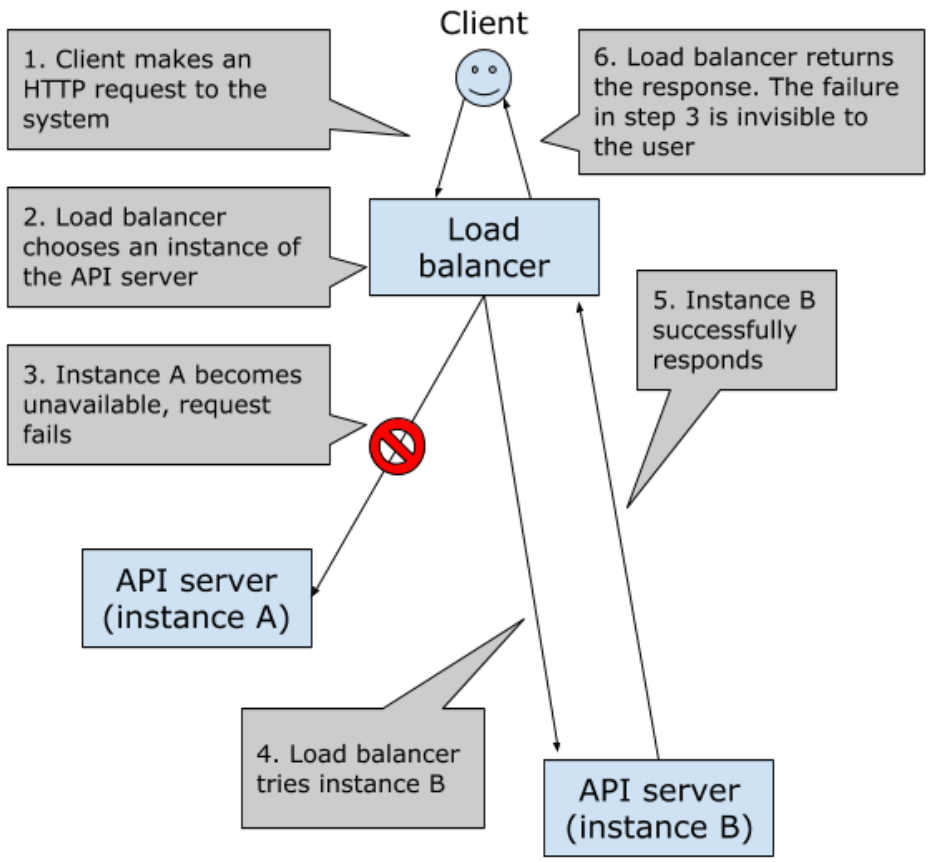


Figure 2.1 FizzBuzz as a Service technical architecture

The other element we have at our disposal is the logs. A relevant sample of the logs looks like this (similar lines appear multiple times):

```
[14658.582809] ERROR: FizzBuzz API instance exiting, exit code 143
[14658.582809] Restarting
[14658.582813] FizzBuzz API version 0.0.7 is up and running.
```

While a little bit light on the details, it does provide us with some valuable clues about what is going on: we can see that their API server instances are restarted and we can also see something called an exit code. These restarts are a good starting point to design a chaos experiment around. But before we do that, it's important that you know how to read exit codes like these and use them to understand what happened to a process before it died. With the Criminal Minds theme in the background, let's take a look at the basics of Linux forensics.

2.3 Linux forensics 101

When doing chaos engineering, we will often find ourselves trying to understand why a program died. It often feels like playing detective solving mysteries in a popular crime tv series. Let's put the detective hat on and solve a case!

In the scenario above, what we have at our disposal amounts to a *black box* program that you can see died, and you want to figure out the *why*. What do you do, and how do you check what happened? In this section we will cover exit codes and killing processes, both manually through the *kill* command and by the Out Of Memory Killer, a part of Linux responsible for killing processes when the system runs low on memory. This will prepare you to deal with processed dying in real life. Let's begin with the exit codes.

> **DEFINITIONS** In software engineering, we often refer to systems which are "opaque" to us (we can only see their inputs and outputs, but have no insight into its inner workings) as *black boxes*. The opposite of a *black box* is also sometimes called a *white box*. You might have heard about the bright orange recording devices installed on airplanes. They are also often referred to as black boxes, because they are designed to prevent tampering with them, despite their real color. When practicing chaos engineering, we will often be able to operate on entire systems or system components as *black boxes*.

2.3.1 Exit codes

When dealing with a black box piece of code, the first thing you might want to think about is running the program and seeing what happens. Unless it's supposed to rotate the nuclear plant access codes, it might be a good idea. To show you what that could look like, I wrote a program which dies for you. Let's warm up by running it and investigating what happens. From the provided VM, open a new bash session and start a mysterious program by running this command:

```
~/src/examples/killer-whiles/mystery000
```

You will notice that it exits immediately and prints an error message like this:

```
Floating point exception (core dumped)
```

It was kind enough to tell us why it died: something to do with floating point arithmetic error. That's great for a human eye, but *Linux* provides a better mechanism of understanding what happened to the program. When a process terminates, it returns a number to inform the user whether it succeeded or not. That number is called an *exit code*. We can check the exit code returned by the preceding command by running the following command at the prompt:

```
echo $?
```

In this case, you will see the following output:

```
136
```

It means that the last program executed exited with code 136. Many (not all) *Unix* commands return 0 when the command was successful, and 1 when it failed. Some will use different return codes to differentiate between different errors. Bash has a fairly compact convention on exit codes[1], that I encourage you to take a look at. The codes in range 128-192 are decoded by 128+n, where n is the number of kill signal. In this example, the exit code is 136, which corresponds to 128 + 8, meaning that the program received a kill signal number 8, which is *SIGFPE.* It is sent to a program when it tries to execute an erroneous arithmetic operation. Don't worry, you don't have to remember all the kill signal numbers by heart. You can see them with their corresponding numbers by running *kill -L* in the command prompt. Note, that some of the exit codes are going to differ between bash and other shells.

Remember that a program can return any exit code, sometimes by mistake. But assuming that it gives us a meaningful exit code, we know where to start debugging and life tends to be good. The program did something wrong, it died, the cold kernel justice was served. But what happens if you suspect it was a murder?

Available signals

If you're curious about the various signals you can send (for example via kill command) you can list them easily by running the following command in your terminal:

```
kill -L
```

You will see output similar to the following one:

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
```

[1] http://www.tldp.org/LDP/abs/html/exitcodes.html

```
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

2.3.2 Killing processes

To show you how to explicitly kill processes, let's play both the good cop and the bad cop. Open two bash sessions in two terminal windows. In the first one, run the following command to start a long-running process:

```
sleep 3600
```

Much like it says on the tin, the *sleep* command just blocks for the specified number of seconds. This is just to simulate a long running process. Your prompt will block waiting for the command to finish. To confirm it's there, in the second terminal, run the following command to list the running processes (the f flag shows visually parent-child relationships between processes):

```
ps f
```

In the following output, you can see `sleep 3600` as a child of the other bash process:

```
  PID TTY      STAT   TIME COMMAND
 4214 pts/1    Ss     0:00 bash
 4262 pts/1    R+     0:00  \_ ps f
 2430 pts/0    Ss     0:00 bash
 4261 pts/0    S+     0:00  \_ sleep 3600
```

Now, still in the second terminal, let's commit a process crime: kill our poor *sleep* process:

```
pkill sleep
```

You will notice the sleep process die in the first terminal. It will print this output and the prompt will become available again:

```
Terminated
```

This is useful to see, but most of the time the processes we care about will die when we're not looking at them, and we'll be interested in gathering as much information about the circumstances of their death as possible. That's when the exit codes we covered before become handy. We can verify what exit code the sleep process returned before dying by using the familiar command:

```
echo $?
```

The exit code is 143. Similar to 136 before, it corresponds to 128 + 15, or *SIGTERM*, the default signal sent by the kill command. This is the same code that was displayed in the

FizzBuzz logs, giving us an indication that their processes were being killed. This is an 'aha' moment: a first piece to our puzzle!

If we chose a different signal, we would see a different exit code. To illustrate that, start the sleep process again from the first terminal by running the same command:

```
sleep 3600
```

To send a `KILL` signal, run the following command from the second terminal:

```
pkill -9 sleep
```

It will result in getting a different exit code. To see what the exit code is, run this command from the first terminal, the one in which the process died:

```
echo $?
```

You will see the following output:

```
137
```

As you might expect, the exit code is 137, or 128 + 9. Note, that there is nothing preventing us from using *kill -8*, and getting the same exit code as in the previous example where we had an arithmetic error in the program. All of this is just a convention, but most of the popular tooling will follow it.

So now you've covered another popular way for a process to die, by an explicit signal. It might be an administrator issuing a command, it might be the system detecting an arithmetic error, or it might be done by some kind of daemon managing the process. Of the latter category, an interesting example is the Out Of Memory Killer. Let's take a look at the mighty OOM Killer.

2.3.3 Out Of Memory (OOM) Killer

The OOM Killer can be a surprise the first time you learn about it. If you haven't yet, I'd like you to experience it firsthand. Let's start with a little mystery to solve. To illustrate what OOM is all about, run the following program I've prepared for you from the command line:

```
~/src/examples/killer-whiles/mystery001
```

Can you find out what the program is doing? Where would you start? The source code is in the same folder as the executable, but stay with me for a few minutes before you read it. Let's try to first approach it as a black box.

After a minute or two of running it you might notice your VM getting a little sluggish, which is a good hint to check the memory utilization. You can see that by running the *top* command from the command line, like the following. Note the use of `-n1` flag to print one output and exit, rather than update continuously, and `-o+%MEM` to sort the processes by their memory utilization.

```
top -n1 -o+%MEM
```

Your output will be similar to the following:

```
top - 21:35:49 up  4:21,  1 user,  load average: 0.55, 0.46, 0.49
Tasks: 175 total,   3 running, 172 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.8 us, 29.4 sy,  0.0 ni, 35.3 id, 20.6 wa,  0.0 hi,  2.9 si,  0.0 st
MiB Mem :   3942.4 total,     98.9 free,   3745.5 used,     98.0 buff/cache          #A
MiB Swap:      0.0 total,      0.0 free,      0.0 used.      5.3 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 5451 chaos     20   0 3017292   2.9g      0 S   0.0  74.7   0:07.95 mystery001      #B
 5375 chaos     20   0 3319204 301960  50504 S  29.4   7.5   0:06.65 gnome-shell
 1458 chaos     20   0  471964 110628  44780 S   0.0   2.7   0:42.32 Xorg
(...)
```

#A Available memory at around 100MB
#B Memory usage (RES and %MEM) and the name of mystery001 process in bold font

We can see that `mystery001` is using 2.9GB of memory, almost three quarters for my VM, and my available memory hovers around 100MB. Your top might start dying on you, or struggle to allocate memory. Unless you're busy encoding videos or maxing out games, that's rarely a good sign. While you're trying to figure out what's going on, if my timing is any good, you should see your process die in the prompt:

```
Killed
```

A murder! But what happened, who killed it? The title of this section is a little bit of a giveway, so let's check the kernel log to look for some clues. To do that, we can use `dmesg`. It's a Linux utility which displays kernel messages. Let's search for our `mystery001` by running the following in a terminal:

```
dmesg | grep -i mystery001
```

You will see something similar to the following output. As you read through these lines, the plot thickens. Something called `oom_reaper` just killed your mysterious process.

```
[14658.582932] Out of memory: Kill process 5451 (mystery001) score 758 or sacrifice child
[14658.582939] Killed process 5451 (mystery001) total-vm:3058268kB, anon-rss:3055776kB,
       file-rss:4kB, shmem-rss:0kB
[14658.644154] oom_reaper: reaped process 5451 (mystery001), now anon-rss:0kB, file-
       rss:0kB, shmem-rss:0kB
```

What is that and why is it claiming rights to your processes? If we browse through `dmesg` a bit more, we will see a little bit of information about what OOM did, including the list of processes it evaluated before sacrificing our program on the altar of RAM.

Here's an example, shortened for brevity. Notice the `oom_score_adj` column, which displays the scores of various processes from the OOM's point of view (I bolded out the name for easier reading):

```
[14658.582809] Tasks state (memory values in pages):
[14658.582809] [  pid  ]   uid  tgid total_vm   rss pgtables_bytes swapents oom_score_adj
       name
(...)
[14658.582912] [   5451]  1000  5451 764567   763945  6164480        0             0
       mystery001
(...)
[14658.582932] Out of memory: Kill process 5451 (mystery001) score 758 or sacrifice child
```

```
[14658.582939] Killed process 5451 (mystery001) total-vm:3058268kB, anon-rss:3055776kB,
       file-rss:4kB, shmem-rss:0kB
[14658.644154] oom_reaper: reaped process 5451 (mystery001), now anon-rss:0kB, file-
       rss:0kB, shmem-rss:0kB
```

OOM Killer is one of the more interesting (and controversial) memory management features in Linux kernel. Under low memory conditions, the OOM Killer kicks in and tries to figure out what processes to kill in order to reclaim some memory and for the system to regain some stability. It uses some heuristics (including `niceness`, how recent the process is and how much memory it uses, see https://linux-mm.org/OOM_Killer for more details) to score each process and pick the unlucky winner. If you're interested in how it came to be and why it was implemented the way it was, the best article on this subject that I know of can be found at https://lwn.net/Articles/317814/.

So, there it is, the third popular reason for processes to die, one which often surprises newcomers. In the FizzBuzz logs sample, we know that the exit code we saw could be a result of either an explicit kill command or perhaps the OOM Killer. Unfortunately, unlike other exit codes which have a well-defined meaning, the one we saw in the logs sample doesn't help us conclude the exact reason for the processes dying. Fortunately, chaos engineering allows us to make progress regardless of that. Let's go ahead and get busy applying some chaos engineering!

OOM Killer settings

The OOM Killer behavior can be tweaked via flags exposed by the kernel. From the kernel documentation, https://www.kernel.org/doc/Documentation/sysctl/vm.txt:

```
==============================================================

oom_kill_allocating_task

This enables or disables killing the OOM-triggering task in
out-of-memory situations.

If this is set to zero, the OOM killer will scan through the entire
tasklist and select a task based on heuristics to kill.  This normally
selects a rogue memory-hogging task that frees up a large amount of
memory when killed.

If this is set to non-zero, the OOM killer simply kills the task that
triggered the out-of-memory condition.  This avoids the expensive
tasklist scan.

If panic_on_oom is selected, it takes precedence over whatever value
is used in oom_kill_allocating_task.

The default value is 0.
```

There is also `oom_dump_tasks` which will dump extra information when killing a process for easier debugging. In the provided VM based off Ubuntu Disco Dingo, we can see both flags defaulting to 0 and 1, respectively, meaning that the OOM Killer will attempt to use its heuristics to pick the victim and then dump extra information when killing processes. If you want to check the settings on your system, you can run the following commands:

```
cat /proc/sys/vm/oom_kill_allocating_task
cat /proc/sys/vm/oom_dump_tasks
```

2.4 The first chaos experiment

The exit codes from the logs didn't give us a good indication of what was causing FizzBuzz's API servers to die. While this might feel like an anticlimax, it is by design. Through that dead end, I wanted to lead you to a powerful aspect of chaos engineering: we work on hypotheses about the entire system as a whole.

As you recall (look at figure 2.2 below for a refresher), the system is designed to handle API server instances dying through load balancing with automatic rerouting if one instance is down. Alas, the users are complaining that they are seeing errors!

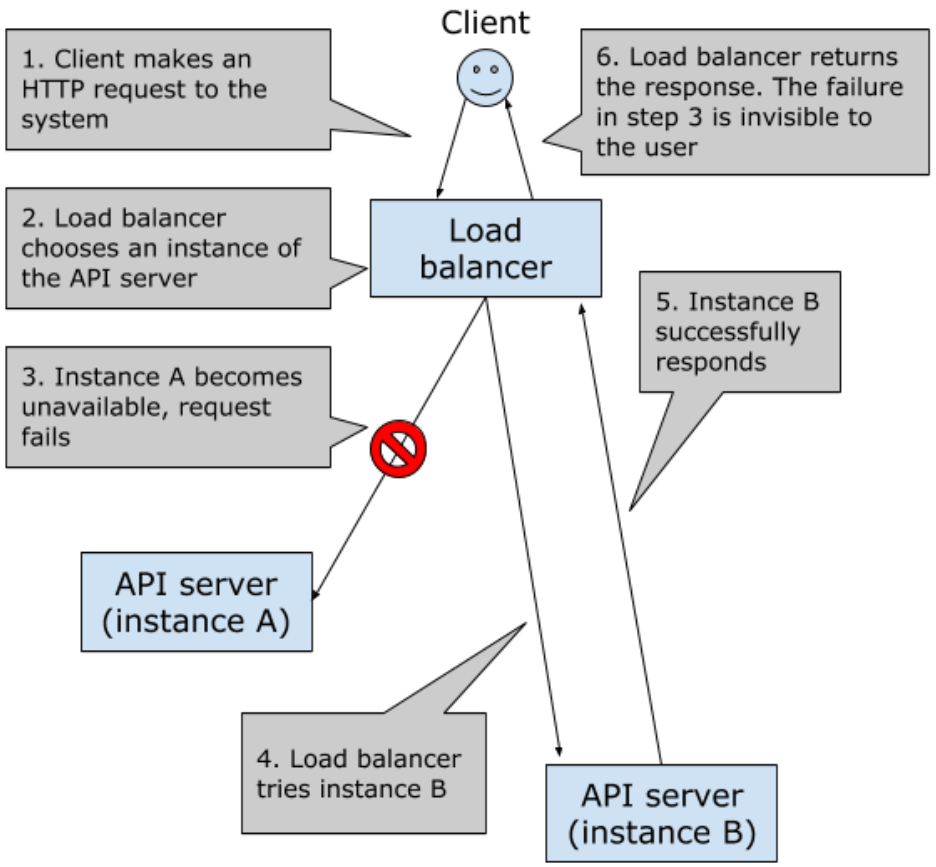


Figure 2.2 FizzBuzz as a Service technical architecture - repeated

While drilling down and fixing the reason why the API server instances get killed is important, from the perspective of the whole system we should be more concerned that the clients are seeing the errors when they shouldn't. In other words, fixing the issue that gets the API server instances killed would 'solve' our problem for now, until another bug, outage or human error reintroduces it, and the end users are impacted. In our system, or any bigger distributed system, components dying is a norm, not an exception.

Please take a look at figure 2.3, which illustrates the difference in thinking about the system's properties as whole, as compared to figure 2.2. The client interacts with the system and just for a minute we stop thinking about the implementation and think about how the system should behave as a single unit.

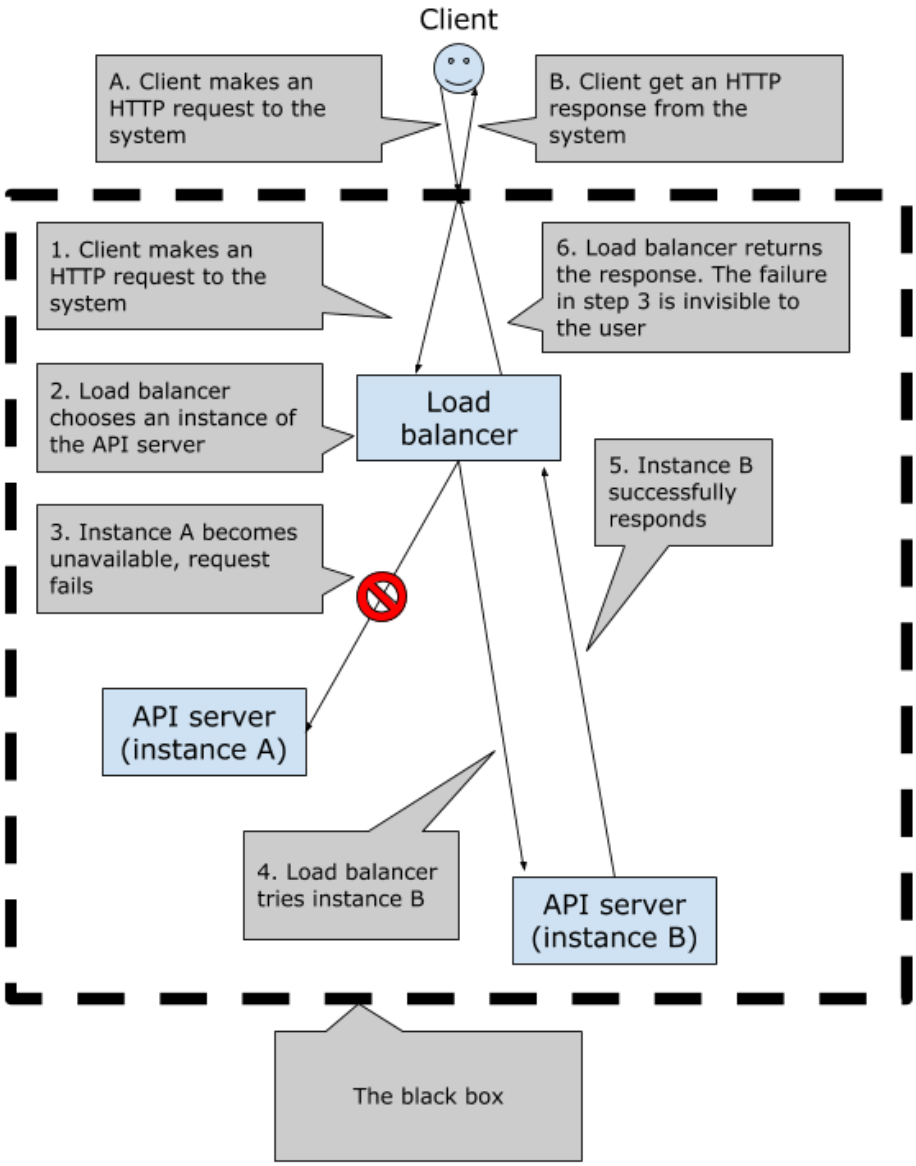


Figure 2.3 FizzBuzz as a Service whole system properties

Let's design our first chaos experiment to replicate the situation our clients are facing and see what happens for ourselves. In the previous chapter you covered the four steps to designing a chaos experiment:

1. Observability
2. Steady state
3. Hypothesis
4. Run the experiment!

It's best to start as simply as possible. We need a metric to work with (observability), preferably one we can produce easily. In this case, let's pick the number of failed HTTP responses that we receive from the system. We could write a script to make some requests and count the failed ones for us, but there are tools which can do that for us out there already. To keep things simple, let's use one that's well known: *Apache Bench*. We can use it to both produce the HTTP traffic for us to validate the steady state and to produce the statistics on the number of error responses encountered in the process. If the system behaves correctly, we should see no error responses, even if we kill an instance of the API server during the test. And that's going to be our hypothesis. Finally, implementing and running the experiment will also be simple, as we've just covered killing processes.

To sum it up, I've prepared figure 2.3 which should look familiar to you. It's the four steps template from chapter 1, figure 1.2 with the details of our first experiment filled in. Please take a look.

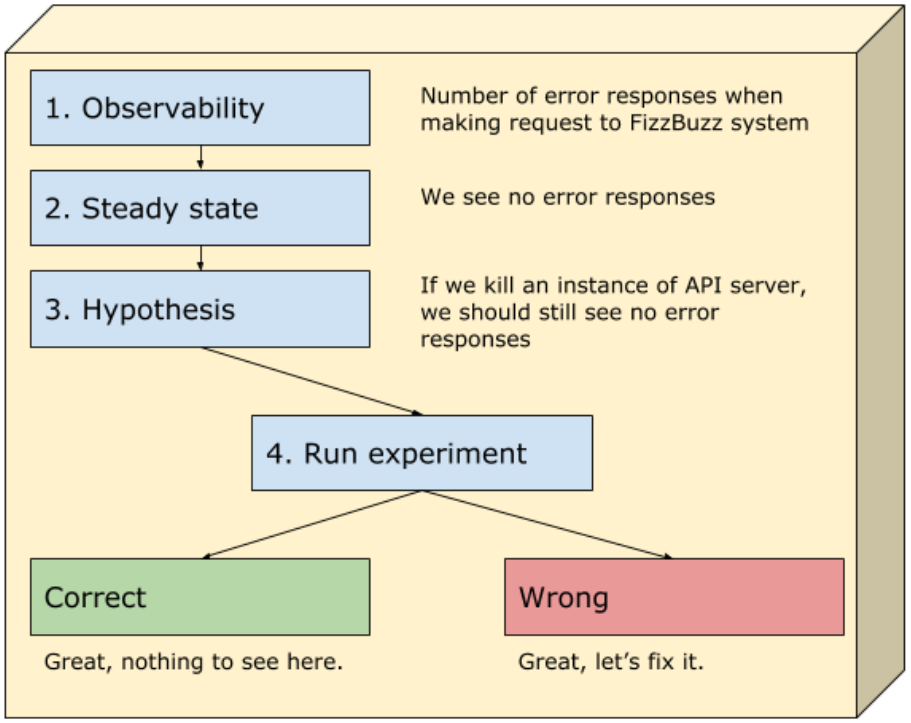


Figure 2.4 The four steps of our first *chaos experiment*

If this sounds like a plan to you, we're on the right track. It's finally time to get your hands dirty! Let's take a closer look at our application. Your VM comes with all the components preinstalled and all the source code can be found in `~/src/examples/killer_whiles` folder. The two instances of the API server are modelled as `systemd` services `faas001_a` and `faas001_b`. They come preinstalled (but disabled by default) so you can use systemctl to check their status. Use the command prompt to run this command for either `faas001_a` or `faas001_b`:

```
sudo systemctl status faas001_a
sudo systemctl status faas001_b
```

The output you'll see will look something like this:

```
● faas001_b.service - FizzBuzz as a Service API prototype - instance A
   Loaded: loaded (/home/chaos/src/examples/killer-whiles/faas001_a.service; static; vendor
       preset: enabled)
   Active: inactive (dead)
```

As you can see, the API server instances are loaded, but inactive. Let's go ahead and start them both via `systemctl` by issuing the following commands in the command line:

```
sudo systemctl start faas001_a
sudo systemctl start faas001_b
```

Note that these are configured to only respond correctly to `/api/v1/` endpoint. All other endpoints will return a 404 response code.

Now, onto the next component - the load balancer. The load balancer is an `nginx` instance, configured to round robin between the two backend instances, and serve on port 8003. It should model the load balancer from our scenario accurately enough. It has a very basic configuration, that you can sneak peek into by issuing this in your command line:

```
cat ~/src/examples/killer-whiles/nginx.loadbalancer.conf | grep -v "#"
```

You will see:

```
    upstream backend {
        server 127.0.0.1:8001 max_fails=1 fail_timeout=1s;
        server 127.0.0.1:8002 max_fails=1 fail_timeout=1s;
    }
    server {
        listen 8003;

        location / {
            proxy_pass http://backend;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
```

Configuring `nginx` and its best practices are beyond the scope of this book, and we won't be needing to know much more than that it should behave like the one we described in the scenario at the beginning of the chapter. The only thing worth mentioning might be the `fail_timeout` parameter set to one second, which means that after one of the servers returns an error (or doesn't respond), it will be taken away from the pool for one second,

and then gracefully reintroduced. `max_fails` configures it to consider a single error response enough to take the instance out of the pool. Nginx is configured to listen on port 8003 on localhost in your VM.

Let's make sure the load balancer is also up and running, by running this in your command prompt:

```
sudo systemctl start nginx
```

To confirm that we can successfully reach the API servers through the load balancer, feel free to use `curl` to reach the load balancer. You can do it by making an HTTP request to localhost, on port 8003, requesting the only implemented endpoint `/api/v1/`. To do that, run the following command in your prompt:

```
curl 127.0.0.1:8003/api/v1/
```

You should see this amazing response:

```
{
    "FizzBuzz": true
}
```

If that's what you've received, we are good to go. If you're tempted to take a look at the source code now, I'm not going to stop you, but I'd recommend holding on for now, and looking a bit later. This way it's easier to think about these components as black boxes with certain behaviors we are interested in. OK, so we're done here, time to make the system do some work by generating some load!

2.4.1 Visibility

There are many ways to generate HTTP load. To keep things simple, let's use Apache Bench, preinstalled and accessible through `ab` command. The usage is straightforward. For example, to run as many requests as we can to our load balancer with concurrency of 10 (`-c 10`) during the period of up to 30 seconds (`-t 30`) or up to 50,000 requests (whichever comes first), while ignoring the content length differences (`-l`) all we need to do is run this command in your prompt:

```
ab -t 30 -c 10 -l http://127.0.0.1:8003/api/v1/
```

The default output of `ab` is pretty informative. The bit of information that we are the most interested in is the `Failed requests` - we will use that as our success metric. Let's go ahead and take a look at what value it has in the steady state.

2.4.2 Steady state

To establish the steady state, or what is the normal behavior, let's execute the `ab` command in your terminal:

```
ab -t 30 -c 10 -l http://127.0.0.1:8003/api/v1/
```

You will see output very similar to the one below. It is a little bit verbose, so I removed the irrelevant parts. As you can see `Failed requests` is 0 and our two API servers are serving the load through the load balancer. The throughput itself is nothing to brag about, but since we're running all of the components in the same VM anyway, we're going to ignore the performance aspect for the time being.

```
(...)
Benchmarking 127.0.0.1 (be patient)
(...)
Concurrency Level:      10
Time taken for tests:   22.927 seconds
Complete requests:      50000
Failed requests:        0
(...)
```

We will use `Failed requests` as our single metric - it is all we need for now to monitor our steady state. Time to write down our hypothesis.

2.4.3 Hypothesis

Like we said before, we expect our system to handle a restart of one of the servers at a time. Our first hypothesis can therefore be written down as "if we kill both instances, one at a time, the users won't receive any error responses from the load balancer". No need to make it any more complex than that, let's go and run it!

2.4.4 Run the experiment

The scene is now set and we can go ahead and implement our very first experiment with some basic `bash` kung-fu. We're going to use `ps` to list the processes we're interested in, and then first `kill` instance A (port 8001), add a small delay, and then `kill` instance B (port 8002), while running `ab` at the same time. I've prepared a simple script for you. Take a look by executing this command in your prompt:

```
cat ~/src/examples/killer-whiles/cereal_killer.sh
```

You will see the following output (shortened for brevity):

```
echo "Killing instance A (port 8001)"
ps auxf | grep 8001 | awk '{system("sudo kill " $2)}'       #A
(...)

echo "Wait some time in-between killings"
sleep 2                                                     #B
(...)

echo "Killing instance B (port 8002)"
ps auxf | grep 8002 | awk '{system("sudo kill " $2)}'       #C
```

#A We search output of ps for a process with string "8001" (faas001_a) in it and kill it
#B We wait 2 seconds to give nginx enough time to detect the instance restarted by systemd
#C We search output of ps for a process with string "8002" (faas001_b) in it and kill it

As you can see, the script first kills one instance, then waits some, and finally kills the other instance. The delay in between killing instances is for `nginx` to have enough time to re-add the killed instance A to the pool before we kill instance B. With that we should be ready to go! We can start the `ab` command in one window, by running:

```
bash ~/src/examples/killer-whiles/run_ab.sh
```

And in another window, we can start killing the instances using the `cereal_killer.sh` script we just looked at. To do that, run this command in your prompt:

```
bash ~/src/examples/killer-whiles/cereal_killer.sh
```

You should see something similar to this (I shortened the output by removing some less relevant bits):

```
Listing backend services
(...)

Killing instance A (port 8001)
● faas001_a.service - FizzBuzz as a Service API prototype - instance A
   Loaded: loaded (/home/chaos/src/examples/killer-whiles/faas001_a.service; static; vendor
       preset: enabled)
   Active: active (running) since Sat 2019-12-28 21:33:00 UTC; 213ms ago
(...)

Wait some time in-between killings

Killing instance B (port 8002)
● faas001_b.service - FizzBuzz as a Service API prototype - instance B
   Loaded: loaded (/home/chaos/src/examples/killer-whiles/faas001_b.service; static; vendor
       preset: enabled)
   Active: active (running) since Sat 2019-12-28 21:33:03 UTC; 260ms ago
(...)

Listing backend services
(...)

Done here!
```

As you can see, both instances are killed and restarted correctly (you can see their PID change, and systemd reports them as active). In the first window, once finished, you should be seeing no errors:

```
Complete requests:      50000
Failed requests:        0
```

Which means that we have successfully confirmed our hypothesis and thus concluded the experiment. Congratulations! You have just designed, implemented and executed your very first chaos experiment. Give yourself a pat on the back!

It looks like our system can survive a succession of two failures of our API server instances. And it was pretty easy to do too. We used `ab` to generate a reliable metric, we established its normal value range and then introduced failure in a very simple bash script. And while the script is simplistic by design, I'm expecting that you thought I was being a

little trigger-happy with the kill command. Which brings me to a new concept called blast radius.

2.5 Blast radius

If you were paying attention, I'm pretty sure you noticed that my previous example `cereal_killer.sh` was a little bit reckless. Take a look at the lines with sudo in them in our `cereal_killer.sh` script by running this command in the prompt:

```
grep sudo ~/src/examples/killer-whiles/cereal_killer.sh
```

You will see these two lines:

```
ps auxf | grep 8001 | awk '{system("sudo kill " $2)}'
ps auxf | grep 8002 | awk '{system("sudo kill " $2)}'
```

It worked fine in the little test, but if there were any processes showing up with a string 8001 or 8002 in the output of `ps`, even just having such a PID, they would be killed. Innocent and with no trial. Not a great look and a tough one to explain to your supervisor at the nuclear power plant. In this particular example, there are many things we could do to fix that, starting from narrowing down our `grep`, to fetching PID from `systemd`, to using `systemctl restart` directly. But I just want you to start keeping this at the back of your head as we go through the rest of the book. To drive the point home, Figure 2.5 illustrates three possible blast radiuses, ranging from a broad grep from the example before to a more specific one, designed to only affect the targeted process.

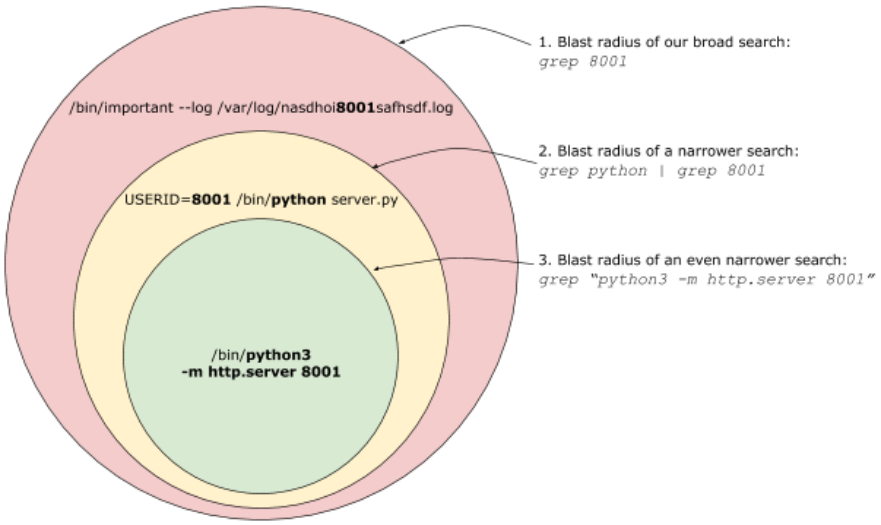


Figure 2.5 Example of blast radiuses

That's what blast radius is all about - limiting the number of things our experiments can affect. We will see different examples of techniques used to limit the blast radius as we cover various scenarios in the following chapters, but in general they fall into two categories: strategic and implementational.

The situation above falls into the latter category of implementational approach. We can proactively look for ways to make the execution safer, but like with any code, we are bound to make mistakes. The former category, strategic, is more about planning our experiments in a way so as to minimize the room for catastrophic events if our experiments go awry. Many good software deployment practices will apply. A few examples include:

- Roll out the experiment on a small subset of your traffic first, and expand later
- Roll out the experiment in a Q&A environment before going to production (we'll talk about testing in production later)
- Automate early, so that you can reproduce your findings more easily
- Be careful with randomness - it's a double-edged sword. It can help find things like race conditions, but it might make things difficult to reproduce (we'll also get back to this a bit later)

All right, so knowing your blast radius is important. For this example, we're not going to change the script, but I'd like you to keep the blast radius at the back of your head from now on. Our first experiment didn't detect any issues, we've patted ourselves on the back, but wait! The FizzBuzz clients are still seeing errors, which indicates that we didn't go deep enough into the rabbit hole. Let's dig deeper!

2.6 Digging deeper

In our first experiment, we have been pretty conservative with our timing, allowing enough time for nginx to re-add the previously killed server to the pool and gracefully start sending it requests. And by 'we have been conservative' I mean to say that I put the sleep there to show you how a seemingly successful experiment might prove insufficient. Let's try to fix that. What would happen if the API server crashed more than once in succession? Would it continue to work?

Let's tweak our chaos experiment, by changing our hypothesis with some concrete numbers: "if we kill an instance A 6 times in a row, spaced out by 1.25s, and then do the same to instance B, we continue seeing no errors". Yes, these numbers are weirdly specific and you're about to see why I picked these in just a second!

I wrote a script that does that for you: it's called `killer_while.sh`. Please take a look at the source code, by running this in your prompt:

```
cat ~/src/examples/killer-whiles/killer_while.sh
```

You will see the body of the script, just like the following. It's essentially a variation of our previous script `cereal_killer.sh` this time wrapped in a couple of while loops. (Yes, I did use while loops instead of for loops so that the killer whiles joke works. Worth it!).

```
# restart instance A a few times, spaced out by 1.25 second delays
i="0"
while [ $i -le 5 ]                                                  # A
```

```
do
    echo "Killing faas001_a ${i}th time"
    ps auxf | grep killer-whiles | grep python | grep 8001 | awk '{system("sudo kill "
       $2)}'                                                                  # B
    sleep 1.25                                                                # C
    i=$[$i+1]
done

systemctl status faas001_a --no-pager                                         # D

(...)
```

#A - we introduce a while loop to repeat the killing 6 times
#B - we do a slightly more conservative series of grep commands to narrow down the target processes, and kill them
#C - we sleep a little bit to give the service enough time to get restarted
#D - display status of the service faas001_a (–no-pager to prevent piping the output to less)

What do you think will happen when you run it? Let's go ahead and find out by running the script in the command prompt like so:

```
bash ~/src/examples/killer-whiles/killer_while.sh
```

You should see output similar to this (again, shortened to show the most interesting bits):

```
Killing faas001_a 0th time
(...)
Killing faas001_a 5th time
● faas001_a.service - FizzBuzz as a Service API prototype - instance A
   Loaded: loaded (/home/chaos/src/examples/killer-whiles/faas001_a.service; static; vendor
       preset: enabled)
   Active: failed (Result: start-limit-hit) since Sat 2019-12-28 22:44:04 UTC; 900ms ago
  Process: 3746 ExecStart=/usr/bin/python3 -m http.server 8001 --directory
       /home/chaos/src/examples/killer-whiles/static (code=killed, signal=TERM)
 Main PID: 3746 (code=killed, signal=TERM)

Dec 28 22:44:04 linux systemd[1]: faas001_a.service: Service RestartSec=100ms expired,
       scheduling restart.
Dec 28 22:44:04 linux systemd[1]: faas001_a.service: Scheduled restart job, restart counter
       is at 6.
Dec 28 22:44:04 linux systemd[1]: Stopped FizzBuzz as a Service API prototype - instance A.
Dec 28 22:44:04 linux systemd[1]: faas001_a.service: Start request repeated too quickly.
Dec 28 22:44:04 linux systemd[1]: faas001_a.service: Failed with result 'start-limit-hit'.
Dec 28 22:44:04 linux systemd[1]: Failed to start FizzBuzz as a Service API prototype -
       instance A.
Killing faas001_b 0th time
(...)
Killing faas001_b 5th time
● faas001_b.service - FizzBuzz as a Service API prototype - instance B
   Loaded: loaded (/home/chaos/src/examples/killer-whiles/faas001_b.service; static; vendor
       preset: enabled)
   Active: failed (Result: start-limit-hit) since Sat 2019-12-28 22:44:12 UTC; 1s ago
  Process: 8864 ExecStart=/usr/bin/python3 -m http.server 8002 --directory
       /home/chaos/src/examples/killer-whiles/static (code=killed, signal=TERM)
 Main PID: 8864 (code=killed, signal=TERM)

(...)
```

Not only do we end up with errors, but both of our instances end up being completely dead. How did that happen? It was restarting just fine a minute ago, what went wrong? Let's double check we didn't mess something up with the `systemd` service file. You can see it by running this command in your prompt:

```
cat ~/src/examples/killer-whiles/faas001_a.service
```

You will see this output:

```
[Unit]
Description=FizzBuzz as a Service API prototype - instance A

[Service]
ExecStart=python3 -m http.server 8001 --directory /home/chaos/src/examples/killer-
      whiles/static
Restart=always
```

The `Restart=always` part sounds like it should always restart, but it clearly isn't. Would you like to take a minute to try to figure it out by yourself? Did you notice any clues in the output above?

2.6.1 Saving the world

As it turns out, the devil's in the details. If you read the logs in the previous section carefully, systemd is complaining about the start request being repeated too quickly. From the systemd documentation[2], we can get some more details:

```
DefaultStartLimitIntervalSec=, DefaultStartLimitBurst=
Configure the default unit start rate limiting, as configured per-service by
      StartLimitIntervalSec= and StartLimitBurst=. See systemd.service(5) for details on
      the per-service settings. DefaultStartLimitIntervalSec= defaults to 10s.
      DefaultStartLimitBurst= defaults to 5.
```

As it turns out, unless `StartLimitIntervalSec` is specified, the default values only allow for five restarts within a 10 second moving window and will stop restarting the service if that's ever exceeded. Which is both good news and bad news. Good news, because we're only two lines away from tweaking the `systemd` unit file to make it always restart. Bad, because once we fix it, the api itself might keep crashing and our friends from Glanden might never fix it, because their clients are no longer complaining!

Let's fix it. Execute the commands below in your prompt to add the extra parameter `StartLimitIntervalSec` set to `0` to the service description:

```
cat >> ~/src/examples/killer-whiles/faas001_a.service <<EOF
[Unit]
StartLimitIntervalSec=0
EOF
```

After that, we will need to reload the systemctl daemon and start the two services again. You can do it by running the following command:

```
sudo systemctl daemon-reload
```

[2] https://www.freedesktop.org/software/systemd/man/systemd-system.conf.html

```
sudo systemctl start faas001_a
sudo systemctl start faas001_b
```

You should now be good to go. With this new parameter, the instance A will be restarted indefinitely, thus surviving repeated errors, while instance B still fails. To test that, we can now run the `killer_while.sh` again by executing the following command in your prompt:

```
bash ~/src/examples/killer-whiles/killer_while.sh
```

You will see output similar to this one (again, shortened for brevity):

```
Killing faas001_a 0th time
(...)
Killing faas001_a 5th time
● faas001_a.service - FizzBuzz as a Service API prototype - instance A
   Loaded: loaded (/home/chaos/src/examples/killer-whiles/faas001_a.service; static; vendor
       preset: enabled)
   Active: active (running) since Sat 2019-12-28 23:16:39 UTC; 197ms ago
(...)
Killing faas001_b 0th time
(...)
Killing faas001_b 5th time
● faas001_b.service - FizzBuzz as a Service API prototype - instance B
   Loaded: loaded (/home/chaos/src/examples/killer-whiles/faas001_b.service; static; vendor
       preset: enabled)
   Active: failed (Result: start-limit-hit) since Sat 2019-12-28 23:16:44 UTC; 383ms ago
  Process: 9347 ExecStart=/usr/bin/python3 -m http.server 8002 --directory
       /home/chaos/src/examples/killer-whiles/static (code=killed, signal=TERM)
 Main PID: 9347 (code=killed, signal=TERM)
(...)
```

As you can see, instance A now survived the restarts and reports as active, but instance B still fails. We made instance A immune to the condition we've discovered. We have successfully fixed the issue!

If you fix *faas001_b* the same way and then rerun the experiment with `killer_while.sh`, you will notice that we no longer see any error responses. The order of the universe is restored and our friends in Glanden can carry on with their lives. We just used chaos engineering to test out the system without looking once into the actual implementation of the API servers, and we found a weakness that's easily fixed. Good job. Now you can pat yourself on the back and I promise not to ruin that feeling for at least 7.5 minutes! Time for the next challenge!

2.7 Summary

- When doing chaos experiments, it's important to be able to observe why a process died - from a crash, a kill signal or from OOM
- Blast radius is the maximum number of things which can be affected by an action or an actor
- Limiting the blast radius consists of techniques which minimize the risk associated with running chaos experiments, and is an important aspect of planning the experiments
- Useful chaos experiments can be implemented with a handful of bash commands, as

illustrated in this chapter, by applying the simple four-step template that we saw in the first chapter

3

Observability

This chapter covers

- Diagnosing system performance issues with the *Utilization, Saturation, Errors* (USE) method
- Understanding basic system metrics we can use in chaos experiments
- Using Linux tools like *top*, *free*, *df*, *vmstat*, *sar* and *Berkeley packet filter* to check system metrics
- Using time series database *prometheus* to gain continuous insight into system performance

Strap in. We're about to tackle one of the more annoying situations you'll face when practicing chaos engineering: the infamous "my app is slow" complaint. If the piece of software in question went through all the stages of development and made it to production, chances are that it passed a decent number of tests and that multiple people signed it off. If later on, for no obvious reason, it begins to slow down it tends to be a sign we're in for a long day at work. "My app is slow" offers much more subtlety than an ordinary "'my app doesn't work" and can sometimes be rather tricky to debug. In this chapter you'll learn how to deal with one of the popular reasons for that - resource contention. We will cover tools necessary to detect and analyze this kind of issue.

There is a thin line between chaos engineering, site reliability engineering (SRE) and system performance engineering in the day to day life. In the ideal world, the job of a chaos engineer would only involve prevention. In practice, we will often need to debug, and then design an experiment to prevent the issue from happening again. Therefore, the purpose of this chapter is to give you just enough tools and background you'll need in the practice of chaos engineering. If I do my job well, by the end of the chapter I expect you to feel comfortable discussing basic Linux performance-analysis tools with that slightly weird uncle

at the next Thanksgiving dinner. Shoot me an email when you do! Let's set the scene with another helping of our dramatic friends from Glanden - the FaaS crowd. What are they up to?

3.1 The app is slow

It was a cold, windy, and exceptionally rainy November afternoon. The clouds were thick and heavy, ripping open with buckets of water pounding on the roof of the yellow cab stuck in traffic in Midtown Manhattan. Alice, the head of engineering (all five of them) at FaaS, trapped inside of the cab was doing some last-minute changes to the presentation she was going to give to a client in a few minutes. She had a bad feeling about that. Since the moment she flew in that day, she had been feeling like something was going to go terribly wrong--the feeling of impending doom. When she stepped out of the car into a wall of water, her phone started ringing. As she looked at her phone, lightning struck. It was the Big Client, accounting for most of their income. They never called with good news. Alice picked up. A cold shiver went down her spine. The wind wrestled away her umbrella and sent it flying away. Alice nodded a few times and hung up. The client said the four words that were about to shake her world: "the app is slow." Cue very loud thunder.

If you've ever had to deal with a weird slowness issue in your system, I'm sure you can relate. They make for good stories and anecdotes later on but at the time they're anything but fun. Depending on the nature of the system, a little bit of slowness might go unnoticed (probably a bad thing anyway), but enough of it means that the system is as good as down. I'm sure you've heard stories about companies and products getting positive media attention only to succumb to the spike in traffic and receive negative coverage for unreliability shortly after. Slowness is dangerous for any business, and we need to be able to diagnose and fight it.

Great, but what does all of this have to do with chaos engineering? Plenty, as it turns out. When practicing chaos engineering, we often either try to actively prevent situations like this from happening (through simulating it and seeing what happens) or we're involved in debugging an ongoing situation and then trying to prevent it from happening again. Either way, in order to wreak havoc responsibly, we need to be able to get a good insight (observability) into the system's performance metrics and fast. Typically, during problems like that, everyone is in panic mode and you need to think quickly. In this chapter I want to give you all the information you're going to need to get started. Let's begin with a high-level overview of the methodology.

3.2 The USE method

Just like skinning a cat, there are many ways to go about debugging a server performance issue. My favorite and the one we're going to cover is called ***USE***, which stands for Utilization, Saturation and Errors (Brendan Gregg http://www.brendangregg.com/usemethod.html). The idea is simple: for each type of resource, check for errors, utilization, and saturation to get a high-level idea of what might be going wrong.

DEFINITIONS In this chapter, we're going to talk a lot about *utilization* and *saturation of resources*. **A resource** is any of the physical components making up a physical server, like CPU, disk, networking devices and RAM. There can also be software resources, like threads, pids, or inode ids. **Utilization** of a resource is an average time or proportion of the resource used. For example, for a CPU, a meaningful metric will be the percentage of time spent doing work. For a disk, the percentage of the disk that is full can be a meaningful metric, but so can its throughput. Finally, **saturation** is the amount of work that the resource can't service at any given moment (often queued). Note, that a high saturation might be a sign of a problem, but it also may very well be desirable (for example in a batch processing system, where we want to use as close to 100% of the available processing power as possible).

This is illustrated in figure 3.1, which shows the flowchart of how to apply the USE method. We start by identifying resources and then for each one of them we check for errors. If found, we investigate and try to fix them. Otherwise, we check the utilization level. If high, we investigate further. Else, we look for saturation, and if that looks problematic, we dig deeper. If we didn't find anything, at least we've reduced the number of unknown unknowns.

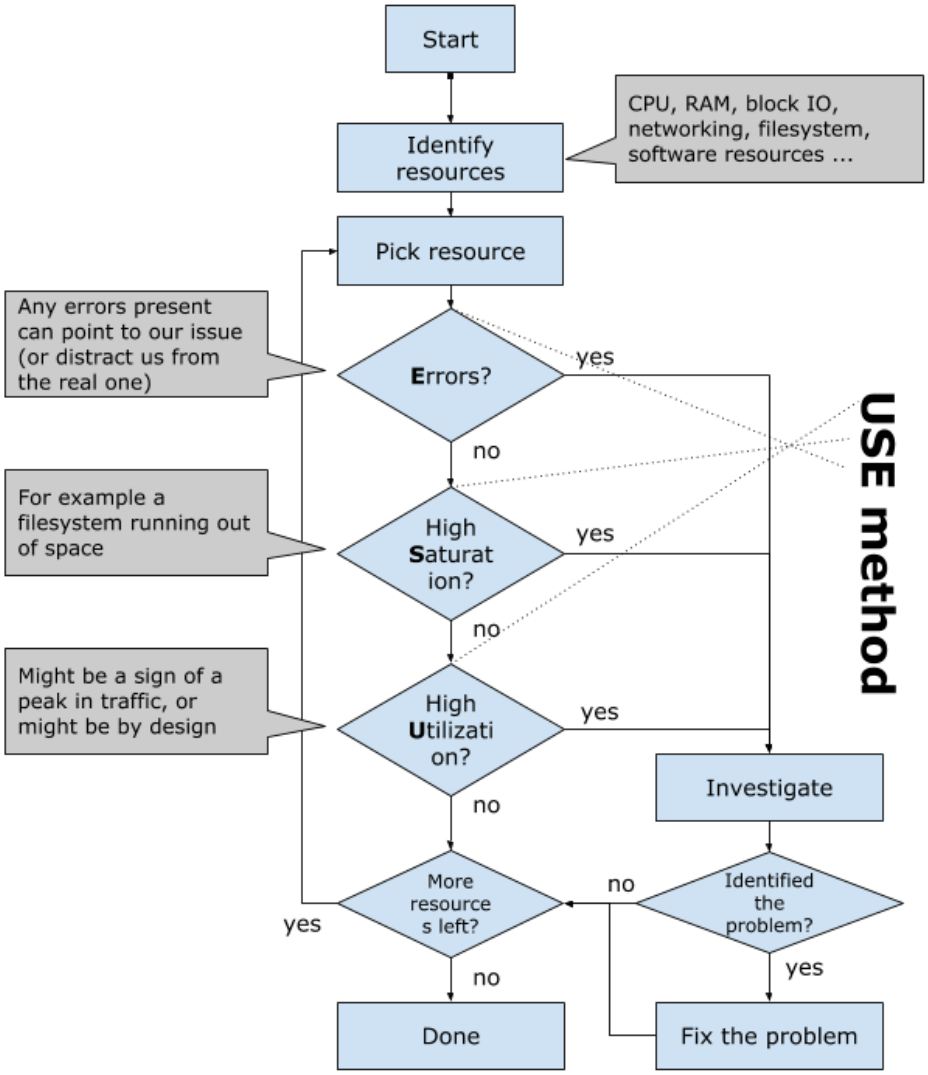


Figure 3.1: Flowchart of the USE method

Known unknowns vs unknown unknowns and the dark debt

Unknowns come in two flavors: known and unknown. Known unknowns are the things we know we don't know. If I haven't opened the fridge, I can't be sure whether there is bacon inside of it (please don't get me started on Schrödinger's bacon or smart fridges with cameras inside). Bacon's already on my radar. But what about things that aren't on my radar, and I don't know I should know? These are the unknown unknowns and every sufficiently complex computer system is going to have some. These are harder to deal with, because usually by the time we realize we need to know about them, it's too late. For instance after an incident, we might come up with some monitoring which

would have alerted us on the problem. That's an unknown unknown becoming a known unknown. Unknown unknowns are also often referred to as the dark debt. If that doesn't sound like something from a galaxy far, far away, then I don't know what does.

This approach lets us quickly identify bottlenecks. It's worth noting that at the various steps of the flowchart we will often find *a problem*, but not necessarily *the problem* causing the issue that prompted us to start the investigation. That is fine; we can add them to the TODO list and carry on with our investigation.

It also needs to be said that books have been (and will be) written about Linux performance observability, and in this chapter my goal is to give you just enough to cover the useful types of issues in the context of chaos engineering. For those who find them new, getting comfortable with these tools will be left for you as an exercise. For those who aren't, please don't hate me for not including your favorite tool! With that asterisk out of the way, let's take a closer look, starting with the resources.

3.3 Resources

I prepared figure 3.2 to illustrate the different types of resources we will be looking into. It's a high- level overview and we'll be zooming in to the different sections later on, but for now I would like you to just take a look at the broad categories of resources we will be working with. At the bottom sit four main logical components of a physical computer: CPU, RAM, networking and block IO. Above, there is a layer of OS, which also provides some software resources (like file descriptors or threads). At the top we have an application layer, in which I included libraries and runtimes. Please take a look at the figure.

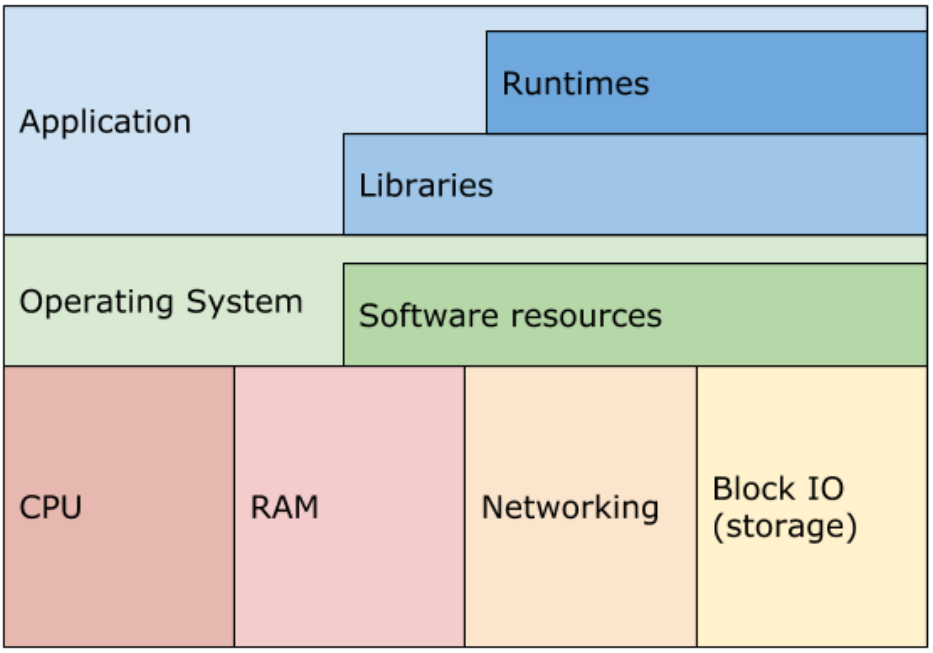


Figure 3.2 A simplified view of system resources

Now, going back to Alice and her terrible day, I would like you to put yourself in her shoes and I'll guide you through the steps to finding out why her application was being slow. We will use the USE method for that.

To make things more fun, I prepared a simulation of Alice's situation for you. All of the commands we'll cover are already available in your VM, and the code snippets are to be executed inside of a terminal in the VM. To start Alice's application in your VM, run the following command in your terminal:

```
~/src/examples/busy-neighbours/mystery002
```

You will see an output similar to the following, calculating Pi's digits in a loop. Notice the time it takes to do one set of calculations (bold font):

```
Press [CTRL+C] to stop..
Calculating pi's 3000 digits...
3.14159265358979323846264338327950288419716939937510582097494459230 7\

real   0m4.183s
user   0m4.124s
sys    0m0.022s
```

It's an approximation of what Alice was seeing when she logged in, running indefinitely.

> **NOTE** This will run your CPUs hot (it's set up to use two cores), so if you're running this on modest hardware and don't want your laptop to work as a heater, you might want to switch it on and off as we go through the tools.

In the rest of the chapter, I'll assume that you have this running in a command line window. When you're done with it, just use CTRL-C to kill it. If you're curious how it works, feel free to have a look, but it would be most fun if you tried to figure it out by letting me walk you through a collection of visibility tools available for Linux.

After the first few iterations, you should notice that the calculations of Pi begin to take much longer, and with more variance in terms of time. This is going to be our "my app is slow" simulation. We'll feel some of Alice's pain.

The following sections each cover a subgroup of tools that you're going to be able to use when trying to gain visibility (observability) of your system, prompted by a slowness of the application. Let's dive in. First stop, "tools that apply to the system as a whole." I know, a pretty lousy name for a bus stop.

3.3.1 System overview

Let's begin by covering two basic tools that give us information about the whole system, *uptime* and *dmesg*. Let's start by looking at *uptime*.

3.3.1.1 UPTIME

U*ptime* is often the first command you're going to run. Apart from telling you how long the system has been up (has it restarted recently), it gives you what's called the *load averages*. Load averages are a quick way of seeing the direction (trend) in which your system is going in terms of load. Run the command `uptime` in your terminal window. You will see an output similar to the following:

```
05:27:47 up 18 min,  1 user,  load average: 2.45, 1.00, 0.43
```

The three numbers represent a moving window sum average of processes competing over CPU time to run over 1, 5, and 15 minutes. The numbers are exponentially scaled, so a number twice as large doesn't mean twice as much load.

In this example, the 1-minute average is 2.45, 5-minute average is 1.00, and 15-minute average is 0.43, which indicates that the amount of load on the system is on the rise. It's an increasing trend. This is really only useful for seeing the direction in which the load is going (increasing or decreasing) but the values themselves don't paint the whole picture by themselves. In fact, don't worry about the values at all. Just remember, that if it's decreasing sharply, it might mean that we're too late and the program eating up all the resources went away. And if it's increasing, it's a nice proxy for the rising load on the system. And that's it really for *uptime*. Let's take a look at dmesg.

Load averages

If you're ever interested in writing a program that uses load averages like the ones printed by *uptime*, you're in for a treat. Linux has you covered. All you need to do is read `/proc/loadavg`. If you print its contents by running the command `cat /proc/loadavg` in the terminal, the output you'll see is similar to this one:

```
0.12 0.91 0.56 1/416 5313
```

The first three numbers are the 1, 5 and 15-minute moving window averages we saw above in *uptime*. The fourth and fifth, separated by a slash are, respectively, the number of currently runnable kernel schedulable entities (process, thread) and the total number of kernel schedulable entities currently existing, respectively. The last number is the PID of the most recently started program. To learn more, just run `man proc` in your terminal and search for *loadavg*.

3.3.1.2 DMESG

Dmesg reads the message buffer of the kernel. Think of it as kernel and driver logs. You can read these logs by running the following command in at your terminal prompt. Because there will be multiple pages of the output, we're piping it to *less* for paging and easy searching

```
dmesg | less
```

What are we looking for? Any errors and anomalies that can give us a clue about what's going on. Do you remember the OOM killer from the previous chapter? You can search for "Kill" in the logs by typing `/Kill` and pressing enter inside of *less*. If your OOM actually killed any processes, you should see an output similar to the following.

```
[14658.582932] Out of memory: Kill process 5451 (mystery001) score 758 or sacrifice child
[14658.582939] Killed process 5451 (mystery001) total-vm:3058268kB, anon-rss:3055776kB,
       file-rss:4kB, shmem-rss:0kB
```

We want to give it a quick glance to ensure that there isn't anything remarkable going on. If there are any error messages, they might or might not be related to what we're diagnosing. If there isn't anything interesting in the logs, we can move on. Dmesg also has a `--human` option, which makes the output slightly easier to read by displaying times in a human-readable format. You can run it with the following line at your command prompt:

```
dmesg --human
```

The output will then have relative times taken by each line, similar to this output (I've shortened the lines for brevity):

```
[Jan28 05:09] Linux version 5.0.0-38-generic (buildd@lgw01-amd64-036)  (...)
[  +0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-5.0.0-38-generic (...)
```

The logs take a little while to get used to, but they are worth giving a quick check every time we want to debug a system performance issue. Don't worry if you're seeing things you don't understand in the logs, the kernel messages are pretty verbose. Most of the time you can ignore anything that doesn't mention 'error'. That's all you need to know about *dmesg* for now. So far so good. Let's segue into the next group of resources - the block IO.

3.3.2 Block IO

Let's take a closer look at block input-output (block IO) devices, such as disks and other types of storage on your system. These have an interesting twist that can affect your system in two ways: they can be underperforming or they can be full. Thus, we'll need to look at their utilization from both of these perspectives: their throughput and their capacity. Figure 3.3 shows you what we are zooming in at, relative to the entire resource map from figure 3.2, including the tools we're going to use to get more information about the utilization and saturation.

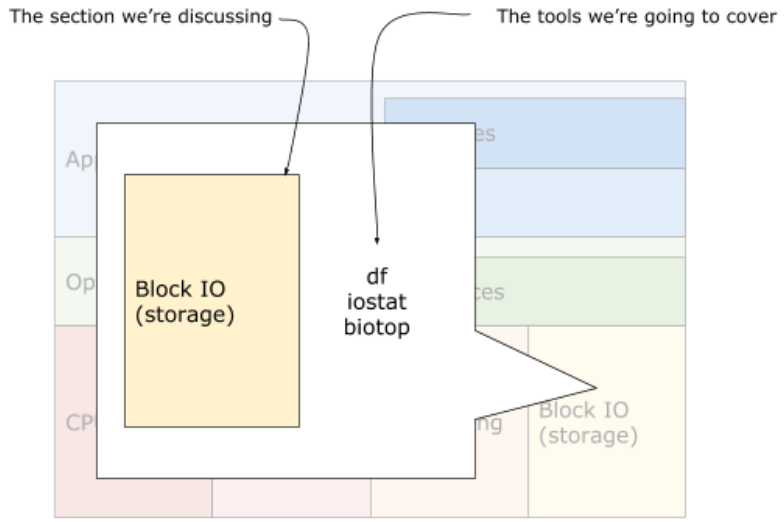


Figure 3.3 Zooming into block IO related observability tools

Let's review some of the tools available for this investigation, starting with *df*.

3.3.2.1 DF

The definition of utilization is the percentage of the resource used. To assess that, we can use *df* which reports filesystem disk space usage. Using it is straightforward: type the following command at your terminal prompt (`-h` here, sneakily, stands for "human readable", not "help") to list all the filesystems mounted and shows their size and used space:

```
df -h
```

You will see an output similar to this one (`/dev/sda1`, my main filesystem, in bold):

```
Filesystem      Size  Used Avail Use% Mounted on
udev            2.0G     0  2.0G   0% /dev
tmpfs           395M  7.9M  387M   2% /run
/dev/sda1        40G   13G   27G  33% /
tmpfs           2.0G     0  2.0G   0% /dev/shm
```

```
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           2.0G     0  2.0G   0% /sys/fs/cgroup
tmpfs           395M   24K  395M   1% /run/user/1000
```

For the device `/dev/sda1`, the utilization is at 33%. When the filesystem gets full, nothing more can be written to it, and it *will* become a problem. But how much data it can hold is just one of the two sides of utilization that a storage device provides. The other is how much it can write in a unit of time - the *throughput*. Let's investigate that part using *iostat*.

3.3.2.2 IOSTAT

Iostat is a great tool for looking into the performance and utilization (in terms of throughput) of block IO devices such as disks. One flag we're going to use is `-x` to get the extended statistics, including percentage of utilization. Run the following command in your terminal:

```
iostat -x
```

You should see an output similar to the following one. In my example, the numbers of reads and writes per second (r/s and w/s respectively) seem reasonable, but by themselves don't say much about what is going on. The fields rkB/s and wkB/s which stand for read and write kilobytes per second, respectively, show the total throughput. Together, the two metrics (raw number and throughput) also give you a feel of an average size of a read or write. Aqu-sz is the average queue length of the requests issued to the device (nothing to do with Aquaman), a measure of saturation, and it shows the system is doing some work. Again, the bare number is hard to interpret, but we can look at whether it's increasing or decreasing. Note that, depending on the host system you run your VM on, you might see very different values. My 2019 Macbook Pro is managing almost 750MB/s, which is comfortably below the values which online benchmarks set it at.

```
Linux 5.0.0-38-generic (linux)     01/28/2020        _x86_64_ (2 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          57.29    0.00   42.71    0.00    0.00    0.00

Device            r/s     w/s     rkB/s     wkB/s   rrqm/s   wrqm/s  %rrqm  %wrqm r_await
       w_await aqu-sz rareq-sz wareq-sz  svctm  %util
loop0            0.00    0.00      0.00      0.00     0.00     0.00   0.00   0.00    0.00
       0.00   0.00     0.00     0.00   0.00   0.00
sda              0.00  817.00      0.00 744492.00     0.00     0.00   0.00   0.00    0.00
       3.44   1.29     0.00   911.25   0.56  46.00
```

Finally, the %util column shows the utilization, here defined as the percentage of time the device spent doing work. A high value *might* indicate saturation, but it's important to remember a couple of things. First, a logical device representing something more complex, like a RAID, might show a high saturation, whereas the underlying disks might actually be underused so be careful when interpreting that. And second, a high saturation doesn't automatically translate into a performance bottleneck in the application, because various techniques are developed to try and do something productive while waiting for IO.

All in all, in my example above, the *iostat* shows some activity writing to my primary disk, but it seems to be comfortably within the range of what it should be able to do at

around 740MB/s writes and 46% utilization. Nothing really suspicious to see here, let's move to the next tool - *biotop*.

3.3.2.3 BIOTOP

Biotop, standing for "block IO top," is part of the suite of tools called BCC (https://github.com/iovisor/bcc) which provides a toolkit for writing kernel-monitoring and tracing programs. It leverages eBPF and provides example utilities to show what you can do with it, very useful in their own right.

Berkeley Packet Filter (BPF)

Berkeley Packet Filter (BPF) is a powerful feature of the Linux kernel, which allows a programmer to execute code inside of the kernel in a way that guarantees safety *and* performance. It allows for a host of different applications, most of which are out of scope of this book, but I strongly recommend you become familiar with it. The BCC project builds on BPF and makes it much easier to work with it by providing wrappers and extra layers of abstraction. The official website of BCC (https://github.com/iovisor/bcc/tree/master/tools) has the source code of all the example applications, including the *biotop*, *opensnoop* and *execsnoop* that we'll cover in this chapter, and many more. The tools themselves are written in a manner making it easy to get started with your own programs. The "e" in eBPF stands for "extended," a more modern version of it, but often simply BPF is used to describe eBPF, and "classic BPF" to talk about the non-extended version.

They are preinstalled on your VM and you can install them from https://github.com/iovisor/bcc/blob/master/INSTALL.md. I would love to show you just how powerful eBPF is and I recommend you get a book or two about it (start with http://www.brendangregg.com/bpf-performance-tools-book.html.) For now, let's just get a taste of a few example tools, starting with *biotop*. On Ubuntu which your VM is running, the tools come appended with -bpfcc. Run *biotop* by typing the following command into your terminal.

```
sudo biotop-bpfcc                  #A
```

#A sudo is required here, because running BPF requires administrator privileges

You should see an output similar to this one, refreshed every second (pro tip: you can add `-C` if you'd like to prevent it from clearing the screen every time):

```
06:49:44 loadavg: 2.70 1.24 0.47 6/426 5269

PID    COMM             D MAJ MIN DISK       I/O  Kbytes  AVGms
5137   kworker/u4:3     W 8   0   sda        677  611272   3.37
246    jbd2/sda1-8      W 8   0   sda          2     204   0.20
```

Biotop helps us identify where the load writing to the disk is coming from. In this case, we can see a process called *kworker*, which is writing more than 600MB/s to the disk and on occasion some other, less hungry processes. We've established that in our case this is fine and we can let it carry on doing its thing. But if we were looking for the culprit eating up all

the resources, this is the tool that will help you with that. Good to remember when you're stressed out![1]

It's also worth noting, that the tools installed by the bpfcc-tool package are written in Python, so if you're curious about what their source code looks like, you can sneak peek directly from your command line by running this command (replace *biotop-bpfcc* with the command you want to investigate) in your terminal:

```
less $(which biotop-bpfcc)
```

All right. So that covers what we're going to need for now in terms of finding out utilization and saturation of the block IO. Let's take a look at the next section - networking!

3.3.3 Networking

Networking in Linux can get pretty complex and my assumption here is that you have an idea of how it works. In this section we're going to focus on establishing the utilization and saturation of the network interfaces and on gaining insight into TCP. Figure 3.4 shows how it fits into our resource map and mentions the tools we're going to look into: *sar* and *tcptop*. Let's start by looking into the network interfaces utilization with *sar*.

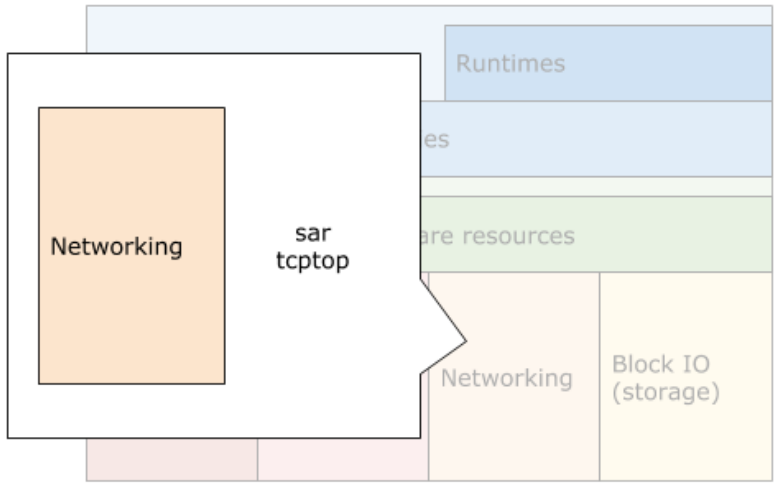


Figure 3.4 Zooming in to network-related observability tools

3.3.3.1 SAR

Sar is a tool to collect, report, and save system metrics. I've preinstalled it into your VM, but in order for it to collect system metrics, we need to activate it. You can do it by editing the file `/etc/default/sysstat` to change `ENABLED="false"` to `ENABLED="true"`. In order for

[1] Brendan Gregg, the author of the BCC project, also maintains a set of graphics about linux tooling you can look into at http://www.brendangregg.com/linuxperf.html They provide a memory aid of what tools you can use when you need to debug a particular part of the system and can be very valuable attached to the wall of your cubicle!

sysstat to pick up the changes, you'll also need to restart its service. You can do it by running the following command at the prompt:

```
sudo service sysstat restart
```

Sar provides various metrics around your system usage, but here we're going to focus on what it offers for the networking. Let's start by checking the utilization. We can use a *DEV* keyword provided by *sar*, which gives us a comprehensive overview of the network interfaces.

Interval and count

Note that *sar*, as well as many tools in the BCC suite, takes two optional, positional parameters at the end: [interval] [count]. They steer how often the output should be printed in seconds (interval) and how many times it should be printed before the program exits (count). Often, the default is one second and infinite count. In our examples, we'll often use 1 1 to just print a single set of stats and exit.

Run the following command in your prompt:

```
sar -n DEV 1 1
```

You should see an output similar to this one (the utilization field and value are in bold font and the output is shortened for easier reading). In this example (what you're likely to see when you run that command in your VM) there is nothing really using networking, so all the stats are at 0. Sar is showing two network interfaces - *eth0* (the main network card) and *lo* (loopback):

```
Linux 5.0.0-38-generic (linux)     01/29/2020        _x86_64_ (2 CPU)

07:15:57 AM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
       %ifutil
07:15:58 AM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
       0.00
07:15:58 AM      eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00
       0.00
(...)
```

The *%ifutil* field is where we can read our utilization from. The other field names are not 100% straightforward, so let me include their definitions here from `man sar`:

- rxpck/s -- Total number of packets received per second.
- txpck/s -- Total number of packets transmitted per second.
- rxkB/s -- Total number of kilobytes received per second.
- txkB/s -- Total number of kilobytes transmitted per second.
- rxcmp/s -- Number of compressed packets received per second (for cslip and so on.).
- txcmp/s -- Number of compressed packets transmitted per second.
- rxmcst/s -- Number of multicast packets received per second.

To generate some traffic, let's download a large file from the internet. You can download an iso image with Ubuntu 19.10 from a relatively slow mirror by running this command from the prompt:

```
wget \
http://mirrors.us.kernel.org/ubuntu-releases/19.10/ubuntu-19.10-desktop-amd64.iso
```

While that is downloading, you can use another terminal window to issue the same *sar* command we did before:

```
sar -n DEV 1 1
```

This time, the output should show the traffic going through on the eth0 interface (again, the utilization in bold font):

```
07:29:44 AM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
       %ifutil
07:29:45 AM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
       0.00
07:29:45 AM      eth0   1823.00    592.00   1616.29     34.69      0.00      0.00      0.00
       1.32
```

Sar also supports another keyword, *EDEV*, to display error statistics on the network. To do that, issue the following command at the prompt:

```
sar -n EDEV 1 1
```

You will see an output similar to the following:

```
Linux 5.0.0-38-generic (linux)     01/29/2020        _x86_64_ (2 CPU)

07:33:53 AM     IFACE   rxerr/s   txerr/s    coll/s  rxdrop/s  txdrop/s  txcarr/s  rxfram/s
       rxfifo/s  txfifo/s
07:33:54 AM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
       0.00      0.00
07:33:54 AM      eth0      0.00      0.00      0.00      0.00      0.00      0.00      0.00
       0.00      0.00
(...)
```

As you can see, there are no errors showing in our example. It doesn't look like Alice's problem lies here.

Again, the field names might seem a little confusing, especially at first, so let me bring the definitions for your convenience:

- rxerr/s
    - Total number of bad packets received per second.
- txerr/s
    - Total number of errors that happened per second while transmitting packets.
- coll/s
    - Number of collisions that happened per second while transmitting packets.

- rxdrop/s
    - Number of received packets dropped per second because of a lack of space in linux buffers.
- txdrop/s
    - Number of transmitted packets dropped per second because of a lack of space in linux buffers.
- txcarr/s
    - Number of carrier-errors that happened per second while transmitting packets.
- rxfram/s
    - Number of frame alignment errors that happened per second on received packets.
- rxfifo/s
    - Number of FIFO overrun errors that happened per second on received packets.
- txfifo/s
    - Number of FIFO overrun errors that happened per second on transmitted packets.

Finally, let's explore another two keywords offered by sar: *TCP* (for TCP statistics) and *ETCP* (for errors in TCP layer). You can run both at the same time by issuing the following command at your prompt:

```
sar -n TCP,ETCP 1 1
```

You will see an output similar to this one. There are no errors showing up, which means that it's not the source of Alice's trouble, not this time at least. We can safely move on to the next tool.

```
Linux 5.0.0-38-generic (linux)     01/29/2020       _x86_64_ (2 CPU)

07:56:30 AM  active/s passive/s    iseg/s    oseg/s
07:56:31 AM      0.00      0.00   1023.00    853.00

07:56:30 AM  atmptf/s  estres/s retrans/s isegerr/s   orsts/s
07:56:31 AM      0.00      0.00      0.00      0.00      0.00

Average:     active/s passive/s    iseg/s    oseg/s
Average:         0.00      0.00   1023.00    853.00

Average:     atmptf/s  estres/s retrans/s isegerr/s   orsts/s
Average:         0.00      0.00      0.00      0.00      0.00
```

Again, for your convenience, the descriptions of the field names:

- active/s
    - The number of times TCP connections have made a direct transition to the SYN-SENT state from the CLOSED state per second [tcpActiveOpens].

- passive/s
    - The number of times TCP connections have made a direct transition to the SYN-RCVD state from the LISTEN state per second [tcpPassiveOpens].
- iseg/s
    - The total number of segments received per second, including those received in error [tcpInSegs]. This count includes segments received on currently established connections.
- oseg/s
    - The total number of segments sent per second, including those on current connections but excluding those containing only retransmitted octets [tcpOutSegs].
- atmptf/s
    - The number of times per second TCP connections have made a direct transition to the CLOSED state from either the SYN-SENT state or the SYN-RCVD state, plus the number of times per second TCP connections have made a direct transition to the LISTEN state from the SYN-RCVD state [tcpAttemptFails].
- estres/s
    - The number of times per second TCP connections have made a direct transition to the CLOSED state from either the ESTABLISHED state or the CLOSE-WAIT state [tcpEstabResets].
- retrans/s
    - The total number of segments retransmitted per second - that is, the number of TCP segments transmitted containing one or more previously transmitted octets [tcpRetransSegs].
- isegerr/s
    - The total number of segments received in error (e.g., bad TCP checksums) per second [tcpInErrs].
- orsts/s
    - The number of TCP segments sent per second containing the RST flag [tcpOutRsts].

If the download hasn't finished, please keep it on for the next section. We'll still need to generate some traffic, while we're looking at *tcptop*!

3.3.3.2 TCPTOP

Tcptop is part of the BCC project I mentioned earlier on (https://github.com/iovisor/bcc). It shows the top (by default, 20) processes using tcp, sorted by bandwidth. You can run it from your command line like this:

```
sudo tcptop-bpfcc 1 1
```

You will see an output similar to the following. RX_KB is the received traffic in kilobytes, TX_KB is the traffic sent ("t" is for transmitted). You can see the *wget* command, slowly downloading the Ubuntu image at just over 2MB/s. We know that it's there, because we ran it on purpose to generate some traffic, but *tcptop* can be an invaluable tool allowing you to track down what's using the bandwidth on the system. Isn't BPF pretty cool?

```
08:05:51 loadavg: 0.20 0.09 0.07 1/415 8210

PID    COMM         LADDR                RADDR                RX_KB  TX_KB
8142   wget         10.0.2.15:60080      149.20.37.36:80       2203      0
```

As you can see, the usage is really simple, and in certain circles might even earn you the title of the local computer magician (computer whisperer?) Make sure you remember about it in times of hardship! OK, that's all we'll need to know about *tcptop* and hopefully enough to get you going to use the USE method on the networking part of the system. Next stop: RAM.

3.3.4 RAM

No program can run without Random Access Memory, and RAM contention is often a problem we're going to have to deal with. It's paramount to be able to read the USE metrics of your system. Figure 3.5 shows where we are on our resource map and the tools we're going to cover: *free*, *top*, *vmstat* and *oomkill*. Let's start with *free*.

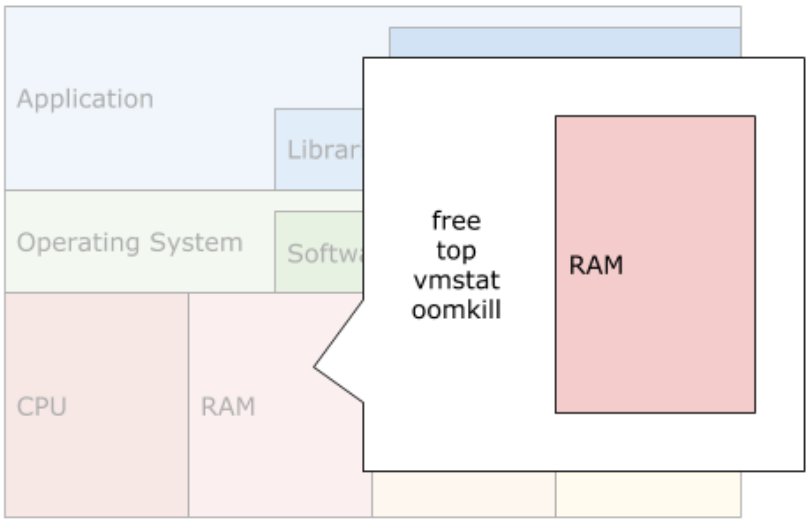


Figure 3.5 Zooming into RAM-related observability tools

3.3.4.1 FREE

Free is an equivalent of *df* for RAM: it shows utilization of RAM. It even accepts the same `-h` argument for human readable output. You can run *free* from your command line like this:

```
free -h
```

You will see the output similar to this one, with the difference that I activated swap here for it to show in the output and yours will be disabled (discussed columns in bold font):

```
               total        used        free      shared  buff/cache   available
Mem:           3.8Gi       1.1Gi       121Mi       107Mi       2.7Gi       2.4Gi
Swap:          750Mi       3.0Mi       747Mi
```

If this is the first time you see this screen, I'd almost bet my breakfast that you'll be confused. If the total memory is 3.8GB and I'm using 1.1GB, then why is only 121MB free? If something smells fishy to you, you're not the only one. In fact, it's such a common reaction, that it has its own website (https://www.linuxatemyram.com/)!

So what's going on? The Linux kernel uses some of the available memory to speed things up for you (by maintaining disk caches), but it's perfectly happy to give it back to you (or any other user) any time you ask for it. So that memory is technically not *free*, but it is indeed *available*. It's an equivalent of your younger brother borrowing your car when you're not using it, except that it always hands it back to you unscathed when you need it. Fortunately, in recent versions of *free*, we have the column "available", just like in the output above. Versions not that long ago didn't have it, and instead provided an extra row called "-/+ buffers/cache," which only added to the confusion. If you are using an older version, you might see an extra row like the following one. It shows the values of *used* minus `buffers`

and `cache` (so used, and can't be reclaimed), as well as `free` plus `buffers` and `cache` (free or can be reclaimed, so available). Also, in that version, `used` used to equal `total` minus `free`.

```
              total        used        free      shared       buffers     cache
Mem:          3.8Gi       2.7Gi       121Mi       107Mi       1.1Gi       1.3Gi
-/+ buffers/cache:       302Mi       2.4Gi
```

So how do you know you've really run out of RAM? The sure-fire giveaways are when the "available" column shows close to zero, and (like we saw before with *dmesg*) the OOM killer goes wild (if active). If the "available" column is showing a reasonable amount left, you're all right. And looking at the output above, it looks like Alice is also all right. Let's move on to the next tool: the good old *top*.

3.3.4.2 TOP

Top gives you an overview of the memory and CPU utilization of your system. Running *top* with default settings is easy. Strike the following three keys at the prompt:

```
top
```

You will see an interactive output refreshing every three seconds, looking something like the one below. Notice that by default the output is sorted by the value of field *%CPU*, which is CPU usage of the program. You can exit by hitting `q` on the keyboard, and again, I'm showing here what it looks like with swap on, yours will be off. I've used a bold font for columns corresponding to CPU utilization (%CPU) and memory utilization (%MEM), as well as the system CPU and memory overview rows.

```
Tasks: 177 total,   6 running, 171 sleeping,   0 stopped,   0 zombie
%Cpu(s): 53.3 us, 40.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  6.7 si,  0.0 st
MiB Mem :   3942.4 total,    687.8 free,   1232.1 used,   2022.5 buff/cache
MiB Swap:    750.5 total,    750.5 free,      0.0 used.   2390.4 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 3508 chaos     20   0  265960 229772    264 R  43.8   5.7   0:02.51 stress
 3510 chaos     20   0    3812     96      0 R  43.8   0.0   0:02.72 stress
 3507 chaos     20   0    3812     96      0 R  37.5   0.0   0:02.63 stress
 3509 chaos     20   0    4716   1372    264 R  37.5   0.0   0:02.43 stress
    7 root      20   0       0      0      0 I  18.8   0.0   0:00.68 kworker/u4:0-flush-8:0
 1385 chaos     20   0  476172 146252  99008 S   6.2   3.6   0:01.95 Xorg
    1 root      20   0   99368  10056   7540 S   0.0   0.2   0:01.38 systemd
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
```

I'm leaving checking the meaning of the other fields to the reader using `man top` as an exercise (which, by the way, is an amazing read; it explains everything from how the memory works, to some stupid tricks you can use to show off at the next team dinner).

You can see that my CPUs are working pretty hard, but more on that in a second. Also, notice the header, which gives you a quick overview of the CPU and memory utilization of the system as a whole. We'll cover what the different values mean in just a little bit, when we talk about CPUs. The memory summary should feel familiar to you, as it's similar to the output of *free* (minus the handy "available" field).

OK. Now, a show of hands of those who have never run *top* in their lives. Exactly, I see no hands up! So why are we even cutting down trees to talk about it? Well, the fun begins when you hit the question mark (`?`) on the keyboard while running *top*. Do it, and you'll see something like this:

```
Help for Interactive Commands - procps-ng 3.3.15
Window 1:Def: Cumulative mode Off.  System: Delay 3.0 secs; Secure mode Off.

  Z,B,E,e   Global: 'Z' colors; 'B' bold; 'E'/'e' summary/task memory scale
  l,t,m     Toggle Summary: 'l' load avg; 't' task/cpu stats; 'm' memory info
  0,1,2,3,I Toggle: '0' zeros; '1/2/3' cpus or numa node views; 'I' Irix mode
  f,F,X     Fields: 'f'/'F' add/remove/order/sort; 'X' increase fixed-width

  L,&,<,> . Locate: 'L'/'&' find/again; Move sort column: '<'/'>' left/right
  R,H,V,J . Toggle: 'R' Sort; 'H' Threads; 'V' Forest view; 'J' Num justify
  c,i,S,j . Toggle: 'c' Cmd name/line; 'i' Idle; 'S' Time; 'j' Str justify
  x,y     . Toggle highlights: 'x' sort field; 'y' running tasks
  z,b     . Toggle: 'z' color/mono; 'b' bold/reverse (only if 'x' or 'y')
  u,U,o,O . Filter by: 'u'/'U' effective/any user; 'o'/'O' other criteria
  n,#,^O  . Set: 'n'/'#' max tasks displayed; Show: Ctrl+'O' other filter(s)
  C,...   . Toggle scroll coordinates msg for: up,down,left,right,home,end

  k,r       Manipulate tasks: 'k' kill; 'r' renice
  d or s    Set update interval
  W,Y       Write configuration file 'W'; Inspect other output 'Y'
  q         Quit
          ( commands shown with '.' require a visible task display window )
Press 'h' or '?' for help with Windows,
Type 'q' or <Esc> to continue
```

It's like accidentally walking into that weird wardrobe taking you to Narnia! Have a look through what that says when you have a minute, but let me highlight a few amazing features you'll love.

> **NOTE** If you're running the Linux VM from a MacOS host, you might be tempted to go and see what the built-in top on MacOS offers by comparison. You will be disappointed, but fortunately there are better alternatives (htop, glances, ...) available through brew and macports.

Toggle memory units: by default, the memory usage is displayed in KB. If you want to toggle it through MB, GB, etc, just press `e` (toggle in the list of processes) or `E` (toggle the summary).

Toggle memory (and cpu) summary: just press `m`, to change the view into progress bars, if you don't fancy comparing numbers in your head. Same works with `t` for toggling CPU usage.

Hide clutter: type `0` (number zero) to hide any zeros on the display.

Change and sort columns: pressing `f` opens a new dialog, in which you can choose which columns to display, rearrange them and choose which one to sort on. The dialog looks like the following output and lists all available options, along with the instructions on how to use them.

```
Fields Management for window 1:Def, whose current sort field is RES
   Navigate with Up/Dn, Right selects for move then <Enter> or Left commits,
   'd' or <Space> toggles display, 's' sets sort.  Use 'q' or <Esc> to end!

* RES     = Resident Size (KiB)    nDRT    = Dirty Pages Count
* PID     = Process Id             WCHAN   = Sleeping in Function
* USER    = Effective User Name    Flags   = Task Flags <sched.h>
* PR      = Priority               CGROUPS = Control Groups
* NI      = Nice Value             SUPGIDS = Supp Groups IDs
* VIRT    = Virtual Image (KiB)    SUPGRPS = Supp Groups Names
* SHR     = Shared Memory (KiB)    TGID    = Thread Group Id
* S       = Process Status         OOMa    = OOMEM Adjustment
* %CPU    = CPU Usage              OOMs    = OOMEM Score current
* %MEM    = Memory Usage (RES)     ENVIRON = Environment vars
* TIME+   = CPU Time, hundredths   vMj     = Major Faults delta
* COMMAND = Command Name/Line      vMn     = Minor Faults delta
  PPID    = Parent Process pid     USED    = Res+Swap Size (KiB)
  UID     = Effective User Id      nsIPC   = IPC namespace Inode
  RUID    = Real User Id           nsMNT   = MNT namespace Inode
  RUSER   = Real User Name         nsNET   = NET namespace Inode
  SUID    = Saved User Id          nsPID   = PID namespace Inode
  SUSER   = Saved User Name        nsUSER  = USER namespace Inode
  GID     = Group Id               nsUTS   = UTS namespace Inode
  GROUP   = Group Name             LXC     = LXC container name
  PGRP    = Process Group Id       RSan    = RES Anonymous (KiB)
  TTY     = Controlling Tty        RSfd    = RES File-based (KiB)
  TPGID   = Tty Process Grp Id     RSlk    = RES Locked (KiB)
  SID     = Session Id             RSsh    = RES Shared (KiB)
  nTH     = Number of Threads      CGNAME  = Control Group name
  P       = Last Used Cpu (SMP)    NU      = Last Used NUMA node
  TIME    = CPU Time
  SWAP    = Swapped Size (KiB)
  CODE    = Code Size (KiB)
  DATA    = Data+Stack (KiB)
  nMaj    = Major Page Faults
  nMin    = Minor Page Faults
```

Note, that from the main screen of *top*, you can also change which column is used for sorting using the `<` and `>` keys, but it's a little bit awkward, because there is no visual indication next to the column name. You can use `x` to toggle the sorted column to be in bold font, which helps with that.

Search (locate) a process name: press `L` to open a search dialog.

Show forest view: much like `ps f`, it shows which processes are children of which parents when you press `V`.

Save the view: you can write the configuration file by pressing `w`. This can be a real time saver. Once you press `w`, `top` will write with all the interactive settings you've changed so that the next time you run it, it can pick up the same settings.

OK, so that might feel a little bit off topic, but it really isn't. When practicing chaos engineering, I can't stress enough how important it is to understand your metrics and how to read them reliably. *Top* is both powerful *and* actually pleasant to use, and it's crucial you can

use it efficiently. If you've used *top* for years, but still learned something new about it from the section above, shoot me an email!

In the initial output we looked at the memory utilization and saturation were pretty low, which indicates that it's not what we're looking for, so let's move on to the next tool. We'll get back to the busy CPUs in just a few moments. Next in line is *vmstat*.

3.3.4.2 VMSTAT

Vmstat shows much more than just the virtual memory statistics its name implies. Please run the *vmstat* command first without any arguments in your command prompt:

```
vmstat
```

You will see an output similar to the following.

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 5  0      0 1242808  47304 1643184    0    0  1866 53616  564  928 17 13 69  1  0
```

The values fit in a single row, which makes it practical to print them every n seconds (with `vmstat n`). The interesting columns include the memory (similar to *free*, with `swpd` showing used swap memory), `r` (the number of runnable processes, running or waiting to run) and `b` (the number of processes in uninterruptible sleep). The number of runnable processes gives as indication of the saturation of the system (the more processes competing for run time, the busier the system is - remember the load averages earlier in the chapter?). The columns `in` and `cs` stand for the total number of interrupts and context switches, respectively. We'll cover the breakdown of the CPU time in the section on CPU very soon.

As you can see, there is an overlap with the other tools like *free* and *top*. Between tools showing the same information, picking the one to use is largely a personal preference. But to help you make an informed decision, here's a few other things that *vmstat* can do for you.

Generate readable system usage stats. If you run *vmstat* with the `-s` flag in your prompt, like this:

```
vmstat -s
```

You will be presented a nicely readable list, just like the following:

```
      4037032 K total memory
      1134620 K used memory
       679320 K active memory
      1149236 K inactive memory
      2049752 K free memory
        17144 K buffer memory
       835516 K swap cache
       768476 K total swap
            0 K used swap
       768476 K free swap
        54159 non-nice user cpu ticks
          630 nice user cpu ticks
        45166 system cpu ticks
        25524 idle cpu ticks
          337 IO-wait cpu ticks
```

```
           0 IRQ cpu ticks
        3870 softirq cpu ticks
           0 stolen cpu ticks
     1010446 pages paged in
   255820616 pages paged out
           0 pages swapped in
           0 pages swapped out
     1363878 interrupts
     1140588 CPU context switches
  1580450660 boot time
        3541 forks
```

Notice the last row - `forks`. It's the number of forks executed since the boot - basically the total number of processes that have run. It's yet another indication of the busyness of the system. You can even get just that piece of information by running `vmstat -f` directly.

Generate readable disk usage stats. If you run `vmstat -d`, you will be presented with utilization/saturation statistics for the disks in your system. You can also run `vmstat -D` to get a one-off summary.

OK, enough about *vmstat*. Let's cover one last utility in the RAM department - the *oomkill*.

3.3.4.3 OOMKILL

Oomkill (part of the BCC project - https://github.com/iovisor/bcc) works by tracing kernel calls[2] to `oom_kill_process` and printing to the screen information about it every time it happens. Do you remember when we covered looking through *dmesg* output, searching for information about processes being killed by the OOM Killer? Well, this is an equivalent of plugging directly into the Matrix - we get the information from the source, and we can plug it into whatever system we are looking at.

To run *oomkill*, execute the following command in one terminal window:

```
sudo oomkill-bpfcc
```

It will start tracing OOM kills. We're now well-equipped for dealing with the kind of situation when a process gets killed by the OOM. If you open another terminal window and run top in it, this time with -d 0.5 to refresh every half a second:

```
top -d 0.5
```

You can press `m` a couple of times to get a nice progress bar showing the memory utilization of the system. Now, for the big finale: in a third terminal window try to eat all the memory using Perl (this actually comes directly from https://github.com/iovisor/bcc/blob/master/tools/oomkill_example.txt):

```
perl -e 'while (1) { $a .= "A" x 1024; }'
```

You should see *top* show more and more memory usage for a few seconds, and then go back to the previous state. In the first window with the *oomkill*, you should see the trace of the assassin:

[2] Look how simple BPF and BCC make it to attach a probe like that: https://github.com/iovisor/bcc/blob/0e63c5c1ad267281f8d0b652eaf87dd494ddba04/tools/oomkill.py#L38 Isn't that amazing?

```
06:49:11 Triggered by PID 3968 ("perl"), OOM kill of PID 3968 ("perl"), 1009258 pages,
       loadavg: 0.00 0.23 1.22 3/424 3987
```

Pretty neat, isn't it? If that was too quick, do you remember the *mystery001* program you were debugging in the previous chapter? We can revisit that by running the following in the third terminal window:

```
./src/examples/killer-whiles/mystery001
```

The memory usage bar in *top* should now be slowly creeping up and in under a minute, you should see the *oomkill* print another message, similar to the following:

```
07:09:20 Triggered by PID 4043 ("mystery001"), OOM kill of PID 4043 ("mystery001"), 1009258
       pages, loadavg: 0.22 0.10 0.36 4/405 4043
```

Sweet. Now you're fully armed to deal with OOM kills and read RAM utilization and saturation. Well done, detective. Time to move on to the next resource - the CPU.

3.3.5 CPU

Time to talk about the workhorse of all the system resources - the CPU! Let's take a minute to appreciate all the hard work the processor is doing for us. I'm running my VM with two cores of my 2019 Macbook Pro. Let's sneak peek at what my Ubuntu sees about the processors, by running the following in your command prompt:

```
cat /proc/cpuinfo
```

You will see an output similar to this one (I removed most of it for brevity), where you can see the details of each processor, including the model and the CPU clock:

```
processor       : 0
(...)
model name      : Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
stepping : 10
cpu MHz         : 2591.998
(...)
```

So during that minute we were appreciating its hard work, each of the two cores did about 2591.998 million cycles * 60 seconds in a minute = ~166 billion cycles total. If only our politicians were that hard working! Now, what were they busy doing all that time? In this section, we'll take a look at that.

Figure 3.6 zooms in on our resource graph to show you where we are, and lists the tools we're going to cover in this section: *top* and *mpstat*. We've already covered the memory-related aspects of *top*, so let's start by finishing that one off by covering what it has to offer in the context of CPU!

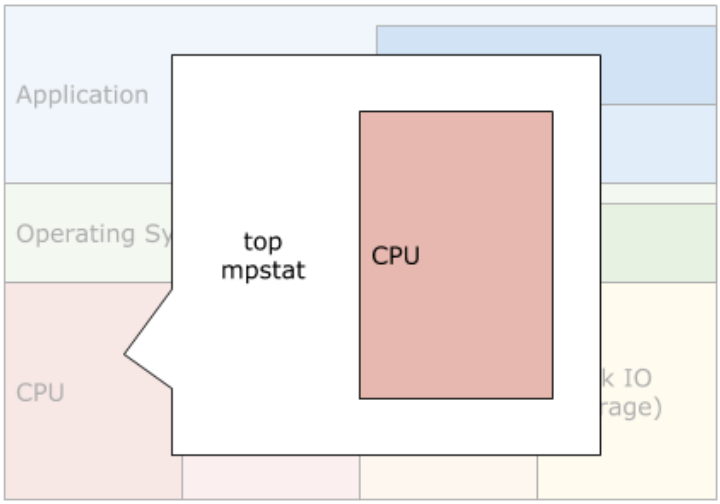


Figure 3.6 Zooming into CPU observability tools

3.3.5.1 TOP

By now you're familiar with how to read *top*'s memory usage, and how to use a few very cool features (if you're not, go back to the section on RAM, earlier in this chapter). Let's finally cover what a processor spends its time doing. Run *top* again from your terminal:

```
top
```

You will see an output similar to the following. This time, let's focus on the `%Cpu(s)` row (bold font).

```
Tasks: 177 total,   6 running, 171 sleeping,   0 stopped,   0 zombie
%Cpu(s): 71.9 us, 25.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  3.1 si,  0.0 st
```

What do all these numbers mean? Let's take a look.

- **us (user time)**: the percentage of time the CPU spent in user space,
- **sy (system time)**: the percentage of the time the CPU spent in kernel space,
- **ni (nice time)**: the percentage of the time spent on low priority processes,
- **id (idle time)**: the percentage of time the CPU spent doing literally nothing (it can't stop!),
- **wa (io wait time)**: the percentage of time the CPU spent waiting on IO,
- **hi (hardware interrupts)**: the percentage of time the CPU spent servicing hardware interrupts,
- **si (software interrupts)**: the percentage of time the CPU spent servicing software interrupts,
- **st (steal time)**: the percentage of time a hypervisor stole the CPU to give it to someone else. Only kicks in in virtualized environments.

Niceness

Niceness is an interesting concept in Linux. It's a numeric value, which shows how happy a process is to give out some CPU cycles to more high priority neighbors (how *nice* it is with others). Allowed values range from -20 to 19. A higher value means nicer, so happier to give CPU away (and so lower priority). Lower value means higher priority. See `man nice` and `man renice` for more info. These are the values you can see in the `NI` column in *top*.

In the output above, you can see that we spend a majority of the time in user space (presumably running Alice's application), 25% in kernel space (probably executing syscalls) and the remainder in software interrupts (most likely handling the syscalls invocations). There is no idle time at all, which means that whatever Alice's application is doing, it's using up all of the available CPU!

Now, if we take a look at the rest of the output of the top command, you will see something similar to this (again, I've removed some output for brevity):

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 2893 chaos     20   0    3812    100      0 R  52.9   0.0   0:02.57 stress
 2894 chaos     20   0  265960 183156    324 R  23.5   4.5   0:02.60 stress
 2895 chaos     20   0    4712   1376    264 R  23.5   0.0   0:02.62 stress
 2896 chaos     20   0    3812    100      0 R  17.6   0.0   0:02.64 stress
 2902 chaos     20   0    3168   2000   1740 R  17.6   0.0   0:01.90 bc

(...)
```

We can see the top commands taking up pretty much all of the available two cores: *stress* and *bc*. It's now a good time to look under the covers of the simulation we've been investigating. Run this in your terminal to look at the *mystery002* command you've been running:

```
cat ~/src/examples/busy-neighbours/mystery002
```

You will see this output, which is a simple bash script, calculating pi's digits and running some totally benign script in the background.

```
#!/bin/bash
echo "Press [CTRL+C] to stop.."

# start some completely benign background daemon to do some __lightweight__work
#    ^ this simulates Alice's server's environment
export dir=$(dirname "$(readlink -f "$0")")
(bash $dir/benign.sh)&                                                          #A

# do the actual work
while :
do
    echo "Calculating pi's 3000 digits..."
    time echo "scale=3000; 4*a(1)" | bc -l | head -n1                           #B
done
```

#A this is the background process being started
#B here's our bc command, creatively used to calculate pi's digits!

So far so good, but let's double check how benign that background process really is by running this command in your prompt:

```
cat ~/src/examples/busy-neighbours/benign.sh
```

You will see the following output:

```
#!/bin/bash

# sleep a little, sneakily
sleep 20

# Just doing some lightweight background work
# Nothing to see here ;)
while :
do
    stress --cpu 2 -m 1 -d 1 --timeout 30 2>&1 > /dev/null                          #A
    sleep 5
done
```

#A here's the stress command we were seeing in `top`!

There we go. Here's our problem - our app (*bc* command calculating pi) was competing for CPU time with the other commands in the system (stress), and as you can see in the output of `top` earlier in this section, it wasn't always winning (`stress` command tended to get more %CPU allocated). For your convenience, let me repeat that output.

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 2893 chaos     20   0    3812    100      0 R  52.9   0.0   0:02.57 stress
 2894 chaos     20   0  265960 183156    324 R  23.5   4.5   0:02.60 stress
 2895 chaos     20   0    4712   1376    264 R  23.5   0.0   0:02.62 stress
 2896 chaos     20   0    3812    100      0 R  17.6   0.0   0:02.64 stress
 2902 chaos     20   0    3168   2000   1740 R  17.6   0.0   0:01.90 bc

(...)
```

This is the source of our perceived slowness, your honor. The stress processes were a classic case of a busy neighbor. Case closed. Another mystery solved. Well done!

There should now be no *top* secrets any more, and you're basically a certified *top* agent (I really couldn't help it). Hopefully your computer hasn't overheated yet, but I do expect that your room temperature has risen since you started the chapter. We'll talk about how to deal with resource starvation and busy neighbors just a little bit later. Let's just cover the last tool really quickly and we're done with the CPU stuff!

3.3.5.2 MPSTAT -P ALL 1

Mpstat is another tool which can show you the CPU utilization. The nice thing about it is that it can show you each CPU separately. Run the following in a terminal:

```
mpstat -P ALL 1
```

It will display an output similar to this one, printed every second:

```
01:14:08 PM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice
       %idle
01:14:09 PM  all   60.10    0.00   33.33    0.00    0.00    6.57    0.00    0.00    0.00
```

```
       0.00
01:14:09 PM    0   41.41    0.00   45.45    0.00    0.00   13.13    0.00    0.00    0.00
       0.00
01:14:09 PM    1   78.79    0.00   21.21    0.00    0.00    0.00    0.00    0.00    0.00
       0.00
```

The same statistics we were looking at with top are visible here, but now split separately for each CPU. What's useful about it, is that this way you can see the distribution of load and analyze it. If you'd like to kill and restart the *mystery002* process from the beginning of the chapter, you should see that for the first 20 seconds, the bc command is allowed to take as much CPU as it wants, but since it's single-threaded, it's only scheduled on a single CPU anyway. And then, after the initial 20 seconds, when the *stress* command starts running, it creates workers for both CPUs, and both of them become busy.

I like the output of *mpstat*, because the columns are more readable (nothing to do with the fact that it starts with my initials!). If you don't have *mpstat* available on your system, *top* also supports a split view similar to this one and in the version available on our Ubuntu VM, you can toggle it by pressing `1` (number one).

All right, so that puts *mpstat* on your radar. You have now more than enough tooling to detect what's going on, and see when someone is eating up the CPU time you were hoping to get yourself. So, the question now becomes - how do you prevent that from happening? Let's take a look at your options.

3.3.5.3 My dog ate my CPU - how do I fix it?

We found out that the app was being slow for the simple reason of it not getting enough CPU. This might sound pretty basic in our simulated environment, but in a larger, shared environment it might be everything but obvious. A lot of very serious production problems aren't rocket science, and that's fine.

I would like you to recall the four steps of the chaos experiment from the first chapter: observability, steady state, hypothesis and running the experiment. Let's look at what we have done so far:

1. We observed the times needed for our program to calculate the 3000 digits of Pi - **our metric**
2. We saw that initially an iteration was taking a certain time - **our steady state**
3. We expected that that time per iteration remains the same - **our hypothesis**
4. But when we ran the experiment, the times were larger - **we were wrong**

Look ma, no hands! We've just applied what we learned in the previous chapters, and conducted a reasonable chaos experiment. Take a look at figure 3.7 which sums all of it up in a format you should now be familiar with.

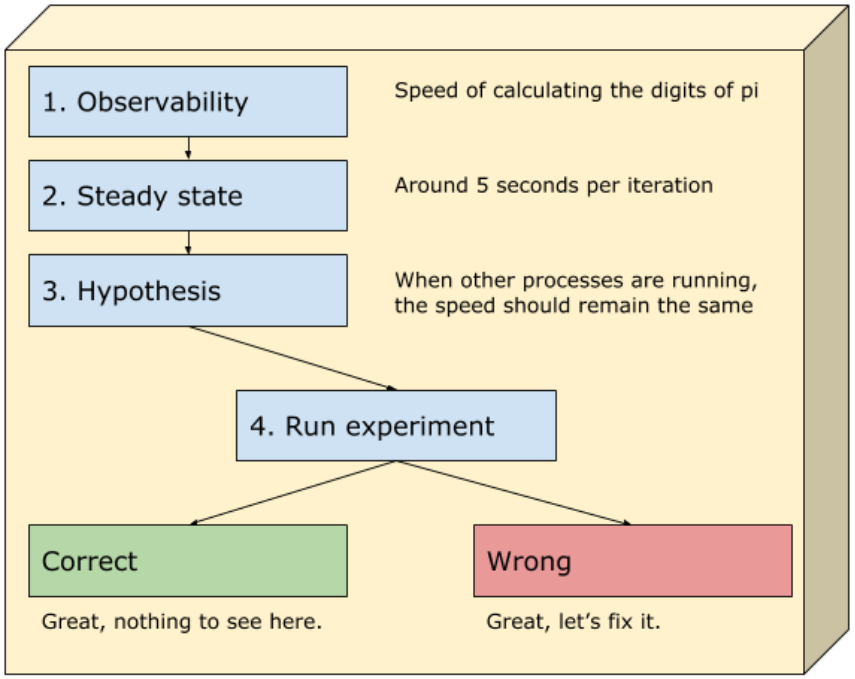


Figure 3.7 The four steps of our second *chaos experiment*

Our experiment showed that the hypothesis was wrong - when the background processes were running, our application was slowing down considerably. How can we go about fixing it? One option would be to use *niceness*, a property we saw earlier in the chapter, which allows us to set a higher relative priority for our process compared to other processes on the system to ensure it gets more CPU time. This could work, but it has one major drawback - it's hard to control precisely how much CPU they would get.

Linux offers another tool we can use in this situation - control groups. Control groups are a feature in Linux kernel, which allows the user to specify exact amounts of resources (CPU, memory, IO) that the kernel should allocate to a group of processes. We will play with them a fair bit in the chapter about Docker, but for now I wanted to give you a quick taste of what it can do.

Let's start by using cgcreate to create two control groups: *formulaone* and *formulatwo*. You can do it by running these commands in your prompt:

```
sudo cgcreate -g cpu:/formulaone
sudo cgcreate -g cpu:/formulatwo
```

Think of them as... Tupperware (oh my, was I just about to say "containers"?), in which you can put processes and have them share that space. We can put a process in one of these lunchboxes by starting it with *cgexec*. Let's tweak our initial *mystery002* script to use

cgcreate and *cgexec*. I've included a modified version for you. You can see it by running this command in your prompt:

```
cat ~/src/examples/busy-neighbours/mystery002-cgroups.sh
```

You will see this output (the modified parts in bold font):

```
#!/bin/bash
echo "Press [CTRL+C] to stop.."

sudo cgcreate -g cpu:/formulaone                                    #A
sudo cgcreate -g cpu:/formulatwo

# start some completely benign background daemon to do some __lightweight__work
#    ^ this simulates Alice's server's environment
export dir=$(dirname "$(readlink -f "$0")")
(sudo cgexec -g cpu:/formulatwo bash $dir/benign.sh)&               #B

# do the actual work
while :
do
    echo "Calculating pi's 3000 digits..."
    sudo cgexec -g cpu:/formulaone bash -c 'time echo "scale=3000; 4*a(1)" | bc -l | head -
       n1' #C
done
```

#A we create the cpu-controlled control groups
#B we execute the benign.sh script in its own control group
#C we execute the main, pi-digit-calculating code in a separate control group

By default, each control group gets 1024 shares, or 1 core. You can confirm that it works yourself, by running the new version of the script in one terminal:

```
~/src/examples/busy-neighbours/mystery002-cgroups.sh
```

And in another terminal, running `top`. You should see an output like the following, in which all the *stress* processes are sharing roughly one CPU, while the *bc* process is able to use another CPU.

```
Tasks: 187 total,   7 running, 180 sleeping,   0 stopped,   0 zombie
%Cpu(s): 72.7 us, 27.3 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3942.4 total,    494.8 free,   1196.3 used,   2251.3 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2560.1 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 4888 chaos     20   0    3168   2132   1872 R  80.0   0.1   0:03.04 bc
 4823 root      20   0    3812    100      0 R  26.7   0.0   0:06.05 stress
 4824 root      20   0  265960 221860    268 R  26.7   5.5   0:06.13 stress
 4825 root      20   0    4712   1380    268 R  26.7   0.0   0:05.97 stress
 4826 root      20   0    3812    100      0 R  26.7   0.0   0:06.10 stress
```

We will look into that much more in the later chapters. If you're curious to know now, just run `man cgroups` in the terminal. Otherwise, we're done with the CPUs for now. Let's take a step up our resource map, and visit the OS layer.

3.3.6 OS

We've already solved Alice's mystery with her app being slow, but before we go, I wanted to give you a few really powerful tools at the OS level. You know, for the next time the app is slow, but the CPU is not the issue. Figure 3.8 shows where that fits on our resource map. The tools we'll take a look at are *opensnoop* and *execsnoop*, both coming from the BCC project. Let's start with *opensnoop*.

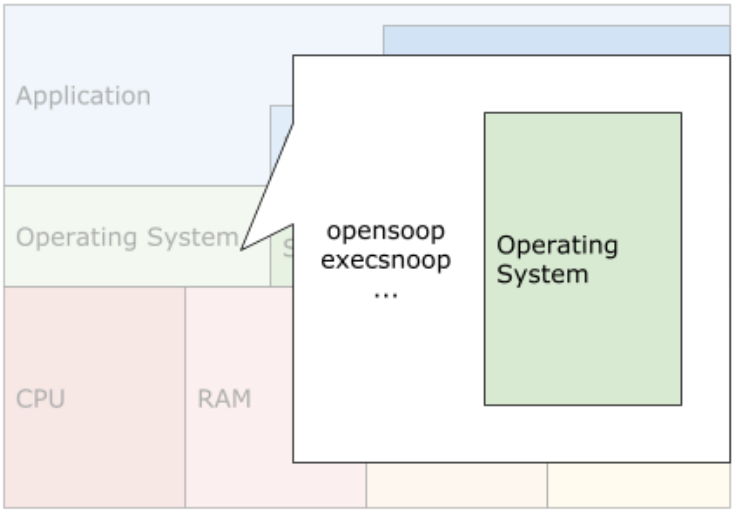


Figure 3.8 Zooming into OS observability tools

3.3.6.1 OPENSNOOP

Opensnoop allows you to see all the files being opened by all the processes on your system, in what's basically real time. BPF really is kind of like a Linux super power, isn't it? To start it (again, remember about the postfix for the Ubuntu package), run this in your command line:

```
sudo opensnoop-bpfcc
```

You should start seeing files being opened by various processes on your system. If you want to get a sample of what it can do, try opening another terminal window, and do just one execution of top, by executing this:

```
top -n1
```

You will see an output similar to this one (I've abbreviated most of it for you):

```
(...)
12396  top                6   0 /proc/sys/kernel/osrelease
12396  top                6   0 /proc/meminfo
12396  top                7   0 /sys/devices/system/cpu/online
12396  top                7   0 /proc
(...)
```

```
12396  top                 8   0 /proc/12386/stat
12396  top                 8   0 /proc/12386/statm
12396  top                 7   0 /etc/localtime
12396  top                 7   0 /var/run/utmp
12396  top                 7   0 /proc/loadavg
(...)
```

This is how you know where *top* is getting all of its info from (feel free to go and explore what's in `/proc`). When practicing chaos engineering, you will often want to know what a particular application you didn't write is actually doing, in order to know how to design or implement your experiments. Knowing what files it opens is a really useful superpower. Speaking of which, here's another one for you - *execsnoop*.

3.3.6.2 EXECSNOOP

Execsnoop is similar to *opensoop*, but it listens for calls to *exec* variants in the kernel, which means that we get a list of all the processes being started on the machine. You can start it by running the following command in a prompt:

```
sudo execsnoop-bpfcc
```

While that runs, try to open another terminal window, and execute `ls`. In the first window, *execsnoop* should print an output similar to this one:

```
PCOMM            PID    PPID   RET ARGS
ls               12419  2073     0 /usr/bin/ls --color=auto
```

Now, instead of `ls`, try running the *mystery002* command we started the chapter with in the second terminal window, by running the following command in your prompt:

```
~/src/examples/busy-neighbours/mystery002
```

You will see all the commands being executed, just like in the following output. You should recognize all the auxiliary commands, like *readlink*, *dirname*, *head* and *sleep*. You will also find the *bc* and *stress* commands starting.

```
PCOMM            PID    PPID   RET ARGS
mystery002       12426  2012     0 /home/chaos/src/examples/busy-neighbours/mystery002
readlink         12428  12427    0 /usr/bin/readlink -f /home/chaos/src/examples/busy-
       neighbours/mystery002
dirname          12427  12426    0
bash             12429  12426    0 /usr/bin/bash /home/chaos/src/examples/busy-
       neighbours/benign.sh
bc               12431  12426    0 /usr/bin/bc -l
head             12432  12426    0 /usr/bin/head -n1
sleep            12433  12429    0 /usr/bin/sleep 20
(...)
stress           12462  12445    0 /usr/bin/stress --cpu 2 -m 1 -d 1 --timeout 30
(...)
```

This is an extremely convenient way of looking into what is being started on a Linux machine. Have I mentioned BFS was really awesome?

3.3.6.3 OTHER TOOLS

OS level offers a large surface to cover, so the purpose of this section is not to give you a full list of all tools available to you, but rather to attract your attention to the fact that we can (and should) consider all of that when we're doing chaos engineering.

I didn't include tools like *strace*, *dtrace* and *perf*, which you might have expected to see here (if you don't know them, do look them up). Instead, I've opted in to give you a taste of what BPF has to offer, because I believe that it will slowly replace the older technologies for this use case. I strongly recommend visiting https://github.com/iovisor/bcc and browsing through other available tools. We don't have time to cover them all here, but I hope that I've given you a taste and I'll leave discovering others to you as an exercise. Let's take a look at the top level of our resource map.

3.4 Application

So here we are, we've reached the top layer of our resource map, the application layer. This is where the code is being written directly to implement what the clients want, be it a serious business app, a video game or bitcoin miner. Every application is different and it often makes sense to talk about high-level metrics provided directly by the application in the context of chaos experiments. For example, we could be looking into bank transaction latencies, the number of players able to play at the same time or a number of hashes processes per second. When doing chaos engineering, we will work with these on a case-by-case basis, because they have unique meanings.

But in-between the OS and the application, there is a lot of code running that we don't always think about - the runtimes and libraries. And these are shared across applications and are therefore easier to look into and diagnose. In this section, we'll look into how to see what's going on inside of a Python application. I'll show you how to use *cProfile*, *pythonstat* and *pythonflow* to give you an idea of what we can easily do. Figure 3.8 is once again showing where all of this fits on the resource map.

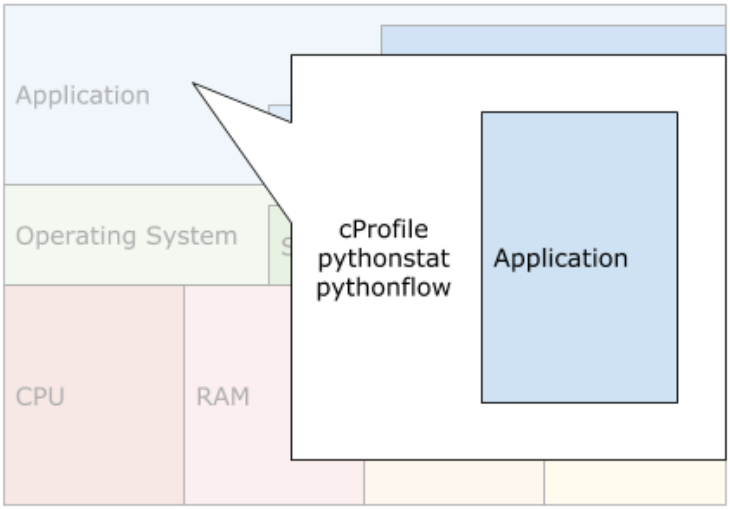


Figure 3.9 Zooming into application observability tools

Let's start with cProfile.

3.4.1 cProfile

Python, true to its "batteries included" motto, ships with two profiling modules: *cProfile* and *profile* (https://docs.python.org/3.7/library/profile.html). We will use the former, as it provides a lower overhead and is recommended for most use cases.

To have a play with it, let's start a Python REPL by running this in a command prompt:

```
python3.7
```

It will present you with some data on the Python binary and a blinking cursor where you can type your commands, much like the following output.

```
Python 3.7.0 (default, Feb  2 2020, 12:18:01)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Imagine that we are again trying to find out why some particular application is slow, and we want to check where it spends its time when executed by Python. That's where a profiler like cProfile can help. In its simplest form, cProfiler can be used to analyze a snippet of code. Try running this in the interactive Python session you just started:

```
>>> import cProfile
>>> import re
>>> cProfile.run('re.compile("foo|bar")')
```

When you run the last line, you should see an output similar to the following one (output abbreviated for clarity). It says that, while running `re.compile("foo|bar")` it makes 243 function calls (236 primitive, or non-recursive), and then lists all the calls. I used a bold font to focus your attention on two columns: `ncalls` (total number of calls, if there are two numbers separated by slash, the second one is the number of primitive calls) and `tottime` (total time spent in there). `Cumtime` is also noteworthy, it gives a cumulative time spent in that call and all its subcalls.

```
         243 function calls (236 primitive calls) in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
(...)
        1    0.000    0.000    0.000    0.000 re.py:232(compile)
(...)
        1    0.000    0.000    0.000    0.000 sre_compile.py:759(compile)
(...)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.exec}
       26    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
    30/27    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.max}
        9    0.000    0.000    0.000    0.000 {built-in method builtins.min}
        6    0.000    0.000    0.000    0.000 {built-in method builtins.ord}
       48    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'
      objects}
        5    0.000    0.000    0.000    0.000 {method 'find' of 'bytearray' objects}
        1    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
        2    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}
        1    0.000    0.000    0.000    0.000 {method 'setdefault' of 'dict' objects}
        1    0.000    0.000    0.000    0.000 {method 'sort' of 'list' objects}
```

To make sense of that, a certain level of understanding of the source code will be really helpful, but using this technique we can at least get an indication of where the slowness might be happening.

If you'd like to run a module or a script, rather than just a snippet, you can run *cProfile* from the command line like this:

```
python -m cProfile [-o output_file] [-s sort_order] (-m module | myscript.py)
```

For example, to run a simple http server, you can run the following command at the prompt. It will wait until the program finishes, so when you're done with it you can press Ctrl-C to kill it.

```
python3.7 -m cProfile -m http.server 8001
```

In another command prompt, make an HTTP call to the server, to check that it works and to generate some more interesting stats, by running this command:

```
curl localhost:8001
```

When you press Ctrl-C in the first prompt, cProfile will print the statistics. You should see a large amount of output, and among these lines, one line of particular interest. This is where our program spent most of its time - waiting to accept new requests.

```
       36   17.682    0.491   17.682    0.491 {method 'poll' of 'select.poll' objects}
```

Hopefully this gives you a taste of how easy it is to get started profiling Python programs and what kind of information you can get out of the box, with just the Python standard library. There are other Python profilers (check https://github.com/benfred/py-spy for example) which offer more ease of use and visualization capabilities. Unfortunately, we won't have time to cover these. Let's take a quick look at another approach - let's leverage BPF.

3.4.2 BCC and Python

In order to use *pythonstat* and *pythonflow* we're going to need a Python binary that was compiled with *--with-dtrace* support to enable us to use the USDT probes (User Statically-Defined Tracing, read more on https://lwn.net/Articles/753601/). These probes are places in the code, where authors of the software defined special endpoints to attach to with DTrace, to debug and trace their applications. Many popular applications, like MySQL, Python, Java, PostgreSQL, Node.js and many more can be compiled with these probes. BPF (and BCC) can also use these probes, and that's how the two tools we're going to use work.

I've precompiled a suitable Python binary for you in `~/Python3.7.0/python`. It was built with *--with-dtrace* to enable support for the USDT probes. In a terminal window, run the following command to start a simple game:

```
~/Python-3.7.0/python -m freegames.life
```

It's Conway's Game of Life implementation which you can find at https://github.com/grantjenks/free-python-games. Now, in another terminal, start *pythonstat* by running:

```
sudo pythonstat-bpfcc
```

You should see an output similar to the following, showing the number of method invocations, garbage collections, new objects, class loaded, exceptions and new threads per second, respectively:

```
07:50:03 loadavg: 7.74 2.68 1.10 2/641 7492

PID    CMDLINE              METHOD/s   GC/s   OBJNEW/s   CLOAD/s  EXC/s  THR/s
7139   /home/chaos/Python-3 480906     3      0          0        0      0
7485   python /usr/sbin/lib 0          0      0          0        0      0
```

Pythonflow, on the other hand, allows you to trace beginnings and ends of execution of various functions in Python. Try it by starting an interactive session in one terminal, by running this command:

```
~/Python-3.7.0/python
```

In another terminal, start *pythonflow* by running the following command:

```
sudo pythonflow-bpfcc $(pidof python)
```

Now, as you execute commands in the Python prompt, you will see the calls stack in the *pythonflow* window. For example, try running this:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

In the *pythonflow* window, you will see the whole sequence needed to import that module.

```
Tracing method calls in python process 7539... Ctrl-C to quit.
CPU PID    TID    TIME(us) METHOD
1   7539   7539   4.547    -> <stdin>.<module>
1   7539   7539   4.547      -> <frozen importlib._bootstrap>._find_and_load
1   7539   7539   4.547        -> <frozen importlib._bootstrap>.__init__
1   7539   7539   4.547        <- <frozen importlib._bootstrap>.__init__
1   7539   7539   4.547        -> <frozen importlib._bootstrap>.__enter__
(...)
```

Try running other code in Python and see all the method invocations appear on your screen. Once again, when practicing chaos engineering, we will often work with other people's code, and being able to sneak peek into what it's doing is going to prove extremely valuable.

I picked Python as an example, but each language ecosystem is going to have its own equivalent tools and methods. Each stack will let you profile and trace applications. We will cover a few more examples in the later chapters of this book. Let's move on to the last piece of this chapter's puzzle - the automation.

3.5 Automation - using time series

All of the tools we've looked into so far are very useful. We've seen how to check which system resources are saturated, how to see system errors, how to look into what's going on at the system level and even to get insight into how various runtimes behave. But they also have one drawback - we needed to sit down and execute each one of them. In this section I would like to discuss a little bit what we can do to automate getting this insight.

There are various monitoring systems available on the market right now. Popular ones include Datadog (https://www.datadoghq.com/), New Relic (https://newrelic.com/) and SysDig (https://sysdig.com/). They all provide some kind of agent you need to run on each of the machines you want to gain insight for, and then give you a way to browse through, visualize and alert on the monitoring data. Figure 3.9 shows a very nice-looking website of Datadog.

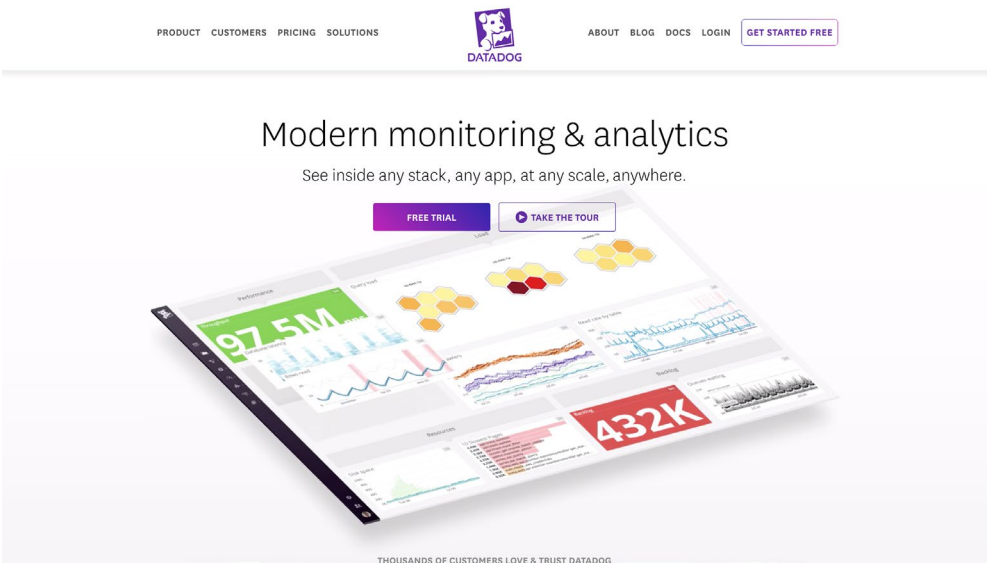


Figure 3.10 Datadog website at https://www.datadoghq.com/

If you'd like to learn more about these commercial offerings, I'm sure their sales people will be delighted to give you a demo. In the context of this book, on the other hand, I'd like to focus on an open source alternatives - *Prometheus* and *Grafana*.

3.5.1 Prometheus & Grafana

Prometheus (https://prometheus.io/) is an open source monitoring system and a time-series database. It provides everything you need to gather, store, query and alert on monitoring data. *Grafana* (https://grafana.com/) is an analytics and visualization tool which works with various data sources, including *Prometheus*. A subproject of *Prometheus* called *Node Exporter* (https://github.com/prometheus/node_exporter) allows for exposing a large set of system metrics.

Together they make for a very powerful monitoring stack. We won't cover setting up production Prometheus, but I want to show you how easily you can get the USE metrics into a time series database by using this stack. To make things faster, we'll use Docker. Don't worry if you're not sure how it works, we'll cover that in later chapters. For now, just treat it as a program launcher.

Let's start by launching Node Exporter, by executing this command at the prompt:

```
docker run -d \
  --net="host" \
  --pid="host" \
  -v "/:/host:ro,rslave" \
  quay.io/prometheus/node-exporter \
  --path.rootfs=/host
```

When it's finished, let's confirm it works, by calling the default port, using the following command:

```
curl http://localhost:9100/metrics
```

You should see an output similar to the following one. This is Prometheus' format - one line per metric, in a simple, human-readable format.

```
promhttp_metric_handler_requests_total{code="200"} 0
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
```

Each line corresponds to a time series. In this example, the same name of the metric (`promhttp_metric_handler_requests_total`) has three different values (200, 500 and 503) for the label "code". That translates to three separate time series, each having some value at any point in time.

Now, Prometheus works by "scraping" metrics, which means making an HTTP call to an endpoint like the one we just called, interpreting the time series data and storing each value at the timestamp corresponding to the time of scraping. Let's start an instance of Prometheus and make it scrape the Node Exporter endpoint. You can do it by first creating a configuration file in your home directory, called `prom.yml` with the following content:

```
global:
  scrape_interval: 5s                                    #A
scrape_configs:
- job_name: 'node'
  static_configs:
  - targets: ['localhost:9100']                          #B
```

#A we set scraping interval to 5 seconds, so that we get the metrics more quickly
#B we tell Prometheus to scrape the Node Exporter which runs on port 9100 (default port)

And then starting Prometheus and passing this configuration file to it, by running this command in your prompt:

```
docker run \
    -p 9090:9090 \
    --net="host" \
    -v /home/chaos/prom.yml:/etc/prometheus/prometheus.yml \
    prom/prometheus
```

When it starts, open Firefox and navigate to http://127.0.0.1:9090/ You will see the Prometheus UI. The UI lets you see the configuration, status and query various metrics. Go ahead and query for the CPU metric node_cpu_seconds_total in the query window and click "Execute". You should see an output similar to figure 3.11.

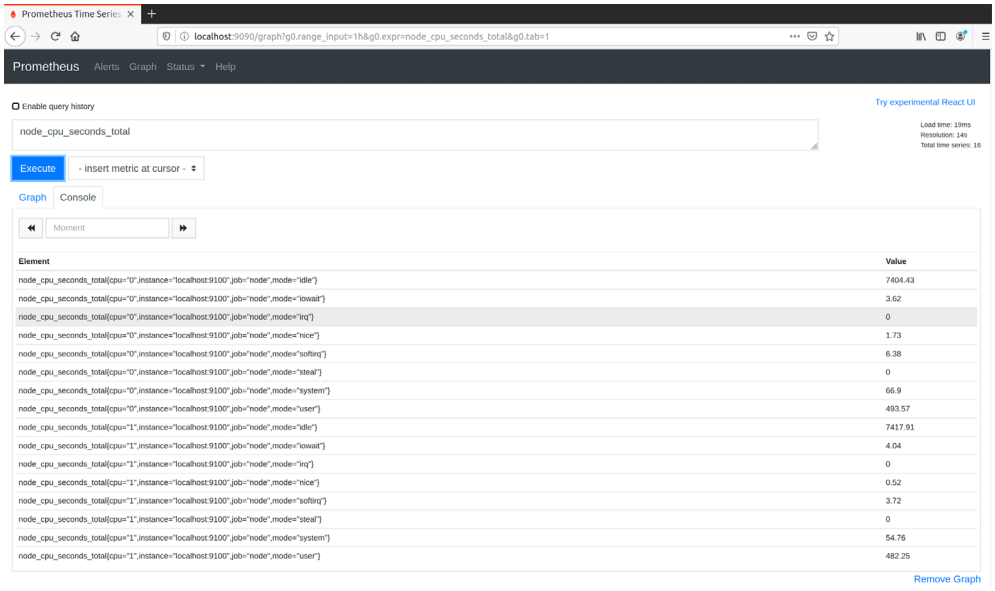


Figure 3.11 Prometheus UI in action, showing node_cpu_seconds_total metric

Notice the different values for the label "mode": idle, user, system, steal, nice, and so on. These are the same categories we were looking at in top. But now, they are a time series and we can plot over time, aggregate them and alert on them easily.

We won't have time to cover querying Prometheus or building Grafana dashboards, so I'm going to leave that as an exercise to the reader. Go to https://prometheus.io/docs/prometheus/latest/querying/basics/ to learn more about Prometheus query language. If you'd like an inspiration for a Grafana dashboard, there are many available at https://grafana.com/grafana/dashboards Take a look at Figure 3.12 which shows one of the dashboards available for download.

Figure 3.12 An example of a Grafana dashboard available at https://grafana.com/grafana/dashboards/11074

OK, hopefully it was as fun for you as it was for me. About time to wrap it up, but before we do, let's look into where to find more information on this subject.

3.6 Further reading

This chapter has been tricky for me. On one hand, I wanted to give you tools and techniques you'll need in the following chapters to practice chaos engineering, so this section grew quickly. On the other hand, I wanted to keep it to a minimum because it's not a system performance book. That means that I had to make some choices, and had to skip some great tools. If you'd like to delve deeper into the subject, I recommend the following books:

- "Systems Performance: Enterprise and the Cloud" by Brendan Gregg, Prentice Hall; 1 edition (October 26, 2013) (http://www.brendangregg.com/sysperfbook.html)
- "BPF Performance Tools" by Brendan Gregg, Addison-Wesley Professional; 1 edition (December 23, 2019) (http://www.brendangregg.com/bpf-performance-tools-book.html)

- "Linux Kernel Development: Linux Kernel Development" by Robert Love, Addison-Wesley Professional; 3 edition (June 22, 2010) (https://rlove.org/)

And that's a wrap!

3.7 Summary

- When debugging a slow application, we can use USE method: check for Utilization, Saturation and Errors
- Resources to analyze include physical devices (CPU, RAM, disk, network, etc) as well as software resources (syscalls, file descriptors, etc)
- Linux provides a rich ecosystem of tools available, like *free*, *df*, *top*, *sar*, *vmstat*, *iostat*, *mpstat*, *BPF*
- BCC makes it easy to leverage BPF to gain deep insights into the system with often negligible overheads
- We can gain valuable insights at various levels: physical components, OS, library/runtime, application

4

Database trouble & testing in production

This chapter covers

- **Design and implement chaos experiments on a popular application (WordPress with MySQL)**
- **Implement a chaos experiment adding network latency using the Linux traffic control tool**
- **When *testing in production* might make sense and how to approach it**

In this chapter we will apply everything we've learned about chaos engineering so far on a real-world example of a common application you might be familiar with. Have you heard of WordPress? It's a popular blogging engine and content management system. According to some estimates, WordPress accounts for more than a third of all pages on the internet, and for most CMS-backed websites (https://w3techs.com/technologies/details/cm-WordPress). It's typically paired with a MySQL database, another very popular piece of technology.

Let's take a vanilla instance of WordPress backed by MySQL and using chaos engineering, try to gain confidence in how reliably we can run it. We'll try to preemptively guess what conditions might disturb it and design experiments to verify how it fares. Ready? Let's see what our friends from Glanden are up to these days.

4.1 We're doing WordPress

It's magical what venture capital (VC) dollars can do while they last. A lot has changed at our favorite startup from Glanden since we last saw them some 30 odd pages ago. The CEO read The *Lean Startup* by Eric Ries last weekend (http://theleanstartup.com/). That, coupled with mediocre FizzBuzz-as-a-service sales, resulted in *pivoting*, or changing direction in The Lean Startup lingo. In practice, apart from a lot of talking, pivoting meant a personnel reshuffle

(Alice is now leading a team of Site Reliability Engineers (SREs) and the engineering is led by a newcomer, Charlie), a new logo (Meower), and a complete change of business model ("Meower is like Uber for cats"). The details of the business model and demand for such feline transportation service remain a little fuzzy.

What's not fuzzy at all is the direct recommendation from the CEO: "We're doing *WordPress* now." Alice's team was tasked to take all the wisdom about running applications reliably from FizzBuzz-as-a-service and apply it to the new, WordPress-based *Meower*. No point arguing with the CEO, so let's get straight to work!

Here is where we come in. We will work with a vanilla installation of WordPress, which comes preinstalled in your Ubuntu VM. Let's take a look under the hood. Figure 4.1 shows an overview of the components of the system. *Apache2* (a popular HTTP server) is used to handle the incoming traffic. *WordPress* application, written in PHP, processes the requests and generates responses. MySQL is used to store the data for the blog.

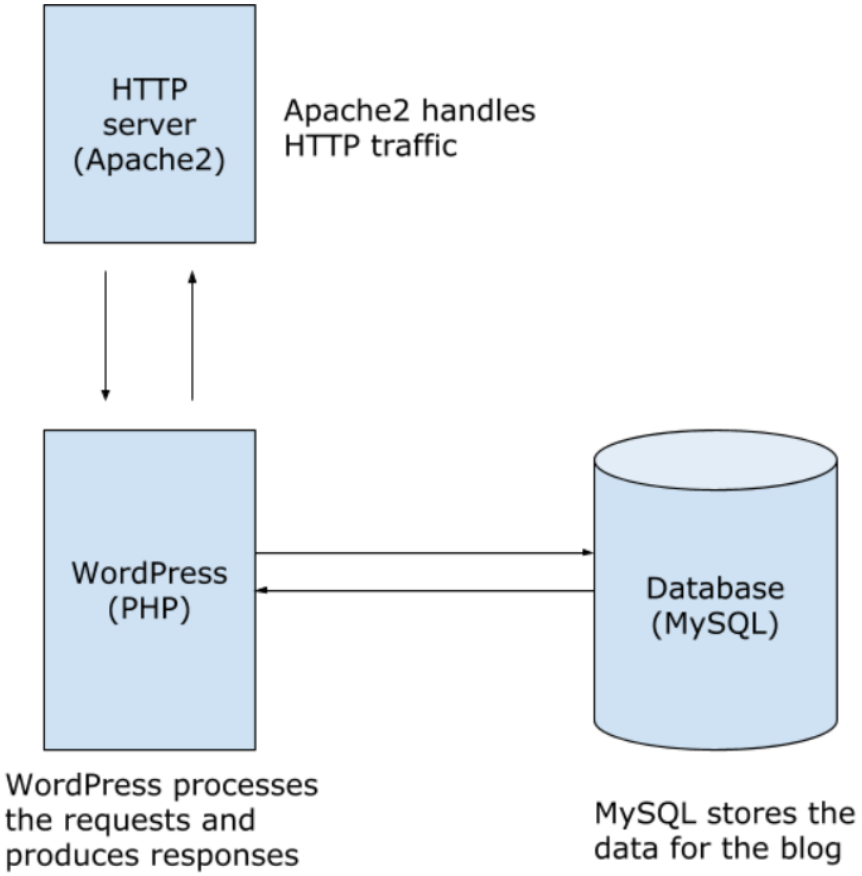


Figure 4.1 WordPress setup handling a user request

WordPress has packages readily available for a wide selection of Linux distributions. In the VM provided with this book, the software comes preinstalled through the default Ubuntu packages and the remaining step is to start and configure it. You can start it by running the following commands at the terminal command prompt inside your VM. It will stop nginx (if it was still running from previous chapters), then start the database and the HTTP server.

```
sudo systemctl stop nginx                          #A
sudo systemctl start mysql
sudo systemctl start apache2
#A Stop nginx if it was running from previous chapters
```

The *Apache2* web server should now be serving *WordPress* on http://localhost/blog. To confirm it's working well, and to configure the WordPress application, open *Firefox* and go to http://localhost/blog. You will see a configuration page. Please fill in the details with whatever you like (just remember the password, we'll need it to log into WordPress later on) and click "Install WordPress." When it's finished, it will allow you to log in and you can start using your WordPress blog.

We should now be ready to roll! Time to put your chaos engineer hat on and generate some ideas for a chaos experiment. In order to do that, let's identify some weak points of this very simple setup.

4.2 Weak links

Let's look at the system again from the perspective of a chaos engineer. How does it work on a high level? Figure 4.2 provides an overview of the setup by showing what happens when a client makes a request to Meower. *Apache2* (a popular HTTP server) is used to handle the incoming HTTP traffic (1). Behind the scenes Apache2 decodes HTTP, extracts the request and calls out to the *PHP* interpreter running *WordPress* application to generate the response (2). *WordPress* (*PHP*) connects to a *MySQL* database to fetch the data it needs to produce a response (3). The response is then fed back to *Apache2* (4), which returns the data to the client as a valid HTTP response (5).

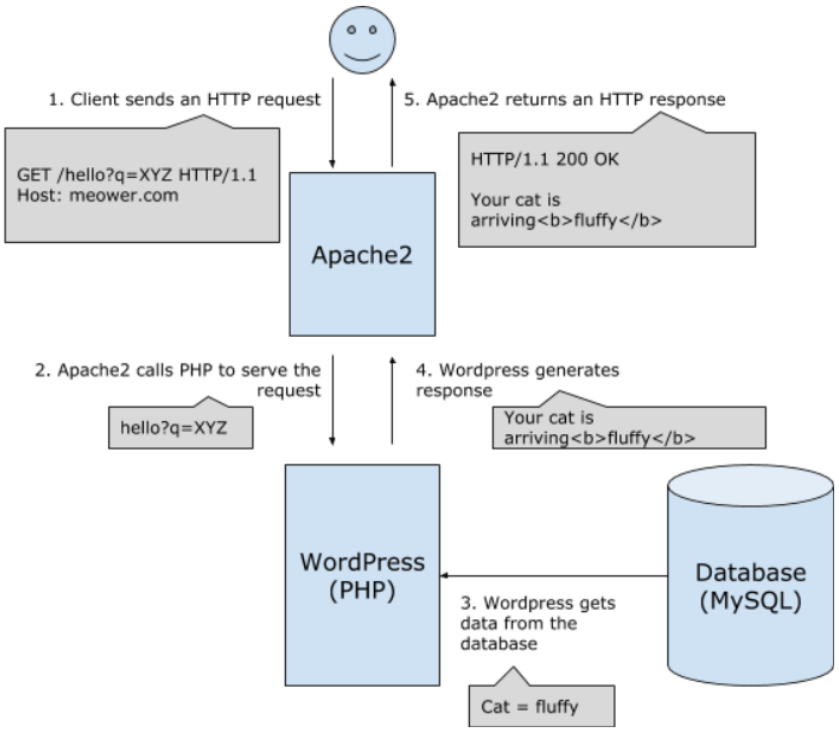


Figure 4.2 WordPress setup handling a user request

This is where the fun part of doing chaos engineering begins. With this high-level idea of how the system works we can start looking into how it breaks. Let's try to predict where the fragile points of the system are, and how to experimentally test whether they are resilient to the type of failure we expect to see. Where would you start?

Finding weak links is often equal measures science and art. Based on an often-incomplete mental picture of how a system works, the starting points for chaos experiments are effectively educated guesses on where fragility might reside in a given system. By leveraging past experience and employing various heuristics, we can make guesses, which we'll then turn into actual science through chaos experiments. One of such heuristics is that often the parts of the system responsible for storing state are the most fragile ones. If we apply that to our very simple example, we can see that the database might be the weak link. With that, here are two examples of educated guesses of a systemic weakness:

1. The database might require good disk IO speeds: what happens when they slow down?
2. How much slowness can we accept in networking between the app server and the database?

They are both great learning opportunities, so let's try to develop them into full-featured chaos experiments, starting with the database disk IO requirements.

4.2.1 Experiment 1: slow disks

We suspect that disk IO degradation might have a negative effect on our application's performance. Right now it's just an educated guess. To confirm or deny, like any mad scientist, we turn to an experiment for answers! Luckily, by now you're familiar with the four steps to designing a chaos experiment introduced in chapter 1:

1. Observability
2. Steady state
3. Hypothesis
4. Run the experiment!

Let's go through the steps and design a real experiment!

First, we need to be able to reliably observe the results of the experiment. To do that, we need a reliable metric. We are interested in our website's performance, so an example of a good metric we could start with is the number of successful requests per second (RPS). It's easy to work with (single number) and we can easily measure it with the Apache bench we've seen in a previous chapter - all of which makes it a good candidate for starters.

Second, we need to establish a steady state. We can do that by running Apache bench on the system without any modifications and reading the normal range of successful requests per second.

Third, the hypothesis. You've only learned about this system at the beginning of this chapter, so it's OK to start with a simple hypothesis and then refine it as we do our experiments and learn more about the characteristics of the system. One example of a simple hypothesis like that could be, "If the disk IO is 95% used, the successful requests per second won't drop by more than 50%." It represents a potential real-world situation, in which another process, let's say a logs cleaner/rotator, kicks in and uses a lot of disk IO for a period of time. The values I chose here (95% and 50%) are completely arbitrary, just to get us started. In the real world, they would come from the SLOs we are trying to satisfy. Right now, we know very little about the system, so let's start somewhere and refine it later.

With these three elements, we're ready to implement our experiment. I'm sure you can't wait, so let's do this!

4.2.1.1 Implementation

Before we make any change to the system, let's measure our baseline - define the steady state. The steady state is the value of our chosen metric during normal operation - that is when we don't run any chaos experiments and the operation of the system is representative of its usual behaviour. With the simple metric of successful RPS, it's simple to measure that steady state with Apache bench. We used it before in chapter 2, but if you need a refresher, you can run `man ab` at your command prompt.

When measuring the baseline, it's important to control all parameters, so that later on we can compare apples to apples, but the values themselves are right now completely arbitrary. Let's start by calling `ab` with a concurrency of one (`-c 1`), for a max of 30 seconds (`-t 30`)

and let's remember to ignore variable length of the response (`-l`) You can do that by running the following command at your command prompt. Be careful to add the trailing slash, because otherwise you'll get a redirect response, which is not what we are trying to test!

```
ab -t 30 -c 1 -l http://localhost/blog/
```

You will see an output similar to the following (abbreviated for clarity). Note, that if you run it multiple times, you will get slightly different values, but they should be similar. In my example output there are no failed requests, and the requests-per-second (RPS) value is 86.33.

```
(...)
Concurrency Level:      1
Time taken for tests:   30.023 seconds
Complete requests:      2592
Failed requests:        0                         #A
Total transferred:      28843776 bytes
HTML transferred:       28206144 bytes
Requests per second:    86.33 [#/sec] (mean)      #B
Time per request:       11.583 [ms] (mean)
Time per request:       11.583 [ms] (mean, across all concurrent requests)
Transfer rate:          938.19 [Kbytes/sec] received
(...)
```

#A Failed requests is none
#B RPS is around 86

When I ran it a dozen times, I received similar values. Remember, that it will depend entirely on your hardware and on how you configure your VM. In my example output, we can take the value of 86 RPS above as our steady state.

Now, how do we implement the conditions for our hypothesis? In chapter 3, we were tracking a mysterious process called *stress*. It's a utility program designed to stress-test your system, capable of generating load for CPU, RAM and disks. We can use it to simulate a program hungry for disk IO. The option `--hdd n` allows us to create n workers; each of which writes files to the disk and then removes them.

In our arbitrarily chosen value for the hypothesis, we used a percentage. In order to generate a load of 95%, we first need to see what our practical 100% is: let's see how quickly we can write to disk. In one terminal window, start `iostat`, by running the following command. We will use it to see the total throughput, updated every 3 seconds. We will use that to monitor the disk write speed.

```
iostat 3
```

In a second terminal window, let's run the `stress` command benchmarking disk with the `--hdd` option and start with a single disk-writing worker. Run the following command in the second terminal window, which will run as specified for 35 seconds.

```
stress --timeout 35 --hdd 1
```

In the first window, you will see an output similar to the following. Depending on your PC configuration, the values will vary. In the following output, it tops at around 1GB/s (in bold

font), and for the sake of simplicity we'll assume that this is the practical 100% of our available throughput.

```
Device             tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
loop0             0.00         0.00         0.00          0          0
sda            1005.00         0.00   1017636.00          0    2035272
```

Depending on your setup, you might need to experiment with extra workers to see what your 100% throughput is like. Don't worry about the exact number too much though; we are running all of this inside of a VM, so there are going to be multiple levels of caches and platform-specific considerations to take into account that won't be discussed in this chapter. The goal here is to teach you how to design and implement your own experiments, but the low-level details need to be addressed case by case.

To double-check your numbers, we can run another test. `dd` is a utility for copying data from one source to another. If we copy enough data to stress test the system, it will give us an indication of how quickly we can go. In order to copy data from `/dev/zero` to a temporary file 15 times in blocks of 512MB, run the following command at your prompt:

```
dd if=/dev/zero of=/tmp/file1 bs=512M count=15
```

The output will look similar to the following (the average write speed is in bold font). In this example, my speed was around 1GB/s, similar to what we found with *stress*. Once again to simplify, let's go with 1GB/s write speed as our throughput.

```
15+0 records in
15+0 records out
8053063680 bytes (8.1 GB, 7.9 GiB) copied, 8.06192 s, 998 MB/s
```

Finally, we should compare our findings against the theoretic limits. Although Apple doesn't publish official numbers for their SSD drives, benchmarks on the internet put the value at about 2.5GB/s. The results we found at less than half that speed in our VM running with the default configuration sound therefore plausible. So far so good.

Now, in our initial hypothesis, we wanted to simulate 95% disk write utilization. As we saw earlier, a *stress* command with a single worker consumes just about 95% of that number. How convenient! It's almost like someone chose that value on purpose! Therefore, to generate the load we wanted, we can just reuse the same `stress` command as earlier on. The scene is set!

Let's run the experiment. In one terminal window, let's start stress with a single worker for 35 seconds (giving us the extra 5 seconds to start ab in the other terminal), by running the following command:

```
stress --timeout 35 --hdd 1
```

In a second terminal window, let's rerun our initial benchmark with Apache bench. Do that by running the following command:

```
ab -t 30 -c 10 -l http://localhost/blog/
```

When ab is finished, you should see an output similar to the following. There are still no errors and the RPS in this sample is 53.92, or a 38% decrease.

```
(...)
Concurrency Level:      1
Time taken for tests:   30.009 seconds
Complete requests:      1618
Failed requests:        0                         #A
Total transferred:      18005104 bytes
HTML transferred:       17607076 bytes
Requests per second:    53.92 [#/sec] (mean)      #B
Time per request:       18.547 [ms] (mean)
Time per request:       18.547 [ms] (mean, across all concurrent requests)
Transfer rate:          585.92 [Kbytes/sec] received
(...)
```

#A Failed requests is none
#B RPS is around 54

Conveniently, this value fits comfortably within the 50% slowdown that our initial hypothesis allowed for and lets us conclude this experiment with success. Yes, if some other process on the same host as our database suddenly starts writing to the disk, taking 90%+ of the bandwidth, our blog continues working, and slows down by less than 50%. In absolute terms, the average time per request went from 12ms to 19ms, which is unlikely to be noticed by any human.

Deus Ex Machina

In this example it is indeed very convenient that we didn't need to limit the writing speed of our stress command to another value, like 50%. If we did, one way of achieving the desired effect would be to calculate the maximum throughput that we want to allow as a percentage of the total throughput we discovered (for example, 50% of 1GB/s would be 512MB/s) and then leverage cgroups v2 (https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html) to limit stress command to that value.

Congrats, another chaos experiment under your belt. But before you pat yourself on the back, let's discuss our science.

4.2.1.3 Discussion

One of the big limitations of this implementation is that all the processes involved -- the application server, the application, the database, the stress command, and the ab command -- run on the same host (and the same VM). That means, that while we were trying to simulate the disk writes, the action of writing to the disk requires CPU time and that's what probably had a larger impact on the slowdown than the writing itself. And even if the writing is the main factor, which component does it affect the most? These are all things we brushed aside in here, but I want you to start being mindful of them because they will become relevant in the more serious applications of chaos engineering. When writing this book, I've tried to make it as simple as possible for you to follow the examples and see things for yourself. In this case, I chose to sacrifice realism for the ease of use to help the learning process. Please don't petition to kick me out of The Royal Society (I'm not a member) just yet!

Another thing worth noting, is that average RPS, while a good starting point, is not a perfect metric, because like any average, it loses information about the distribution. For example, if we average two requests, one which took 1ms and another which took 1s, the average is ~0.5s, but it doesn't say anything about the distribution. A much more useful metric would be a 90th, 95th, or 99th percentile. I chose the simple metric for learning purposes and in the later chapters we will look at the percentiles.

Also, in this example we chose to simulate using up the disk's throughput through writing. What would happen if we chose to do a lot of reading instead? How would the filesystem caching come into play? What filesystem should we use to optimize our results? Would it be the same, if we had NVMe disks, instead of SATA, which can do some of the reading and writing in parallel? What would happen if we did some writing and then some reading to try to use up the disk-writing bandwidth? All of these are relevant questions, which we would need to consider when implementing a serious chaos experiment. And much like in this example, often we will be uncovering new layers as we implement the experiment and realize the importance of other variables. We will not have time to drill into any of these questions right now, but I do recommend that you try to research some of them as an exercise.

Finally, in both our cases, we were running with a single request at a time. This made it easier to manage in our little VM, but in the real world, it's a very unlikely scenario. Most traffic will be bursty. It's possible that a different usage pattern would put more stress on the disk, and would yield different results.

With all these caveats out of the way, let's move on to the second experiment: what happens when networking slows down?

4.2.2 Experiment 2: slow connection

Our second idea of what could go wrong with our application was around the networking being slow: how would that affect the end-user speed of the blog? In order to turn that idea into a real chaos experiment, we need to define what "being slow" means and how we expect it to affect our application. From there, we can follow the four steps to a chaos experiment.

The definition of "being slow" is wildly contextual. A person spending 45 minutes picking something to watch on Netflix will likely get offended by an accusation of being slow, but the same person waiting the same 45 minutes for a life-saving organ donation to be delivered from a different hospital will have a very different experience of time (unless they're in anesthesia). Time truly is relative.

Similarly, in the computer world, a high-frequency trading fund will care about every millisecond of latency, but let's be honest: the latest cat video on YouTube taking an extra second to load is hardly a dealbreaker. In our case, Meower needs to become a commercial success, so we need the website to feel snappy for the users. Following the current best practices, it looks like the website needs to load for our users in less than 3 seconds, or otherwise the probability of users leaving increases significantly (https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/). We will need to account for the actual time it takes for the user to download our content, so let's start with a goal of not going more than 2.5 seconds in the average response time.

With that goal in mind, let's go through the steps of designing a chaos experiment:

1. Observability
2. Steady state
3. Hypothesis
4. Run the experiment!

First, observability. We care about the response time, so for our metric we can stick with the number of successful requests per second (RPS) - the same we leveraged in the previous chaos experiment. It's easy to use and we already have tools to measure it. I mentioned the downsides of using averages in the previous section, but for our use in this example, the successful RPS will do just fine.

Second, the steady state. Because we're using the same metric, we can reuse the work we've done with `ab` to establish our baseline.

Third, the actual hypothesis. We already observed in the previous experiment that with a concurrency of 1, we were in double-digit milliseconds for average response time. Remember, that all of our components are running on the same host, so the overhead of networking is much smaller than it would be if the traffic was going over an actual network. Let's see what happens if we add two-seconds delay communicating to our database. Our hypothesis can therefore be, "If the networking between WordPress and MySQL experiences a delay of 2 seconds, the average response time remains less than 2.5 seconds." Again, these initial values are pretty arbitrary. The goal is to start somewhere, and then refine as needed. With that, we can get our hands dirty with the implementation!

4.2.2.1 INTRODUCING LATENCY

How can we introduce latency to communications? Fortunately, we don't need to lay extra miles of cable (which is a viable solution to achieve that. I recommend reading Michael Lewis' "Flash Boys" if you haven't already; https://en.wikipedia.org/wiki/Flash_Boys), because Linux comes with tools that can do that for us. One of the tools is `tc`.

`Tc` stands for traffic control and is a tool used to show and manipulate traffic-control settings - effectively change how the Linux kernel schedules packets. Tc is many things, but *easy to use* is not one of them. If you type `man tc` at your terminal prompt inside of the VM, you will be greeted with the output that follows (abbreviated). Note that the mysterious-sounding *qdisc* is a "queueing discipline" (scheduler), nothing to do with disks.

```
NAME
       tc - show / manipulate traffic control settings

SYNOPSIS
       tc  [ OPTIONS ] qdisc [ add | change | replace | link | delete ] dev DEV [ parent
       qdisc-id | root ] [ handle qdisc-id ] [
       ingress_block BLOCK_INDEX ] [ egress_block BLOCK_INDEX ] qdisc [ qdisc specific
       parameters ]

(...)

        OPTIONS  := { [ -force ] -b[atch] [ filename ] | [ -n[etns] name ] | [ -nm | -
       nam[es] ] | [ { -cf | -c[onf] } [ filename
       ] ] [ -t[imestamp] ] | [ -t[short] | [ -o[neline] ] }

        FORMAT := { -s[tatistics] | -d[etails] | -r[aw] | -i[ec] | -g[raph] | -j[json] | -
```

```
      p[retty] | -col[or] }
```

Let's learn how to use `tc` by example and see how we can add latency to something unrelated to our setup. Take a look at the `ping` command. `ping` is often used to see connectivity (whether a certain host is reachable) and quality (the speed) of connection. It uses the ICMP protocol and works by sending ECHO_REQUEST datagram and expecting an ECHO_RESPONSE from a host or gateway in response. It's widely available in every Linux distro, as well as other operating systems.

Let's see how long it takes to ping google.com. Run the following command at your terminal prompt. It will try to execute three pings and then print statistics and exit:

```
ping -c 3 google.com
```

You will see output similar to the following. I used a bold font to highlight the times. In this example, for the three pings, it took between 4.28ms (minimum) and 28.263ms (maximum) for an average of 14.292ms. It's not too bad for a free cafe Wi-Fi!

```
PING google.com (216.58.206.110) 56(84) bytes of data.
64 bytes from lhr25s14-in-f14.1e100.net (216.58.206.110): icmp_seq=1 ttl=63 time=4.28 ms
64 bytes from lhr25s14-in-f14.1e100.net (216.58.206.110): icmp_seq=2 ttl=63 time=28.3 ms
64 bytes from lhr25s14-in-f14.1e100.net (216.58.206.110): icmp_seq=3 ttl=63 time=10.3 ms

--- google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 6ms
rtt min/avg/max/mdev = 4.281/14.292/28.263/10.183 ms
```

Now, let's use `tc` to add a static 500ms delay to all connections across the board. You can do that by issuing the following command at the prompt. It will add the delay to the device eth0, the main interface in our VM.

```
sudo tc qdisc add dev eth0 root netem delay 500ms
```

Now, to confirm that it worked, let's re-run the ping command by running the following command at the command line prompt.

```
ping -c 3 google.com
```

This time, the output looks different, similar to the following. Notice that the times are all greater than 500ms, confirming that the `tc` command did its job. Once again, bold font highlights the times.

```
PING google.com (216.58.206.110) 56(84) bytes of data.
64 bytes from lhr25s14-in-f14.1e100.net (216.58.206.110): icmp_seq=1 ttl=63 time=512 ms
64 bytes from lhr25s14-in-f14.1e100.net (216.58.206.110): icmp_seq=2 ttl=63 time=528 ms
64 bytes from lhr25s14-in-f14.1e100.net (216.58.206.110): icmp_seq=3 ttl=63 time=523 ms

--- google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 4ms
rtt min/avg/max/mdev = 512.369/521.219/527.814/6.503 ms
```

Finally, you can remove the latency, by running the following command at the prompt.

```
sudo tc qdisc del dev eth0 root
```

Once that's done, it's worth confirming that it works like before by re-running the ping command and verifying that the times are back to normal. Good, new tool in the toolbox. Let's use it to implement our chaos experiment!

4.2.2.2 IMPLEMENTATION

We should now be well-equipped to implement our chaos experiment. Let's start by re-establishing our steady state. Like in the previous experiment, we can do that using the ab command. Run the following command at the prompt.

```
ab -t 30 -c 1 -l http://localhost/blog/
```

You will see an output similar to the following (again, abbreviated for clarity). The average time per request is 11.583 ms.

```
(...)
Time per request:       11.583 [ms] (mean, across all concurrent requests)
(...)
```

Let's now use `tc` to introduce the delay of 2000ms, in a similar fashion to the previous example. Except that this time, instead of applying it to a whole interface, we would like to only target a single program - the MySQL database. How can we add the latency to the database only? This is something that's going to be much easier to deal with once we cover docker in one of the next chapters, but for now we're going to have to solve that manually.

The syntax of `tc` looks pretty obscure at first. I would like you to see it so that you can appreciate how much easier it is going to get when we use higher level tools in the later chapters. We won't go much into details here (learn more at https://lartc.org/howto/lartc.qdisc.classful.html) but `tc` lets you build tree-like hierarchies, where packets are matched and routed using various queueing disciplines.

To only apply the delay to our database, the idea is to match the packets going there by destination port, and leave all others untouched. Figure 4.3 depicts the kind of structure we're going to build. The root (1:) is replaced with a "prio" qdisc, which has three "bands" (think of them as three possible ways a packet can go from there): 1:1, 1:2 and 1:3. For the band 1:1 we only match IP traffic with destination port 3306 (MySQL) and we attach the delay of 2000ms to it. For the band 1:2, we match everything else. Finally, for the band 1:3 we completely ignore it.

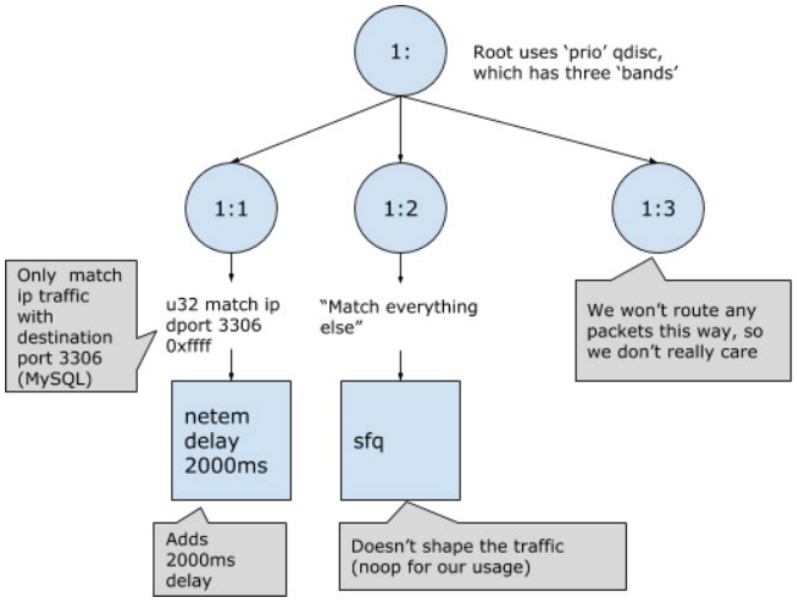


Figure 4.3 High level hierarchy used to classify packets in tc

To set up this configuration, run the following commands at your prompt.

```
sudo tc qdisc add dev lo root handle 1: prio                          #A
sudo tc filter add dev lo protocol ip parent 1: prio 1 u32 match ip dport 3306 0xffff
       flowid 1:1  #B
sudo tc filter add dev lo protocol all parent 1: prio 2 u32 match ip dst 0.0.0.0/0 flowid
       1:2   #C
sudo tc qdisc add dev lo parent 1:1 handle 10: netem delay 2000ms    #D
sudo tc qdisc add dev lo parent 1:2 handle 20: sfq                   #E
```

#A add a "prio" qdisc at the root to create the three "bands": 1:1, 1:2 and 1:3
#B for the band 1:1, match only ip traffic with port 3306 as destination
#C for the band 1:2, match all other traffic
#D add 2000ms delay to band 1:1
#E add Stochastic Fairness Queueing (SFQ) qdisc (noop for our purposes) to band 1:2

That's the "meat" of our experiment. To check that it works, you can now use `telnet` to connect to localhost on port 80 (Apache2) by running the following command at your prompt

```
telnet 127.0.0.1 80
```

You will notice no delay in establishing the connection. Similarly, run the following command at your prompt to test out connectivity to MySQL:

```
telnet 127.0.0.1 3306
```

You will notice that it takes 2 seconds to establish the connection. That's good news. We managed to successfully apply a selective delay to the database only. But if we try to rerun

our benchmark, the results are not what we expected. Run the `ab` command again at the prompt to refresh our benchmark.

```
ab -t 30 -c 1 -l http://localhost/blog/
```

You will see an error message like the following. The program times out before it can produce any statistics.

```
apr_pollset_poll: The timeout specified has expired (70007)
```

We asked `ab` for a 30s test, so a timeout means that it took longer than that to produce a response. Let's go ahead and check how much time it actually takes to generate a response with that delay. We can achieve that by issuing a single request with `curl` and timing it. Run the following command at the prompt to do that.

```
time curl localhost/blog/
```

You should eventually get the response back, and underneath see the output of the `time` command, similar to the following. It took more than 54 seconds to produce a response that used to take 11ms on average without the delay!

```
(...)
real   0m54.330s
user   0m0.012s
sys    0m0.000s
```

To confirm that, let's remove the delay and try the curl command again by running the following commands in the terminal.

```
sudo tc qdisc del dev lo root
time curl localhost/blog/
```

The response will be immediate, similar to the times we were seeing before. What does it say about our experiment? Well, our hypothesis was just proven wrong. Adding a two-second delay in communications going to the database results in much more than a 2.5-second total response time. This is because WordPress communicates with the database multiple times, and with every communication the delay is added. If you'd like to confirm it for yourself, re-run the tc commands changing the delay to 100ms. You will see that the total delay is a multiple of the 100ms we add.

Don't worry though, being wrong is good. This experiment, showing that our initial conception of how the delay would work out, was entirely wrong. And thanks to this experiment we can either find the value we can withstand by playing around with the different delays, or try to change the application to try to minimize the number of round trips and make it less fragile in presence of delays.

There is one more thought that I would like to plant in your head before we move on: that of testing in production.

4.3 Testing in production

I'm expecting that when you saw the delay of 54 seconds caused by the chaos experiment, you thought "Fortunately it's not in production." And that's a fair reaction; in many places, conducting an experiment like this in anything other than a test environment would cause a lot of pain. In fact, testing in production sounds so wrong, that it's become an internet meme.

But the truth is, that whatever testing we do *outside* of the production environment is by definition incomplete. Despite our best efforts, the production environment will always differ from test environments:

- Data will almost always be different
- Scale will almost invariably be different
- User behavior will be different
- Environment configurations will tend to drift away

Therefore, we will never be able to produce 100% adequate tests outside of production. How can we do better? In the practice of chaos engineering, working on (testing) a production system is a completely valid idea. In fact, we strive to do that. After all, it's the only place where the real system - with the real data and the real users - is. Of course, whether that's appropriate will depend on your use case, but it's something we should seriously consider. Let me show you why with an example.

Imagine you're running an internet bank and that you have an architecture consisting of various services communicating with each other. Your software goes through a simple software development life cycle:

- Unit tests are written
- Feature code is written to comply with the unit tests
- Integration tests are run
- Code is deployed to a test stage
- More end-to-end testing is done by a quality assurance (QA) team
- Code is promoted to production
- Traffic is progressively routed to the new software in increments of 5% points of total traffic over a few days

Now, imagine that a new release contains a bug that passed through all these stages, but will only start manifesting itself in rare network slowness conditions. This sounds like something chaos engineering was invented for, right? Yes, but if we only do it in test stages, there are some potential issues:

- Test stage hardware is using previous generation of servers, with a different networking stack, so the same chaos experiment which would catch it in production, wouldn't catch it in the test stage
- Usage patterns in test stage are different from the real user traffic, so the same chaos experiment might pass in test and fail in production
- And so on...

The only way to be 100% sure something works with production traffic is to use production traffic. Should you test it in production? The decision boils down to whether you prefer the risk of hurting some portion of production traffic now, or potentially running into the bug later. And the answer to that will depend on how you see your risks. For example, it might be cheaper to uncover a problem sooner than later, even at the expense of a percentage of your users running into an issue. But equally, it might be unacceptable to fail on purpose for public image purposes. Like with any sufficiently complex question, the answer is "it depends".

Just to be perfectly clear: none of this is to say that we should skip testing our code and ship it directly in production. But with correct preemptive measures in place (to limit the blast radius), running a chaos experiment in production is a real option and can sometimes be tremendously beneficial. From now on, every time we design a chaos experiment, I would like you to ask yourself a question: "Should I do that in the production environment?"

4.4 Summary

- Linux tool `tc` can be used to add latency to network communications
- Network latencies between components can compound and slow the whole system down significantly
- High-level understanding of a system is often enough to make educated guesses about useful chaos experiments
- Experimenting (testing) in production is a real part of chaos engineering
- But chaos engineering is not only about breaking things in production - it can be beneficial in every environment

5

Poking Docker

This chapter covers

- What Docker is and where it came from
- How Docker works under the hood
- How to design chaos experiments for software running in Docker
- How to do chaos experiments on Docker itself
- Using tools like Pumba to easily implement chaos experiments in Docker

Oh *Docker!*. With its catchy name and the lovely whale logo, it has become the public face of Linux containers in just a few short years since the first release in 2013. I now routinely hear things like, "Have you dockerized it?" and, "Just build an image with that, I don't want to install the dependencies." And it's for a good reason. *Docker* capitalized on existing technology in the Linux kernel to offer a convenient and easy-to-use tool, ready for everyone to adopt. It played an important role in taking container technology from the arcane to the mainstream.

In order to be an effective chaos engineer in the containerized world, we need to understand what containers are, how to peek under the hood, and what new challenges (and wins) they present to us. We will focus on *Docker*, as the most popular container technology.

> **DEFINITIONS** What's exactly a container? We'll define this term shortly, but for now just know it's a construct designed to limit the resources a particular program running *inside of a container* can access.

In this chapter, we will start by looking at a concrete example of an application running on Docker. We'll then do a brief refresher on what Docker and Linux containers are, where they came from, how to use them, and how to observe what's going on. Then we'll get our hands dirty to see what containers really contain with a series of experiments. Finally, armed with

this knowledge, we'll execute chaos experiments on the application running in Docker to improve our grasp of how well it can withstand difficult conditions.

My goal is to help you demystify Docker, peek under its hood, and know how it might break. We'll even go as far as to re-implement a container solution from scratch using what the kernel offers us for free, because there is no better learning than through doing.

If this sounds exciting to you, that makes two of us! Let's get the ball rolling by looking at a concrete example of what an application running on Docker might look like.

5.1 My (dockerized) app is slow!

Do you remember *Meower* from chapter 4, the feline transportation service? Turns out that they have been extremely successful and are now expanding to the USA, first targeting Silicon Valley. The local engineering team has been given a green light to redesign the product for the US customers. They decided that they wanted nothing to do with the decades-old Wordpress and PHP, and decided to go down the fashionable route of NodeJS. They picked Ghost (https://ghost.org/) as their new blogging engine, and decided they wanted to use Docker, for its isolation properties and the ease of use. Every developer can now run a mini *Meower* on their laptop without installing any nasty dependencies directly on the host (that's as long as you don't count Docker itself - not even the Mac version running a Linux VM under the hood https://docs.docker.com/docker-for-mac/docker-toolbox/)! After all, that's the least you are going to expect from a well-funded startup, now equipped with napping pods and serving free, organic, gluten-free, personalisable quinoa salads to its engineers daily.

There is only one problem: just like the first version in chapter 4, the new and shiny setup has customers occasionally complaining about slowness, although from the engineering's perspective everything seems to be working fine! What's going on? Desperate for help, your manager offers you a bonus and a raise if you go to San Francisco to fix the slowness in *Meower USA*, just like you did in the last chapter for *Meower Glenden*. SFO, here we come!

Upon arrival, having had an artisanal, responsibly sourced, quinoa sushi-burrito, you start the conversation with the engineering team by asking two pressing questions. First of which is: how does all of it run?

5.1.1 Architecture

Ghost is a NodeJS (https://nodejs.org/en/about/) application designed as a modern blogging engine. It's commonly published as Docker image and accessible through Docker hub (https://hub.docker.com/_/ghost). It supports MySQL (https://www.mysql.com/), as well as sqlite3 (https://www.sqlite.org/) as the data backend. Figure 5.1 shows the simple architecture the Meower USA team has put in place. They're using a third-party, enterprise-ready, cloud-certified load balancer, which is configured to hit in round-robin fashion the Ghost instances, all running on Docker. The MySQL database is also running on Docker and is used as the main datastore for Ghost to write to and read from. As you can see, the architecture is very similar to the one we covered in chapter 4, and in some ways simpler, because the load balancer has been outsourced to another company. But there is one new

element introducing its own complexity, and its name has been mentioned already 10 times in this short section: *Docker*.

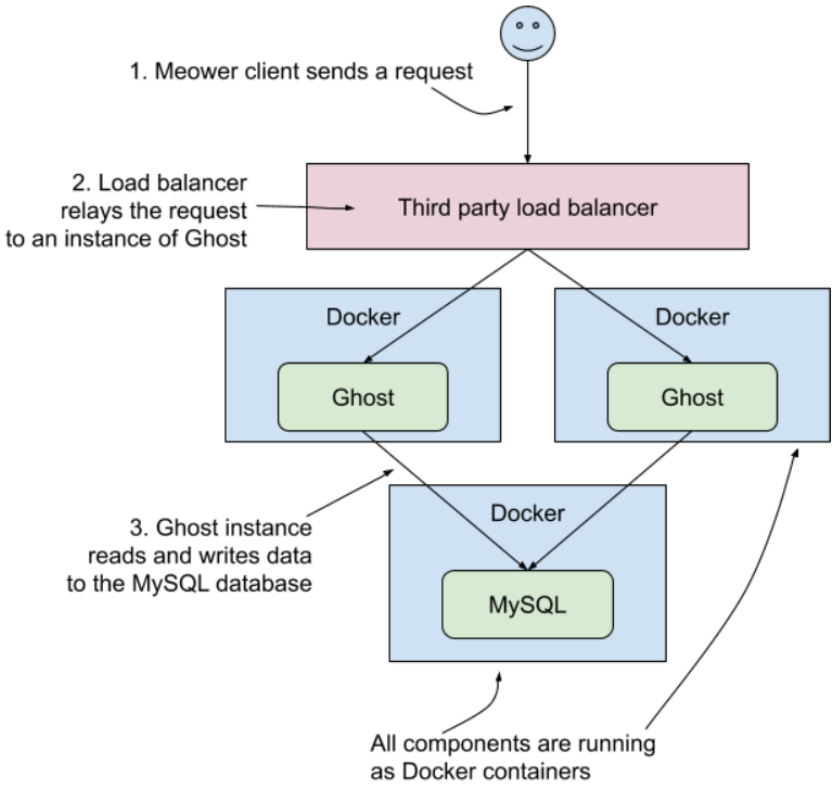


Figure 5.1 High-level overview of Meower USA technical architecture

Which brings to mind the second pressing question you had in mind: what's Docker again? In order to be able to debug and reason about any slowness of the system, you need to build an understanding of what Docker is, how it works, and what underlying technologies it leverages. So take a breath in, a step back, and let's build that knowledge right here, right now. You might want to refill your coffee first. Let's see where Docker came from first.

5.2 The brief history of Docker

When talking about Docker and containers, there are a bunch of connected (and very exciting) concepts that are useful to know. When speaking of them, a lot of things can get a little bit fuzzy, depending on the context, so I'd like to spend a moment to layer the different concepts in a logical order in your brain. Strap in, this is going to be fun. Let's start with emulation, simulation, and virtualization.

5.2.1 Emulation, simulation, and virtualization

An *emulator* is "hardware or software that enables one computer system (called the host) to behave like another computer system (called the guest)" (https://en.wikipedia.org/wiki/Emulator). Why would you want to do that? Well, as it turns out, it's extremely handy. Here are a few examples:

- Testing software designed for another platform without having to own the other platform (potentially rare, fragile or expensive)
- Leveraging existing software designed for a different platform to make products backward-compatible (think new printers leveraging existing firmware)
- Running software (games, anyone?) from platforms that are no longer produced or available at all

I suspect that at least the last point might be close to heart to a lot of readers. Emulators of consoles such as PlayStation, GameBoy, or operating systems like DOS help preserve old games and bring back good memories. When pushed, emulation also allows for more exotic applications, like emulating x86 architecture and running Linux on it... in JavaScript... in a browser (https://bellard.org/jslinux/). Emulation has a broad meaning, but without context people often mean "emulation done entirely in software" when they use this term. Now, to make things more exciting, how does emulation compare to simulation?

A **simulation** is "an approximate imitation of the operation of a process or system; that represents its operation over time" (https://en.wikipedia.org/wiki/Simulation). The keyword here is "imitation." We're interested in the behavior of the system we are simulating, but not necessarily reproducing the internals themselves, like we often do in emulation. Simulators are also typically designed to study and analyze, rather than simply replicate the behavior of the simulated system. A typical example is a flight simulator, where the experience of flying a plane is approximated, or a physics simulation, where the laws of physics are approximated to predict how things will behave in the real world. Simulation is now so mainstream, that films (Matrix anyone?) and even cartoons (https://en.wikipedia.org/wiki/Rick_and_Morty_(season_1) episode 4) talk about it.

Finally, **virtualization** is defined as "the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources" https://en.wikipedia.org/wiki/Virtualization). Therefore, technically speaking, both emulation and simulation can be considered means of achieving virtualization. A lot of amazing work has been done in this domain in the last few decades by companies such as Intel, VMware, Microsoft, Google, Sun, and many more and it's easily a topic for another book.

In the context of *Docker* and containers, we'll be most interested in hardware virtualization (or platform virtualization, which are often used interchangeably), wherein a whole hardware platform (for example, an x86 architecture computer) is virtualized. Of particular interest to us are the following two types of hardware virtualization:

1. **Full virtualization** (virtual machines or VMs) - a complete simulation of the underlying hardware, which results in a creation of a virtual machine that acts like a real computer with an operation system (OS) running on it.

2. **OS-level virtualization** (containers) - the OS ensures isolation of various system resources from the point of view of the software, but in reality they all share the same kernel.

This is summarized by figure 5.2.

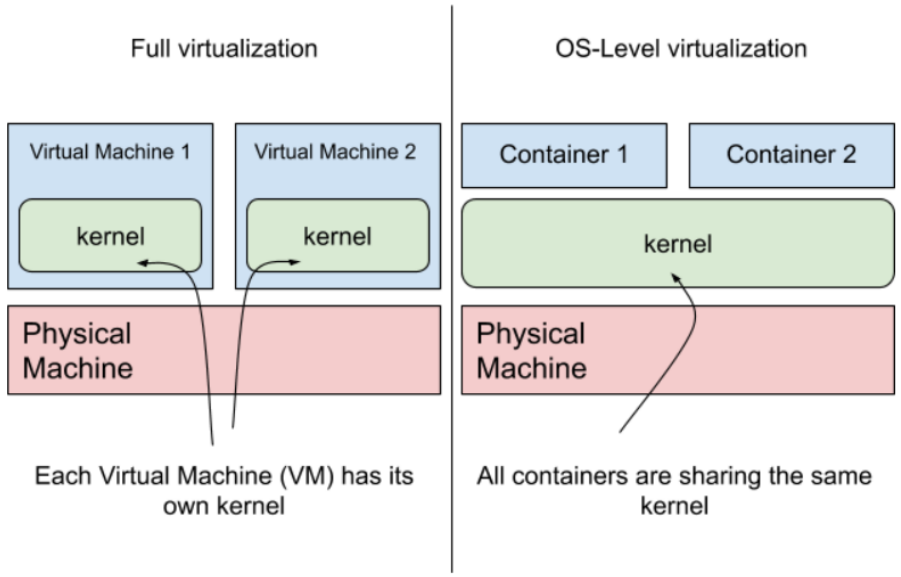


Figure 5.2 Full virtualization vs OS-level virtualization

Sometimes, full virtualization is also referred to as *strong isolation*, and the OS-level virtualization as *lightweight isolation*. Let's take a look at how they compare side to side.

5.2.2 Virtual machines and containers

The industry uses both virtual machines (VMs) and containers for different use cases. Either approach has its own pros and cons. For example, for a virtual machine:

- (pro) fully isolated - more secure than containers
- (pro) can run a different operating system than the host
- (pro) can allow for better resource utilization (VM's unused resources can be given to another VM)
- (con) higher overhead than container due to running operating systems on top of each other
- (con) longer startup time, due to the operating system needing to boot up
- (con) typically running a VM for a single application will result in unused resources

In the same way, for a container:

- (pro) lower overhead, better performance - the kernel is shared
- (pro) quicker startup time
- (con) bigger blast radius for security issues due to shared kernel
- (con) can't run a different OS or even kernel version - it's shared across all containers
- (con) often not all of the OS is virtualized potentially resulting in weird edge cases

Typically, VMs are used to partition larger physical machines into smaller chunks, and offer APIs to automatically create, resize, and delete VMs. The software running on the actual physical host, responsible for managing VMs, is called a *hypervisor*. Popular VM providers include the following:

- VMware vSphere (https://www.vmware.com/products/vsphere.html)
- Xen (https://www.xenproject.org/)
- Microsoft Hyper-V (https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/)
- KVM (https://www.linux-kvm.org/page/Main_Page)
- Oracle VirtualBox (https://www.virtualbox.org/)
- QEMU (https://www.qemu.org/)

Containers, on the other hand, thanks to their smaller overhead and quicker startup time offer one more, crucial benefit: they allow you to package and release software in a truly portable manner. Inside of a container (we'll get to the details in a minute), you can add all the necessary dependencies to ensure it runs well. And you can do that without worrying about conflicting versions or paths on filesystems. It's therefore useful to think of containers as a means of packaging software with some extra benefits (we'll cover them extensively in the next section). Popular containers providers include:

- Docker (https://www.docker.com/)
- LXC (https://linuxcontainers.org/lxc/) and LXD (https://linuxcontainers.org/lxd/)
- Windows Containers (https://docs.microsoft.com/en-us/virtualization/windowscontainers/)

It's worth noting that VMs and containers are not necessarily exclusive; it's not uncommon to run containers inside of VMs. As you will see in the chapter on Kubernetes, it's a pretty common sight right now. In fact, we'll do exactly that later in this chapter!

Finally, virtualization of computer hardware has been around for a while, and various optimizations have been done. It is now expected to have access to *hardware-assisted* virtualization, where the hardware is designed specifically for virtualization, and the software executes approximately at the same speed as if it was run on the host directly.

VM, container, and everything in between

We've been trying to neatly categorize things, but the reality is often more complex. To quote a certain Jeff Goldblum in one of my favorite movies of all time, "Life finds a way." Here are some interesting projects on the verge of a VM and a container:

- https://firecracker-microvm.github.io/ used by Amazon, promises fast startup times and strong isolation microVMs, which would mean the best of both worlds

- https://github.com/kata-containers/runtime offers hardware-virtualized Linux containers, supporting VT-x (Intel), Hyp mode (ARM) and Power Systems, and Z mainframes (IBM)
- https://github.com/solo-io/unik builds applications into unikernels for building microVMs, that can then be booted up on traditional hypervisors, but can boot quickly with low overheads
- https://github.com/google/gvisor offers a user-space kernel, which implements only a subset of Linux system interface, as a way of increasing security level when running containers

Thanks to all these amazing technologies, we now live in a world where Windows ships with a Linux kernel (https://devblogs.microsoft.com/commandline/shipping-a-linux-kernel-with-windows/) and no one bats an eye. I have to confess I quite like this Inception-style reality and I hope that I managed to get you excited as well!

Now, I'm sure you can't wait to dive deeper into the actual focus of this chapter. Time to sink our teeth into Docker.

5.2.3 Linux containers and Docker

Linux containers might look new and shiny, but the journey to where they are today took a little while. I've prepared a handy table for you to track the important events on the timeline (table 5.1). It's not necessary to remember these events to use containers, but it's good to realize the different ideas which appeared in various contexts, which eventually led to (or inspired) what we call Linux containers today. Take a look.

Table 5.1 The chronology of events and ideas leading to the Linux containers we know today

Year	Isolation	Event
1979	filesystem	Unix v7 includes `chroot` system call, which allows changing the root directory of a process and its children to a different location on the filesystem. Often considered the first step towards containers.
2000	files, processes, users, networking	FreeBSD 4.0 introduces `jail` system call, which allows for creation of mini-systems called *jails* which prevent processes from interacting with processes outside of the *jail* they're in.
2001	filesystems, networking, memory	Linux VServer offers a *jail*-like mechanism for Linux, through patching the kernel. Some system calls and parts of `/proc` and `/sys` filesystems are left not virtualized.
2002	namespaces	Linux kernel 2.4.19 introduces `namespaces` which control what set of resources are visible to each process. Initially just for mounts, other namespaces were gradually introduced in later versions (pid, network,

		cgroups, time ...)
2004	sandbox	Solaris releases Solaris Containers (also known as Solaris Zones) which provide isolated environments for processes within them.
2006	CPU, memory, disk IO, network, ...	Google launches *process containers* to limit, account for and isolate the resource usage of groups of processes on Linux. It was later renamed to *control groups* (or `cgroups` for short) and was merged into Linux kernel 2.6.24 in 2007.
2008	containers	LXC (LinuX Containers) offers the first implementation of a container manager for Linux, building on top of cgroups and namespaces.
2013	containers	Google shares lmctfy (Let Me Contain That For You), their container abstraction through an API. Eventually parts of it end up being contributed to the `libcontainer` project.
2013	containers	First version of Docker is released, which builds on top of LXC and offers tools to build, manage and share containers. Later, `libcontainer` is implemented to replace LXC (using `cgroups`, `namespaces` and Linux capabilities.) Containers start exploding in popularity as a convenient way of shipping software, with added resource management (and limited security) benefits.

Docker, through the use of libraries (previously `LXC` and now `libcontainer`), is using features of Linux kernel to implement containers (with some additions we'll look at later in the chapter.) These features are:

- `chroot` - change the root of the filesystem for a particular process
- `namespaces` - isolate what a container can "see" in terms of PIDs, mounts, networking and more
- `cgroups` - control and limit access to resources, such as CPU and RAM
- Capabilities - grant subsets of superuser privileges to users, such as killing other users' processes
- Networking - manage container networking through various tools
- Filesystems - use UnionFS to create filesystems for containers to use
- Security - use mechanisms such as seccomp, SELinux and AppArmor to further limit what a container can do

Figure 5.3 shows what happens when you a user talks to Docker on a conceptual, simplified level. Have a look:

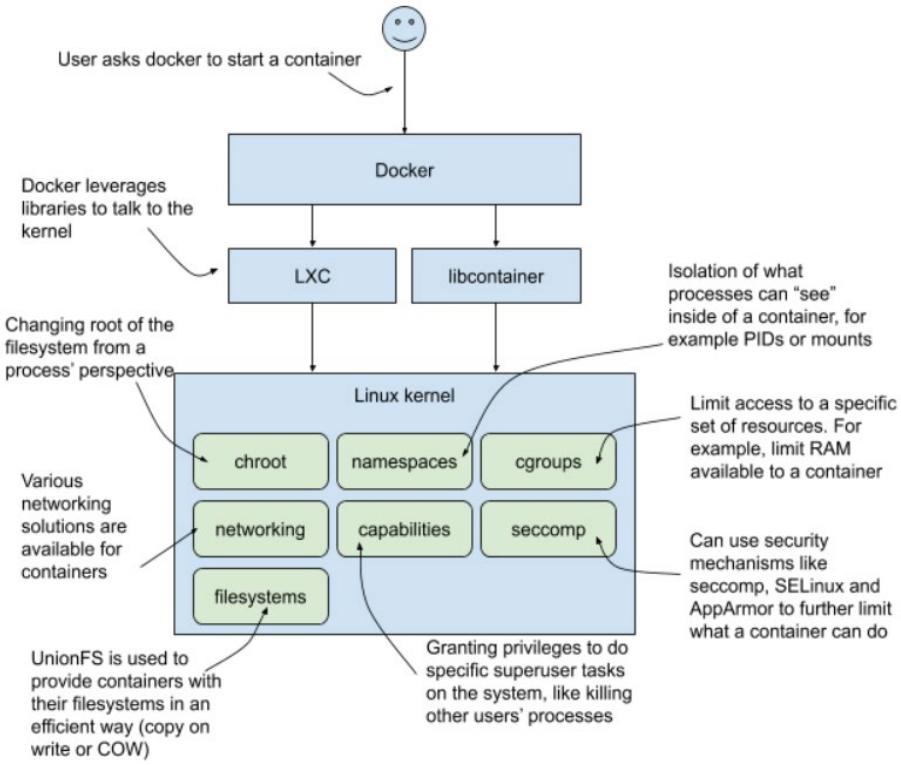


Figure 5.3 High-level overview of Docker interacting with the kernel

So if Docker relies on Linux kernel features for the heavy lifting, what does it actually offer? A whole lot of convenience, like the following:

- *container runtime* - program making the system calls to implement, modify and delete containers, as well as creating filesystems and implementing networking for the containers
- *dockerd* - daemon providing an API for interacting with the container runtime
- *docker* - command line client of *dockerd* API used by the end users
- *Dockerfile* - format for describing how to build a *container*
- *Container image* format - describing an archive containing all the files and metadata necessary to start a container based off that *images*
- *Docker registry* - hosting solution for images
- A protocol for exporting (packaging into an archive), importing (pulling) and sharing (pushing) *images* to *registries*

- Docker Hub - public registry where you can share your *images* for free

Basically, it made using Linux containers easy from the user's perspective, by abstracting all the complicated bits away, smoothing out the rough edges and offering standardized ways of building, importing, and exporting container images.

That's a lot of Docker lingo, so I've prepared figure 5.4 to represent that process. Let's just repeat that to let it sink in:

- A Dockerfile (we'll see some in just a minute) allows you to describe how to *build* a container.
- The container then can be exported (all its contents and metadata stored in a single archive) to an *image*, and *pushed* to a *Docker registry*, for example the *Docker hub* (https://hub.docker.com/), from where other people can *pull* it.
- Once they *pull* an *image*, they can *run* it using the command line *docker* utility.

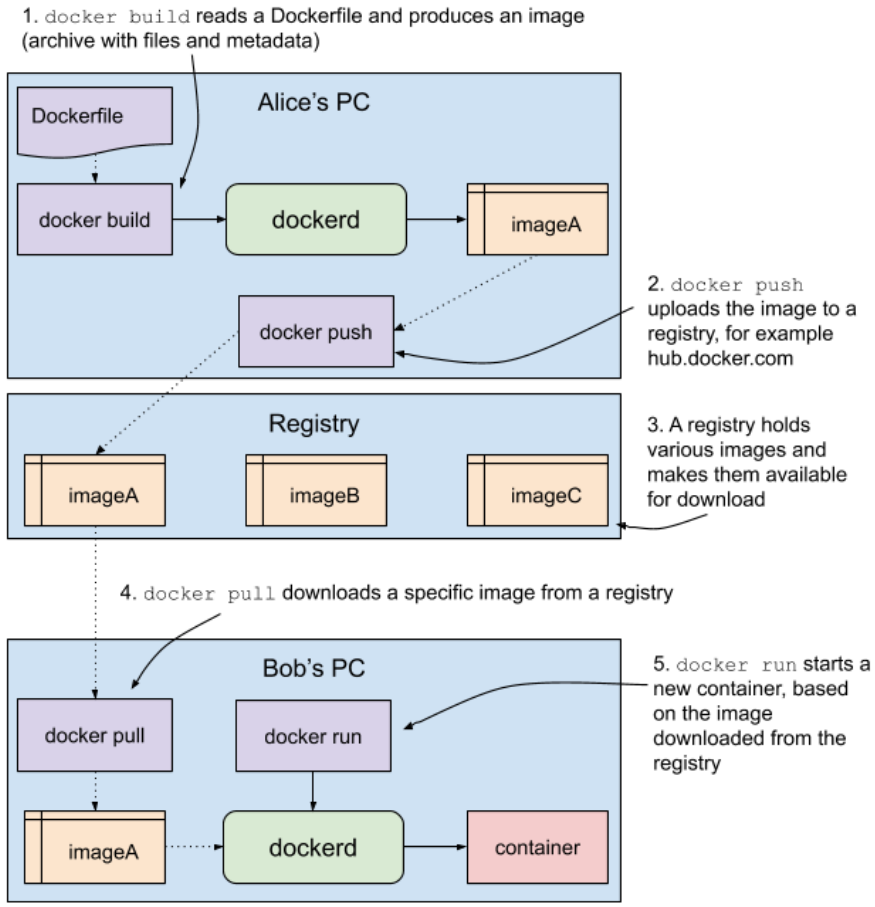


Figure 5.4 Building, pushing, and pulling Docker images

If you haven't used Docker before, don't worry. We're about to look into how all of this works and we'll also cover how to use it. And then break it. Ready to take a peek under the hood?

5.3 Peeking under the Docker's hood

It's time to get our hands dirty. In this section, we'll start a container and see how Docker implements the isolation and resource limits for the containers it runs. Using Docker is simple, but understanding what it does under the hood is essential for designing and executing meaningful chaos engineering experiments.

Let's begin by starting a Docker container! You can do that by running the following command in a terminal inside of your VM. Note, that if you'd like to run it on a different system, you'll most likely need to prepend the commands below with `sudo`, since talking to the Docker daemon requires admin privileges. The VM has been set up to not require that to save you some typing. To make things more interesting, let's start a different Linux distribution - Alpine linux:

```
docker run \
--name firstcontainer \                          #A
-ti \                                            #B
--rm \                                           #C
alpine:3.11 \                                            #D
/bin/sh                                                  #E
```

#A give our container a name "firstcontainer"
#B keep STDIN open and allocate a pseudo-TTY to allow us to type commands (note the single hyphen!)
#C remove the container after we're done with it
#D run image "alpine" in version (tag in Docker parlance) "3.11"
#E execute /bin/sh inside of the container

You should see a simple prompt of your new container running. Congrats! To stop it, all you need to do is exit the shell session. You can type `exit` or press Ctrl-D in this terminal. The `--rm` flag will take care of deleting it once it exits, so that you can start another one with the same name with the exact same command later on. For the rest of this section, I'll refer to commands run in this terminal, inside of the container, as the *first terminal*. So far so good. Let's inspect what's inside!

5.3.1 Uprooting processes with chroot

What's Alpine, anyway? Alpine Linux (https://alpinelinux.org/) is a minimalistic Linux distro, geared for minimal usage of resources and quite popular in the container world. And I'm not joking when I'm saying it's small. Open a second terminal window and keep that second terminal open for a while; we'll use it to look at how things differ from the container's perspective (first terminal) and on the host (second terminal). In the second terminal, run the following command to list all images available to Docker:

```
docker images
```

You will see an output similar to the following (bold font shows the size of the alpine image):

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
```

```
alpine                3.11                f70734b6a266        36 hours ago        5.61MB
(...)
```

As you can see, the `alpine` image is really small, clocking in at 5.6MB. Now, don't take my word for it, let's confirm what we're running by checking how the distro identifies itself. You can do that by running the following command in the first terminal:

```
head -n1 /etc/issue
```

You will see the following output:

```
Welcome to Alpine Linux 3.11
```

In the second terminal, run the same command:

```
head -n1 /etc/issue
```

This time, you will see a different output:

```
Ubuntu 18.04.4 LTS \n \l
```

The content of the file at the same path in the two terminals (inside of the container and outside) is different. How come? In fact, the entire filesystem inside of the container is `chroot`'ed, which means that the `/` inside of a container is a different location on the host system. Let me explain what I mean. Take a look at figure 5.5, that shows an example of a `chroot`'ed filesystem. On the left hand side, there is a host filesystem, with a folder called `/fake-root-dir`. On the right, there is an example of what the filesystem might look like from a perspective of a process `chroot`'ed to use `/fake-root-dir` as the root of its filesystem. This is exactly what you are seeing happen in the container you just started!

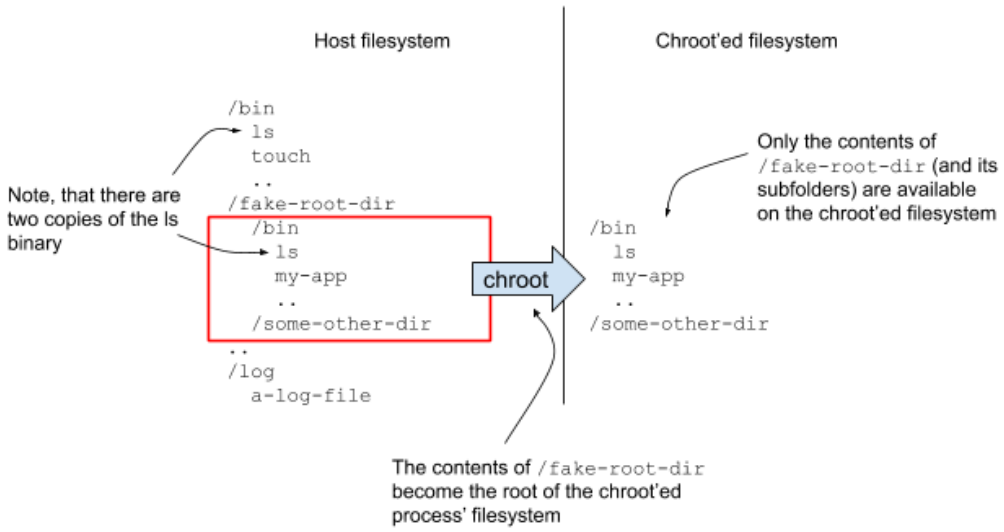


Figure 5.5 Visual example of a chroot'ed filesystem

Union filesystems, overlay2, layers and Docker

An important part of implementing a container solution is to provide a robust mechanism for managing contents of the filesystems the containers start with. One such mechanism, used by Docker, is a union filesystem. In a union filesystem, two or more folders on a host can be presented as a single, merged folder (called union mount) transparently for the user. These folders, arranged in a particular order, are called layers. Upper layers can "hide" lower layers' files by providing another file at the same path. In a Docker container, by specifying the base image, you tell Docker to download all the layers that image is made of, make an union of them and start a container with a fresh layer on top of all of that. This allows for a reuse of these read-only layers in a very efficient way, by only having a single file which can be read by all containers using that layer. Finally, if the process in the container needs to modify a file present on one of the lower layers, it is first copied in its entirety into the current layer (copy on write or COW). Overlay2 is a modern driver implementing this behavior. Learn more about how it works at https://docs.docker.com/storage/storagedriver/overlayfs-driver/

Where is it then? Depending on the storage settings of Docker, it might end up in different places on the host filesystem. To find out where it is, we can use a new command, `docker inspect`. It gives you all the information the Docker daemon has about a particular container. To do that, run the following command in the second terminal:

```
docker inspect firstcontainer
```

The output you're going to see is pretty long, but for now we're just interested in the `GraphDriver` section of it. See the following, abbreviated output showing just that section. The long IDs will be different in your case, but the structure and the `Name` member ("overlay2", the default on the Ubuntu installation in your VM) will be the same. You will notice "LowerDir", "UpperDir", and "MergedDir" (bold font). These are, in respective order, the top layer of the image the container is based on, the read-write layer of the container, and the merged (union) view of the two.

```
...
        "GraphDriver": {
            "Data": {
                "LowerDir":
"/var/lib/docker/overlay2/dc2e04091408ca1215bf4d1f1e8478f1f80034fdf8d1f460e660c3d398
6e8239-
init/diff:/var/lib/docker/overlay2/caf1d3a22be2d8cc0f6027164bcbb8cf10bd7e8ef43aa827f
eb848a3f6c8fe43/diff",
                "MergedDir":
"/var/lib/docker/overlay2/dc2e04091408ca1215bf4d1f1e8478f1f80034fdf8d1f460e660c3d398
6e8239/merged",
                "UpperDir":
"/var/lib/docker/overlay2/dc2e04091408ca1215bf4d1f1e8478f1f80034fdf8d1f460e660c3d398
6e8239/diff",
                "WorkDir":
"/var/lib/docker/overlay2/dc2e04091408ca1215bf4d1f1e8478f1f80034fdf8d1f460e660c3d398
6e8239/work"
            },
            "Name": "overlay2"
        },
...
```

In particular, we're interested in the `.GraphDriver.Data.MergedDir` path, which gives us the location of the container's merged filesystem. To confirm that we're looking at the same actual file, let's read the inode of the file from the outside. To do that, still in the second terminal, run the following command. It uses the `-f` flag supported by Docker, to access only a particular path in the output, as well as `-i` flag in *ls* to print the inode number:

```
export CONTAINER_ROOT=$(docker inspect -f '{{ .GraphDriver.Data.MergedDir }}'
       firstcontainer)
sudo ls -i $CONTAINER_ROOT/etc/issue
```

You will see an output similar to the following (bold font shows the inode number):

```
800436
       /var/lib/docker/overlay2/dc2e04091408ca1215bf4d1f1e8478f1f80034fdf8d1f460e660c3d3986
       e8239/merged/etc/issue
```

Now, back in the first terminal, let's see the inode of the file from the container's perspective. To do that, run the following command in the first terminal:

```
ls -i /etc/issue
```

The output will look similar to the following (again, bold font to show the inode):

```
800436 /etc/issue
```

As you can see, the inodes from the inside of the container and from the outside are the same; it's just that the file shows in different locations in the two scenarios. This is telling of the containers experience in general - the isolation is really thin. We'll see how that's important from the perspective of a chaos engineer in just a minute, but before we do that, let's solidify your new knowledge about chroot by implementing a simple version of a container ourselves.

5.3.2 Implementing a simple container(-ish) part 1 - using chroot

I believe that there is no better way to really learn something than to try to build one yourself. Let's use what you learned about chroot, and take a first step towards building a simple DIY container. Take a look at figure 5.6, that shows which parts of Docker's underlying technologies we're going to use.

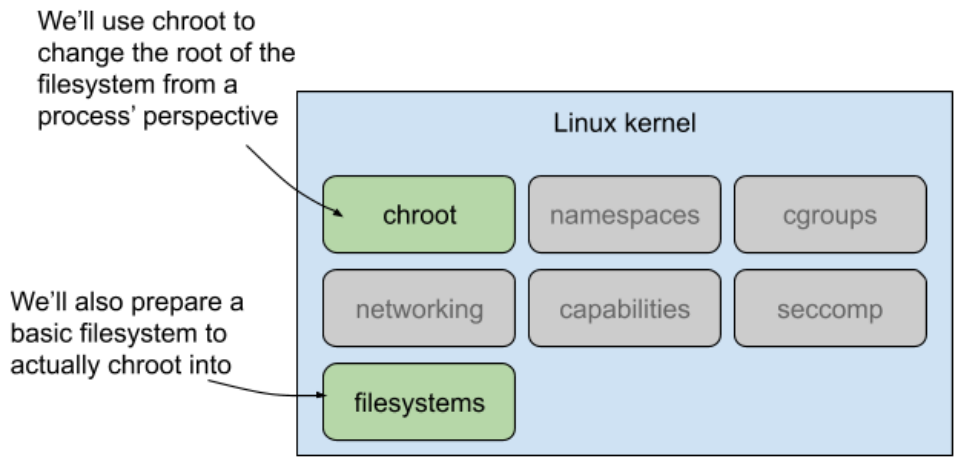


Figure 5.6 DIY container part 1 - chroot and filesystems

As it turns out, changing the root of the filesystem for a new process is rather straightforward. In fact, you can do that with a single command, called - you guessed it - `chroot`. I've prepared a simple script to demonstrate starting a process with the root of its filesystem pointing to a location of your choice. In your VM, open a terminal and type the following command to see the script:

```
cat ~/src/examples/poking-docker/new-filesystem.sh
```

You will see the following output. It's creating a new folder, and copying over some tools and their dependencies, so that we can use it as a root filesystem. It's a very crude way of preparing a filesystem structure to be usable for a `chroot`'ed process. This is necessary so that we can execute something from inside of the new filesystem. The only thing that you might not be familiar with here is the use of the `ldd` command, which prints shared objects dependencies for binaries on linux. These shared objects are necessary for the commands we're copying over to be able to start.

```
#! /bin/bash

export NEW_FILESYSTEM_ROOT=${1:-~/new_filesystem}
export TOOLS="bash ls pwd mkdir ps touch rm cat vim mount"                    #A

echo "Step 1. Create a new folder for our new root"
mkdir $NEW_FILESYSTEM_ROOT

echo "Step 2. Copy some (very) minimal binaries"
for tool in $TOOLS; do
    cp -v --parents `which $tool` $NEW_FILESYSTEM_ROOT;                       #B
done

echo "Step 3. Copy over their libs"
# use ldd to find the dependencies of the tools we've just copied
```

```
echo -n > ~/.deps
for tool in $TOOLS; do
    ldd `which $tool` | egrep -o '(/usr)?/lib.*\.[0-9][0-9]?' >> ~/.deps       #C
done
# copy them over to our new filesystem
cp -v --parents `cat ~/.deps | sort | uniq | xargs` $NEW_FILESYSTEM_ROOT       #D

echo "Step 4. Home, sweet home"
NEW_HOME=$NEW_FILESYSTEM_ROOT/home/chaos
mkdir -p $NEW_HOME && echo $NEW_HOME created!
cat <<EOF > $NEW_HOME/.bashrc
echo "Welcome to the kind-of-container!"
EOF

echo "Done."
echo "To start, run: sudo chroot" $NEW_FILESYSTEM_ROOT                          #E
```

#A list some binaries we'll copy into the new root
#B copy the binaries, maintaining their relative paths with –parents
#C use ldd to list shared libraries they need and extract their locations to .deps
#D copy the libraries, maintaining their structure
#E print usage instructions

Let's go ahead and run this script, passing as an argument the name of the new folder to create in our current working directory. You can do it by running the following command in your terminal:

```
bash ~/src/examples/poking-docker/new-filesystem.sh not-quite-docker
```

After it's done, you will see a new folder, `not-quite-docker`, with a very minimal structure inside of it. We can now start a `chroot`'ed bash session by running the following command in your terminal (sudo is required by chroot):

```
sudo chroot not-quite-docker
```

You will see a short welcome message, and you'll be in a new bash session. Go ahead and explore; you will find you can create folders and files (we copied `vim` over), but if you try to run `ps`, it will complain about the missing `/proc`. And it is right to complain about it - it's not there! The purpose here is to demonstrate to you the workings of `chroot` and to make you comfortable designing chaos experiments. But for the curious, you can go ahead and mount the `/proc` inside of your chrooted process, by running the following commands in your terminal (outside of chroot):

```
mkdir not-quite-docker/proc
sudo mount -t proc /proc/ not-quite-docker/proc
```

In the context of isolating processes, this is something we might or might not want to do. For now, treat this as an exercise or a party trick, whichever works best for you!

Now, with this new piece of knowledge that takes away some of the magic of Docker, you're probably itching to probe it a bit. If the containers are all sharing the same host filesystem, they are just mounted in different locations, it should mean that one container can fill in the disk, and prevent another one from writing, right? Let's design an experiment to find out!

5.3.3 Experiment 1: can one container prevent another one from writing to disk?

Intuition hints that if all containers' filesystems are just `chroot`'ed locations on the host's filesystem, then one busy container filling up the host's storage can prevent all the other containers from writing to disk. But human intuition is fallible, so it's time to invite some science and design a chaos experiment.

First, we need to be able to observe a metric that quantifies "being able to write to disk." To keep it simple, I suggest we create a simple container that tries to write a file, erases it, and retries again every few seconds. We'll be able to see whether it can still write, or not. Let's call that container "control."

Second, our steady state. Using our container, we'll first verify that it can write to disk.

Third, our hypothesis. If another container (let's call it "failure" container) consumes all available disk space until there is no more left, then the control container will start failing to write.

To recap, here's the four steps to our chaos experiment:

1. Observability - a "control" container printing whether it can write every few seconds
2. Steady state - the "control" container can write to disk
3. Hypothesis - if another "failure" container writes to disk until it can't, the "control" container won't be able to write to disk any more
4. Run the experiment!

Implementation time! Let's start with the "control" container. I've prepared a small script continuously creating a 50MB file on the disk, sleeping some, and then recreating it indefinitely. To see it from your VM, run the following command in a terminal:

```
cat ~/src/examples/poking-docker/experiment1/control/run.sh
```

You will see the following content, a simple bash script calling out to fallocate to create a file:

```
#! /bin/bash
FILESIZE=$((50*1024*1024))                                          #A
FILENAME=testfile                                                   #B
echo "Press [CTRL+C] to stop.."
while :
do
    fallocate -l $FILESIZE $FILENAME && echo "OK wrote the file" `ls -alhi $FILENAME` ||
       echo "Couldn't write the file"                              #C
    sleep 2
    rm $FILENAME || echo "Couldn't delete the file"
done
```

#A set the size of the file to 50MB in bytes
#B give the file we'll write a name
#C use fallocate to create a new file of the desired size, and print success or failure messages

I've also prepared a sample Dockerfile to build that script into a container. You can see it by running the following command in a terminal:

```
cat ~/src/examples/poking-docker/experiment1/control/Dockerfile
```

You will see the contents below. This very simple image starts from a base image of Ubuntu Focal, copies the script we've just seen, and sets that script as an entry point of the container, so that when we start the it later on, that script is run:

```
FROM ubuntu:focal-20200423                                    #A
COPY run.sh /run.sh                                           #B
ENTRYPOINT ["/run.sh"]                                        #C
```

#A start from base image ubuntu:focal-20200423
#B copy the script run.sh from the current working directory into the container
#C set our newly copied script as the entry point of the container

The Dockerfile is a recipe for building a container. With just these two files, we can now build our first image. You can do that by running the following command. Note, that Docker uses the current working directory to find files you point to in the Dockerfile, so we move to that directory first:

```
cd ~/src/examples/poking-docker/experiment1/control/
docker build \
-t experiment1-control \                                      #A
.                                                             #B
```

#A give the container we'll build a tag "experiment1-control"
#B use the Dockerfile in the current working directory

When you run this command, you will see the characteristic logs from Docker, in which it will pull the required base image from the Docker hub (separated in layers, the type we discussed earlier), and then run each command from Dockerfile. Each of these commands (or lines in Dockerfile) result in a new container. At the end, it will mark the last container with the tag we specified. You will see an output similar to the following:

```
Sending build context to Docker daemon  4.608kB
Step 1/3 : FROM ubuntu:focal-20200423
focal-20200423: Pulling from library/ubuntu                                  #A
d51af753c3d3: Pull complete
fc878cd0a91c: Pull complete
6154df8ff988: Pull complete
fee5db0ff82f: Pull complete
Digest: sha256:238e696992ba9913d24cfc3727034985abd136e08ee3067982401acdc30cbf3f
Status: Downloaded newer image for ubuntu:focal-20200423
 ---> 1d622ef86b13
Step 2/3 : COPY run.sh /run.sh                                               #B
 ---> 67549ea9de18
Step 3/3 : ENTRYPOINT ["/run.sh"]                                            #C
 ---> Running in e9b0ac1e77b4
Removing intermediate container e9b0ac1e77b4
 ---> c2829a258a07
Successfully built c2829a258a07
Successfully tagged experiment1-control:latest                               #D
```

#A pull the base image in the version (tag) we used as our base
#B copy the script run.sh into the container's filesystem
#C set the newly copied script as the entry point of the container
#D tag the built container

When that's finished, let's list the images available to Docker, which will now include our newly built image. You can list all tagged Docker images by running the following command in a terminal:

```
docker images
```

This will print an output similar to the following (abbreviated to just show your new image and its base):

```
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
(...)
experiment1-control   latest              c2829a258a07        6 seconds ago       73.9MB
ubuntu                focal-20200423      1d622ef86b13        4 days ago          73.9MB
```

If this is the first Docker image you've built yourself, congratulations! Now, to our failure container. In a similar fashion, I've prepared another script, which tries to create as many 50MB files as it can. You can see it by running the following command in the terminal:

```
cat ~/src/examples/poking-docker/experiment1/failure/consume.sh
```

You will see the following content, very similar to our previous script:

```
#! /bin/bash
FILESIZE=$((50*1024*1024))
FILENAME=testfile
echo "Press [CTRL+C] to stop.."
count=0
while :
do
    new_name=$FILENAME.$count
    fallocate -l $FILESIZE $new_name \                                          #A
        && echo "OK wrote the file" `ls -alhi $new_name` \                      #B
        || (echo "Couldn't write the file" $new_name "Sleeping a bit"; sleep 5) #C
    (( count++ ))
done
```

#A try to allocate a new file with a new name
#B on success print a message showing the new file
#C on failure print a failure message and sleep a few seconds

Similarly, I've also prepared a Dockerfile for building the failure container in the same folder (`~/src/examples/poking-docker/experiment1/failure/`) with the following contents:

```
FROM ubuntu:focal-20200423                                                      #A
COPY consume.sh /consume.sh                                                     #B
ENTRYPOINT ["/consume.sh"]                                                      #C
```

#A start from base image ubuntu:focal-20200423
#B copy the script consume.sh from the current working directory into the container
#C set our newly copied script as the entry point of the container

With that, we can go ahead and build the failure container by running the following command in a terminal window:

```
cd ~/src/examples/poking-docker/experiment1/failure/
docker build \
-t experiment1-failure \                                    #A
```

```
.                                                                    #B
```

#A give the container we'll build a tag "experiment1-failure"
#B use the Dockerfile in the current working directory

When that's done, let's list the images available again, by running the following command again in a terminal:

```
docker images
```

You will see an output similar to the following, once again abbreviated to only show the images relevant right now. Note both our control and failure containers are present:

```
REPOSITORY            TAG                 IMAGE ID            CREATED             SIZE
(...)
experiment1-failure   latest              001d2f541fb5        5 seconds ago       73.9MB
experiment1-control   latest              c2829a258a07        28 minutes ago      73.9MB
ubuntu                focal-20200423      1d622ef86b13        4 days ago          73.9MB
```

That's all we're going to need to conduct our experiment. Now, let's prepare two terminal windows, preferably side by side, so that we can see what's happening in each window at the same time. In the first window, let's run our control container. You can do that by issuing the following command:

```
docker run --rm -ti experiment1-control
```

You should see the container starting and printing a message confirming it's able to write every couple of seconds, just like the following:

```
Press [CTRL+C] to stop..
OK wrote the file 919053 -rw-r--r-- 1 root root 50M Apr 28 09:13 testfile
OK wrote the file 919053 -rw-r--r-- 1 root root 50M Apr 28 09:13 testfile
OK wrote the file 919053 -rw-r--r-- 1 root root 50M Apr 28 09:13 testfile
(...)
```

That confirms our steady state: we are able to continuously write a 50MB file to disk. Now, in the second window, let's start our failure container. You can do that by running the following command from the second terminal window:

```
docker run --rm -ti experiment1-failure
```

You will see an output similar to the following. For a few seconds, the container will be successful in writing the files, until it runs out of space and starts failing:

```
Press [CTRL+C] to stop..
OK wrote the file 919078 -rw-r--r-- 1 root root 50M Apr 28 09:21 testfile.0
OK wrote the file 919079 -rw-r--r-- 1 root root 50M Apr 28 09:21 testfile.1
(...)
OK wrote the file 919553 -rw-r--r-- 1 root root 50M Apr 28 09:21 testfile.475
fallocate: fallocate failed: No space left on device
Couldn't write the file testfile.476 Sleeping a bit
```

At the same time, in the first window, you will start seeing your control container failing with a message similar to the following:

```
(...)
```

```
OK wrote the file 919053 -rw-r--r-- 1 root root 50M Apr 28 09:21 testfile
OK wrote the file 919053 -rw-r--r-- 1 root root 50M Apr 28 09:21 testfile
fallocate: fallocate failed: No space left on device
Couldn't write the file
```

This confirms our hypothesis: one container can use up the space that another container would like to use in our environment. In fact, if you investigate the disk usage in our VM while the two containers are still running, you will see that the main disk is now 100% full. You can do that by running the following command in another terminal:

```
df -h
```

You will see an output similar to the following (utilization of our main disk in bold font):

```
Filesystem      Size  Used Avail Use% Mounted on
udev            2.0G     0  2.0G   0% /dev
tmpfs           395M  7.8M  387M   2% /run
/dev/sda1        32G   32G     0 100% /
(...)
```

If you now stop the failure container by pressing Ctrl-C in its window, you will see its storage removed (thanks to the `--rm` option we started it), and in the first window, the control container will resume happily re-writing its file.

The takeaway here is that running programs in containers doesn't automatically prevent one process from stealing disk space from another. Fortunately, the authors of Docker thought about that, and exposed a flag called `--storage-opt size=X`. Unfortunately, when using the overlay2 storage driver, this option requires using an `xfs` filesystem with `pquota` option as the host filesystem (at least for the location where Docker stores its container data, which defaults to `/var/lib/docker`), which our VM running on default settings is not doing.

Therefore, it will require an extra effort to allow Docker containers to be limited in storage, which means that there is a good potential that many systems will not limit it at all. The storage driver setup requires a careful consideration and will be important to the overall health of your systems.

Keeping that in mind, let's take a look at the next building block of a Docker container - the Linux namespaces.

5.3.4 Isolating processes with Linux namespaces

Namespaces are a feature of the Linux kernel that controls what subset of resources is visible to processes associated with a particular namespace. You can think of them as filters, which control what a process can see. For example, as figure 5.7 illustrates, a resource can be visible to zero or more namespaces. But if it's not visible to the namespace, the kernel will make it look like it doesn't exist from the perspective of a process in that namespace.

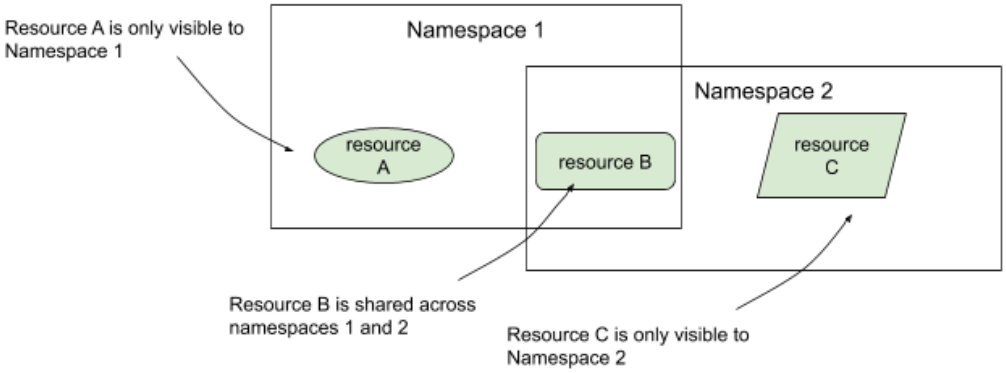


Figure 5.7 High-level idea of namespaces

Namespaces are a crucial part of the Linux container solutions, including Docker. Different types of namespaces deal with different resources. At the time of writing, the following namespaces are available:

- Mounts (**mnt**) - controls which mounts are accessible within the namespace
- Process ID (**pid**) - creates an independent set of PIDs for processes within the namespace
- Network (**net**) - virtualizes the network stack, allows for network interfaces (physical or virtual) to be attached to network namespaces
- Interprocess Communication (**ipc**) - isolate objects used for interprocess communication - System V IPC and POSIX message queues (http://man7.org/linux/man-pages/man7/ipc_namespaces.7.html)
- UTS (**uts**) - allows for different host and domain names in different namespaces
- User ID (**user**) - user identification and privilege isolation per namespace
- Control group (**cname**) - hides the real identity of the control group the processes are a member of
- Time (**time**) - shows different times for different namespaces

> **NOTE** Time namespace was only introduced in version 5.6 of the Linux kernel a few weeks ago. Our VM, running kernel 4.18, doesn't have it yet.

By default, Linux starts with a single namespace of each type and new namespaces can be created on the fly. You can list existing namespaces using the command `lsns`. Type the following command in a terminal window to print the available namespaces:

```
lsns
```

You will see an output similar to the following. Note that the command column, as well as PID, applies to the lowest PID that was started in that namespace. NPROCS shows the number of processes currently running in the namespace (from current user perspective).

```
        NS TYPE   NPROCS   PID USER  COMMAND
4026531835 cgroup     69  2217 chaos /lib/systemd/systemd --user
4026531836 pid        69  2217 chaos /lib/systemd/systemd --user
4026531837 user       69  2217 chaos /lib/systemd/systemd --user
4026531838 uts        69  2217 chaos /lib/systemd/systemd --user
4026531839 ipc        69  2217 chaos /lib/systemd/systemd --user
4026531840 mnt        69  2217 chaos /lib/systemd/systemd --user
4026531993 net        69  2217 chaos /lib/systemd/systemd --user
```

Note, that if you rerun the same command as the root user, you will see a larger set of namespaces, which are created by various components of the system. You can do that by running the following command in a terminal window:

```
sudo lsns
```

You will see an output similar to the following. The important thing to note, is that while there are other namespaces, the ones we saw previously are the same (they have a matching number in the column NS), although the number of processes and the lowest PID are different. In fact, you can see the PID of 1, the first process started on the host. By default, all users are sharing the same namespaces. I used a bold font to point out the repeated ones.

```
        NS TYPE   NPROCS   PID USER            COMMAND
4026531835 cgroup    211     1 root            /sbin/init
4026531836 pid       210     1 root            /sbin/init
4026531837 user      211     1 root            /sbin/init
4026531838 uts       210     1 root            /sbin/init
4026531839 ipc       210     1 root            /sbin/init
4026531840 mnt       200     1 root            /sbin/init
4026531861 mnt         1    19 root            kdevtmpfs
4026531993 net       209     1 root            /sbin/init
4026532148 mnt         1   253 root            /lib/systemd/systemd-udevd
4026532158 mnt         1   343 systemd-resolve /lib/systemd/systemd-resolved
4026532170 mnt         1   461 root            /usr/sbin/ModemManager --filter-
       policy=strict
4026532171 mnt         2   534 root            /usr/sbin/NetworkManager --no-daemon
4026532238 net         1  1936 rtkit           /usr/lib/rtkit/rtkit-daemon
4026532292 mnt         1  1936 rtkit           /usr/lib/rtkit/rtkit-daemon
4026532349 mnt         1  2043 root            /usr/lib/x86_64-linux-gnu/boltd
4026532350 mnt         1  2148 colord          /usr/lib/colord/colord
4026532351 mnt         1  3061 root            /usr/lib/fwupd/fwupd
```

`lsns` is pretty neat. It can do things like print out JSON (`--json` flag, good for consumption in scripts), only look into a particular type of namespace (`--type` flag), or give you the namespaces for a particular PID (`--task` flag). Under the hood, it reads from the `/proc` filesystem exposed by the Linux kernel -- in particular, from `/proc/<pid>/ns` - a location that's good to know your way around.

To see what namespaces a particular process is in, you just need its PID. For the current bash session, we can access it via `$$`. That means that we can check the namespaces our bash session is in by running the following command in a terminal window:

```
ls -l /proc/$$/ns
```

You will see an output similar to the following. For each type of namespace we just covered, you will see a symbolic link:

```
total 0
lrwxrwxrwx 1 chaos chaos 0 May  1 09:38 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 chaos chaos 0 May  1 09:38 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 chaos chaos 0 May  1 09:38 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 chaos chaos 0 May  1 09:38 net -> 'net:[4026531993]'
lrwxrwxrwx 1 chaos chaos 0 May  1 09:38 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 chaos chaos 0 May  1 10:11 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 chaos chaos 0 May  1 09:38 user -> 'user:[4026531837]'
lrwxrwxrwx 1 chaos chaos 0 May  1 09:38 uts -> 'uts:[4026531838]'
```

These symbolic links are a little bit special. Try to probe them with the `file` utility by running the following command in your terminal:

```
file /proc/$$/ns/pid
```

You will see an output similar to the following. It will complain that the symbolic link is broken.

```
/proc/3391/ns/pid: broken symbolic link to pid:[4026531836]
```

That's because the links have a special format: `<namespace type>:[<namespace number>]`. You can read the value of the link using `readlink`, by running the following command in the terminal:

```
readlink /proc/$$/ns/pid
```

You will see an output similar to the following. It's a namespace of type `pid` with the number `4026531836`. It's the same one we saw in the output of `lsns` earlier:.

```
pid:[4026531836]
```

Now you know what namespaces are, what kinds are available and how to see what processes belong to which namespaces. Let's take a look at how Docker uses them.

5.3.5 Docker and namespaces

To see how Docker manages container namespaces, let's start a fresh container. You can do that by running the following command in a terminal window. Note, that I'm using again a particular tag of the Ubuntu Focal image, just so that we use the exact same environment.

```
docker run \
--name probe \                                  #A
-ti \                                           #B
--rm \                                          #C
ubuntu:focal-20200423                           #D
```

#A give our container a name
#B keep STDIN open and allocate a pseudo-TTY to allow us to type commands
#C remove the container after we're done with it
#D run the same Ubuntu image we used earlier

You will enter into an interactive bash session in a new container. You can confirm that by checking the contents of `/etc/issue` like we did earlier in the chapter.

Now, let's see what namespaces Docker created for us. To do that, open a second terminal window, and let's inspect our Docker container. First, let's see the list of running containers, by executing the following command in the second terminal:

```
docker ps
```

You will see an output similar to the following one. We are interested in the container ID (in bold font) of the container we just started (we named it "probe").

```
CONTAINER ID   IMAGE          COMMAND      CREATED     STATUS      PORTS            NAMES
91d17914dd23     ubuntu:focal-20200423   "/bin/bash"          48 seconds ago      Up 47
       seconds              probe
```

Knowing its ID, let's inspect that container. Run the following command, still in the second terminal window, replacing the ID with the one you saw:

```
docker inspect 91d17914dd23
```

The output you see will be pretty long, but for now I'd like you to just focus on the `State` part of it, which will look similar to the following output. In particular, note the `Pid` (in bold font):

```
(...)
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 3603,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2020-05-01T09:38:03.245673144Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
(...)
```

With that PID, we can list the namespaces the container is in by running the following command in the second terminal, replacing the PID with the value from your system (in bold font). Note, that we are going to need to use `sudo`, to access namespace data for a process the current user doesn't own:

```
sudo ls -l /proc/3603/ns
```

In the following output, you will see a few new namespaces, but not all of them.

```
total 0
lrwxrwxrwx 1 root root 0 May  1 09:38 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 May  1 09:38 ipc -> 'ipc:[4026532357]'
lrwxrwxrwx 1 root root 0 May  1 09:38 mnt -> 'mnt:[4026532355]'
lrwxrwxrwx 1 root root 0 May  1 09:38 net -> 'net:[4026532360]'
lrwxrwxrwx 1 root root 0 May  1 09:38 pid -> 'pid:[4026532358]'
```

```
lrwxrwxrwx 1 root root 0 May  1 10:04 pid_for_children -> 'pid:[4026532358]'
lrwxrwxrwx 1 root root 0 May  1 09:38 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 May  1 09:38 uts -> 'uts:[4026532356]'
```

You can match this output to the previous one to see which namespaces were created for the process, but that sounds laborious to me. Alternatively, you can also leverage the `lsns` command to give you an output that's easier to read. Run the following command in the same terminal window (again, changing the value of the PID):

```
sudo lsns --task 3603
```

You can see clearly the new namespaces in the output (where the lowest PID is the one we are looking for - I used bold font to make them stand out):

```
        NS TYPE   NPROCS   PID USER COMMAND
4026531835 cgroup    210     1 root /sbin/init
4026531837 user      210     1 root /sbin/init
4026532355 mnt         1  3603 root /bin/bash
4026532356 uts         1  3603 root /bin/bash
4026532357 ipc         1  3603 root /bin/bash
4026532358 pid         1  3603 root /bin/bash
4026532360 net         1  3603 root /bin/bash
```

You can now kill that container, for example by pressing Ctrl-D in the first window, because we won't be needing it any more.

So Docker created a new namespace of each type, except for `cgroup` and `user` (we'll cover the former later in this chapter). In theory then, from inside of the container, we should be isolated from the host system in all aspects covered by the new namespaces. In practice, theory is often different from practice, so let's do what any self-proclaimed scientist should do - let's experiment and see how isolated we really are. Since we spoke a bit about PIDs, let's pick the **pid** namespace for the experiment.

5.3.6 Experiment 2: killing processes in a different pid namespace

A fun experiment to confirm that the **pid** namespaces work (and that we understand how they're supposed to work!) is to start a container, and try to kill a PID from outside of its namespace. Observing it will be trivial (either the process got killed or not), and our expectation is that it should not work. The whole experiment can be summarized in the following four steps:

1. Observability: "checking if the process is still running"
2. Steady state: "the process is running"
3. Hypothesis: "if we issue a kill command from inside of the container, for a process outside of the container, it should fail"
4. Run the experiment!

Easy peasy. To implement that, we'll need a practice target to kill. I've prepared one for you. You can see it by running the following command in a terminal window of your VM:

```
cat ~/src/examples/poking-docker/experiment2/pid-printer.sh
```

You will see the following output. It doesn't get much more basic than this:

```
#! /bin/bash
echo "Press [CTRL+C] to stop.."
while :
do
    echo `date` "Hi, I'm PID $$ and I'm feeling sleeeeeepy..." && sleep 2      #A
done
```

#A prints a message, include its PID number, and sleeps

To run our experiment, we will use two terminal windows. In the first one, we'll run the target we're trying to kill, and in the second one the container from which we'll issue the kill command. Let's start this script by running the following command in the first terminal window:

```
bash ~/src/examples/poking-docker/experiment2/pid-printer.sh
```

You will see an output similar to the following one, with the process printing its PID every few seconds. I used a bold font for the PID - copy it:

```
Press [CTRL+C] to stop..
Fri May 1 06:15:22 UTC 2020 Hi, I'm PID 9000 and I'm feeling sleeeeeepy...
Fri May 1 06:15:24 UTC 2020 Hi, I'm PID 9000 and I'm feeling sleeeeeepy...
Fri May 1 06:15:26 UTC 2020 Hi, I'm PID 9000 and I'm feeling sleeeeeepy...
```

Now, let's start a new container in a second terminal window. Start a new window, and run the following command:

```
docker run \
--name experiment2 \                                #A
-ti \                                               #B
--rm \                                              #C
ubuntu:focal-20200423                               #D
```

#A give our container a name
#B keep STDIN open and allocate a pseudo-TTY to allow us to type commands
#C remove the container after we're done with it
#D run the same Ubuntu image we used earlier

It looks like we're all set! From inside of the container (the second terminal window), let's try to kill the PID that our target keeps printing. You can do that, by running the following command (replace the PID with your value):

```
kill -9 9000
```

You will see in the output that it did not find such a process.

```
bash: kill: (9000) - No such process
```

We can confirm that in the first window our target is still running, which means that our experiment confirmed our hypothesis - trying to kill a process running outside a container's PID namespace did not work. But the error message we saw indicated that from inside the container, there was no process with a PID like that. Let's see what processes are listed from

inside of the container. You can do that by running the following command from the second terminal window:

```
ps a
```

You will see an output like the following one. Note, that only two processes are listed:

```
  PID TTY      STAT   TIME COMMAND
    1 pts/0    Ss     0:00 /bin/bash
   10 pts/0    R+     0:00 ps a
```

So as far as processes inside of this container are concerned, there is no PID 9000. Or anything greater than 9000. We are now done with the experiment, but I'm sure you're now curious whether we could somehow enter the namespace of the container and start a process in there. The answer is yes.

To start a new process inside of the existing container's namespace we can use the `nsenter` command. It allows you to start a new process inside of any of the namespaces on the host. Let's use that to attach to our container's PID namespace. I've prepared a little script for you. You can see it by running the following command inside of a new terminal window (a third one):

```
cat ~/src/examples/poking-docker/experiment2/attach-pid-namespace.sh
```

You will see the following output, showcasing how to use the `nsenter` command:

```
#! /bin/bash
CONTAINER_PID=$(docker inspect -f '{{ .State.Pid }}' experiment2)                    #A
sudo nsenter \
    --pid \                                                                          #B
    --target $CONTAINER_PID \                                                        #C
    /bin/bash /home/chaos/src/examples/poking-docker/experiment2/pid-printer.sh      #D
```

#A get the PID of our container from `docker inspect`
#B enter the pid namespace
#C of the specified process with the given PID
#D execute the same bash script we previously ran from the common namespace

Run the script with the following command:

```
bash ~/src/examples/poking-docker/experiment2/attach-pid-namespace.sh
```

You will see a familiar output, similar to the following:

```
Press [CTRL+C] to stop..
Fri May 1 12:02:04 UTC 2020 Hi, I'm PID 15 and I'm feeling sleeeeeepy...
```

To confirm that we're in the same namespace, let's run `ps` again from inside of the container (second terminal window):

```
ps a
```

You will now see an output similar to the following, including our newly started script:

```
  PID TTY      STAT   TIME COMMAND
    1 pts/0    Ss     0:00 /bin/bash
```

```
   15 ?        S+     0:00 /bin/bash /home/chaos/src/examples/poking-
      docker/experiment2/pid-printer.sh
  165 ?        S+     0:00 sleep 2
  166 pts/0    R+     0:00 ps a
```

Finally, it's useful to know that the ps command supports printing namespaces too. You can add them by listing the desired namespaces in the `-o` flag. For example, to show the PID namespaces for processes on the host, run the following command from the first terminal window (from the host, not the container):

```
ps ao pid,pidns,command
```

You will see the PID namespaces along with the PID and command, similar to the following output:

```
  PID      PIDNS COMMAND
(...)
 3505 4026531836 docker run --name experiment2 -ti --rm ubuntu:focal-20200423
 4012 4026531836 bash /home/chaos/src/examples/poking-docker/experiment2/attach-pid-
      namespace.sh
 4039 4026531836 bash
 4087 4026531836 ps o pid,pidns,command
```

> **NOTE**: if you'd like to learn how to see the other namespaces a process belongs to, run the command `man ps`. If you're not on Linux, `man` stands for "manual" and is a Linux command displaying help for different commands and system components. To use it, simply type `man` followed by the name of the item you're interested in (like `man ps` above) to display help directly from the terminal. You can learn more at https://www.kernel.org/doc/man-pages/

As you can see, PID namespaces are an efficient and simple-to-use way of tricking an application that it's the only thing running on the host and isolate it from seeing other processes at all. You're probably itching now to play around with it. And because I strongly believe playing is the best way to learn, let's add namespaces to our simple container(-ish) we started on in section 5.2.2.

5.3.7 Implementing a simple container(-ish) part 2 - namespaces

Time to upgrade our DIY container by leveraging what we've just learned - the Linux kernel namespaces. To refresh your memory on where it fits, take a look at figure 5.8. We'll pick a single namespace - PID - to keep things simple and to make for nice demos.

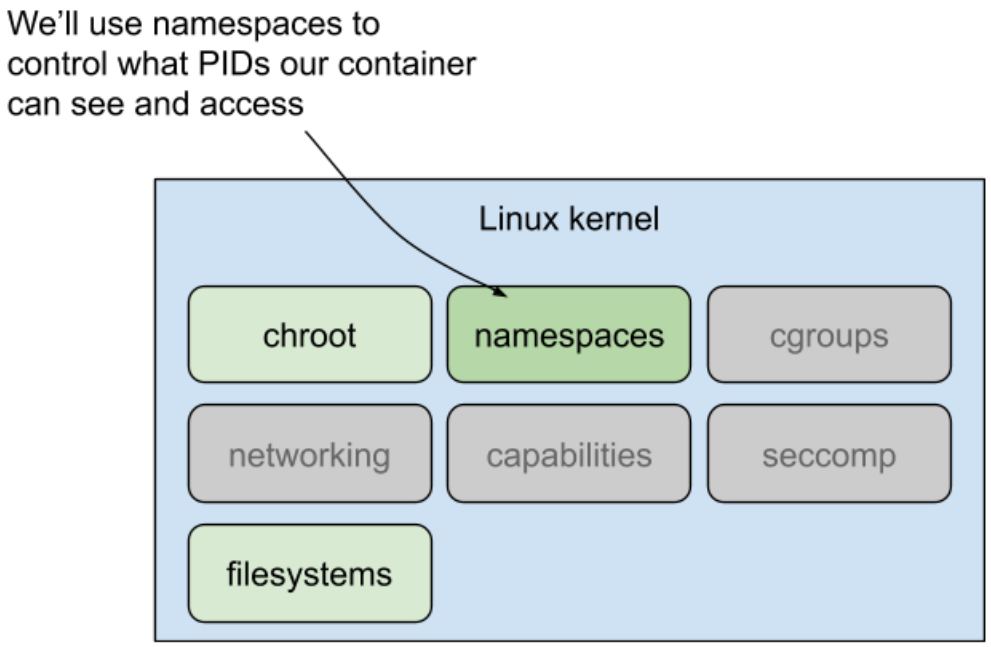


Figure 5.8 DIY container part 2 - namespaces

In section 5.2.2 we used `chroot` to change the root mount from a process' perspective to a subfolder we've prepared that contained some basic structure of a Linux system. Let's leverage that script and now add a separate PID namespace. To create new namespaces and start processes in them, we can use the command `unshare`.

The syntax of `unshare` is straightforward: `unshare [options] [program [arguments]]`. It even comes with a useful example in its man pages (run `man unshare` in a terminal to display it), which shows us how to start a process in a new PID namespace. For example, if you wanted to start a new bash session, you can run the following command in a new terminal window:

```
sudo unshare --fork --pid --mount-proc /bin/bash
```

You will see a new bash session in a new PID namespace. To see what PID your bash (thinks it) has, run the following command in that new bash session:

```
ps
```

You will see an output similar to the following one. The bash command displays a PID of 1.

```
  PID TTY          TIME CMD
    1 pts/3    00:00:00 bash
   18 pts/3    00:00:00 ps
```

Now, we can put together `unshare` and `chroot` (from section 5.2.2) to get closer to a real Linux container. I've prepared a script that does that for your convenience. You can see it by running the following command in a terminal window of your VM:

```
cat ~/src/examples/poking-docker/container-ish.sh
```

You will see the following output. It's a very basic script with essentially two important steps:

- First, calling the previous `new-filesystem.sh` script to create our structure and copy some tools over to it.
- Second, call the `unshare` command with `--pid` flag, which calls `chroot`, which in turn calls `bash`. The `bash` program starts by mounting `/proc` from inside of the container and then starts an interactive session.

```
#! /bin/bash
CURRENT_DIRECTORY="$(dirname "${0}")"
FILESYSTEM_NAME=${1:-container-attempt-2}

# Step 1: execute our familiar new-filesystem script
bash $CURRENT_DIRECTORY/new-filesystem.sh $FILESYSTEM_NAME                              #A
cd $FILESYSTEM_NAME

# Step 2: create a new pid namespace, and start a chrooted bash session
sudo unshare \                                                                          #B
    --fork \                                                                            #C
    --pid \                                                                             #D
    chroot . \                                                                          #E
    /bin/bash -c "mkdir -p /proc && /bin/mount -t proc proc /proc && exec /bin/bash"    #F
```

#A run the `new-filesystem.sh` script which copies some basic binaries and their libraries
#B the unshare command starts a process in a different namespace
#C forking is required for pid namespaces change to work
#D create a new pid namespace for the new process
#E call chroot to change the root of the filesystem for the new process we start
#F mount /proc from inside of the container (for example, to make ps work) and run bash

Let's use that script by running the following command in a new terminal window. Note that it will create a folder for the container(-ish) in the current directory:

```
bash ~/src/examples/poking-docker/container-ish.sh a-bit-closer-to-docker
```

You will see the greetings and a new bash session. To confirm that we successfully created a new namespace, let's see the output of `ps`. Run the following command from inside of your new bash session:

```
ps aux
```

It will print the following list. Note, that our bash claims to have the PID of 1 (bold font).

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
0            1  0.0  0.0  10052  3272 ?        S    11:54   0:00 /bin/bash
0            4  0.0  0.0  25948  2568 ?        R+   11:55   0:00 ps aux
```

Finally, while your kind-of-container is still running, let's open another terminal window and confirm that we can see the new namespace of type PID, by running the following command:

```
sudo lsns -t pid
```

You will see an output similar to the following. I used a bold font to emphasise the new namespace.

```
        NS TYPE NPROCS   PID USER COMMAND
4026531836 pid     211     1 root /sbin/init
4026532173 pid       1 24935 root /bin/bash
```

Note that as we've seen before, Docker creates other types of namespaces for its containers, not just PID. In our example we focus on the PID, because it's easy to demonstrate and helps with learning. I'm leaving tinkering with the other ones as an exercise to the reader.

Having demystified namespaces, let's now move on to the next piece of the puzzle. Let's take a look at how Docker restricts the amount of resources containers can use through `cgroups`.

5.3.8 Limiting resource use of a process with cgroups

Control groups, or `cgroups` for short, are a feature of the Linux kernel that allows for organizing processes into hierarchical groups and then limiting and monitoring their usage of various types of resources, such as CPU and RAM. It allows you, for example, to tell the Linux kernel to only give a certain percentage of CPU to a particular process. Figure 5.9 illustrates what limiting a process to 50% of a core looks like visually. On the left hand side, the process is allowed to use as much CPU as there is available. On the right hand side, a limit of 50% is enforced, and the process is throttled if it ever tries to use more than 50%.

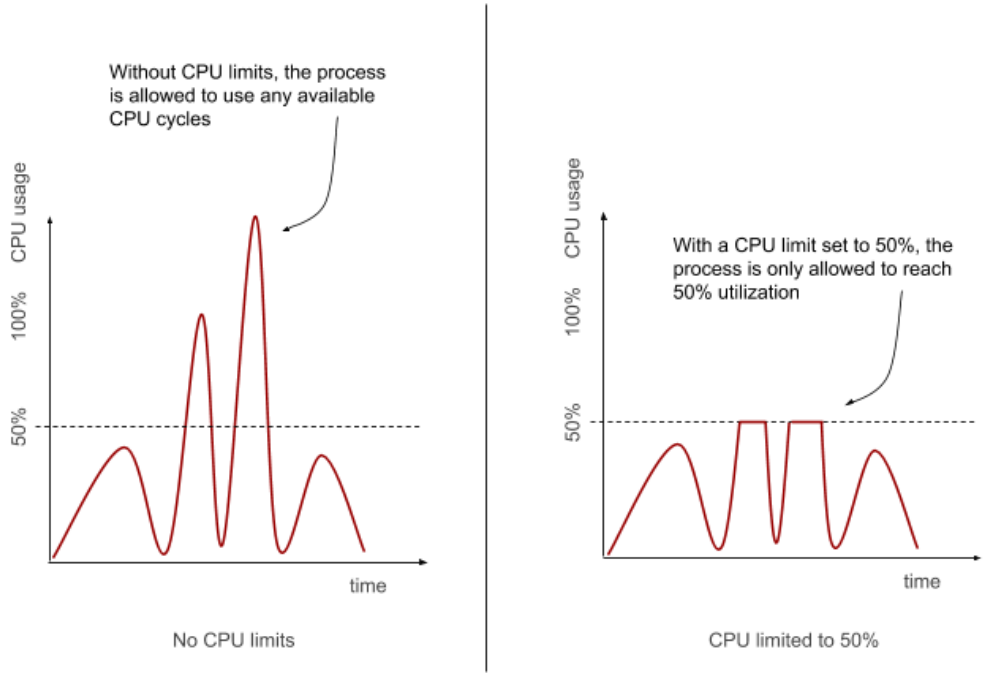


Figure 5.9 An example of CPU limiting possible with cgroups

How do you interact with cgroups? Kernel exposes a pseudo-filesystem called `cgroupfs` for managing the cgroups hierarchy, usually mounted at `/sys/fs/cgroup`.

> **NOTE** There are currently two versions of cgroups available, v1 and v2. V1 evolved over the years in a mostly uncoordinated, organic fashion and v2 was introduced to reorganise, simplify and remove some of the inconsistencies in the v1. At the time of writing, most of the ecosystem still uses v1, or at least defaults to it, while support for v2 is being worked on (for example the work for Docker via runc is tracked in this issue https://github.com/opencontainers/runc/issues/2315). You can read more about the differences between them at http://man7.org/linux/man-pages/man7/cgroups.7.html. We'll stick to v1 for the time being.

Cgroups have the concept of a controller for each type of supported resources. To check the currently mounted and available types of controllers, run the following command in a terminal inside of your VM:

```
ls -al /sys/fs/cgroup/
```

You will see an output similar to the following one. We are going to cover two controllers: cpu and memory (in bold font). Note that `cpu` is actually a link to `cpu,cpuacct`, a controller responsible for both limiting and accounting for cpu usage. Also, `unified` is where `groups` v2 are mounted, if you're curious to play with that as an exercise.

```
total 0
drwxr-xr-x 15 root root 380 May  2 14:23 .
drwxr-xr-x  9 root root   0 May  3 12:26 ..
dr-xr-xr-x  5 root root   0 May  3 12:26 blkio
lrwxrwxrwx  1 root root  11 May  2 14:23 cpu -> cpu,cpuacct
lrwxrwxrwx  1 root root  11 May  2 14:23 cpuacct -> cpu,cpuacct
dr-xr-xr-x  5 root root   0 May  3 12:26 cpu,cpuacct
dr-xr-xr-x  3 root root   0 May  3 12:26 cpuset
dr-xr-xr-x  5 root root   0 May  3 12:26 devices
dr-xr-xr-x  3 root root   0 May  3 12:26 freezer
dr-xr-xr-x  3 root root   0 May  3 12:26 hugetlb
dr-xr-xr-x  5 root root   0 May  3 12:26 memory
lrwxrwxrwx  1 root root  16 May  2 14:23 net_cls -> net_cls,net_prio
dr-xr-xr-x  3 root root   0 May  3 12:26 net_cls,net_prio
lrwxrwxrwx  1 root root  16 May  2 14:23 net_prio -> net_cls,net_prio
dr-xr-xr-x  3 root root   0 May  3 12:26 perf_event
dr-xr-xr-x  5 root root   0 May  3 12:26 pids
dr-xr-xr-x  2 root root   0 May  3 12:26 rdma
dr-xr-xr-x  6 root root   0 May  3 12:26 systemd
dr-xr-xr-x  5 root root   0 May  3 12:26 unified
```

You might recall from section 3.3.5.3 two tools we can use to create cgroups and run programs within them: `cgcreate` and `cgexec`. These are convenient to use, but I'd like to show you how to interact with the `cgroupfs` directly. When practicing chaos engineering on systems leveraging Docker, it will be essential we understand and can observe the limits our applications are running with.

Creating a new cgroup of a particular type consists of creating a folder (or subfolder for nested cgroups) under `/sys/fs/cgroup/<type of the resource>/`. For example, Docker creates its parent cgroup, under which the containers are then nested. Let's take a look at the contents of the CPU cgroup. You can do that by running the following command in a terminal window:

```
ls -l /sys/fs/cgroup/cpu/docker
```

You will see a list just like the following one. For our needs, we'll pay attention to **`cpu.cfs_period_us`**, **`cpu.cfs_quota_us`** and **`cpu.shares`** which represent two ways cgroups offer to restrict CPU utilization of a process.

```
-rw-r--r-- 1 root root 0 May  3 12:44 cgroup.clone_children
-rw-r--r-- 1 root root 0 May  3 12:44 cgroup.procs
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.stat
-rw-r--r-- 1 root root 0 May  3 12:44 cpuacct.usage
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_all
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_percpu
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_percpu_sys
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_percpu_user
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_sys
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_user
-rw-r--r-- 1 root root 0 May  3 12:44 cpu.cfs_period_us
-rw-r--r-- 1 root root 0 May  3 12:44 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 May  3 12:44 cpu.shares
-r--r--r-- 1 root root 0 May  3 12:44 cpu.stat
-rw-r--r-- 1 root root 0 May  3 12:44 notify_on_release
-rw-r--r-- 1 root root 0 May  3 12:44 tasks
```

The first way is to set exactly the ceiling of how many microseconds of CPU time a particular process can get within a particular period of time. This is done by specifying the values for `cpu.cfs_period_us` (the period in microseconds) and `cpu.cfs_quota_us` (the number of microseconds within that period that it can consume.) For example, to allow a particular process to consume 50% of a CPU, we could give `cpu.cfs_period_us` a value of `1000` and `cpu.cfs_quota_us` the value of `500`. A value of -1 means no limitation and it's the default. It's a hard limit.

The other way is through CPU shares (`cpu.shares`). The shares are arbitrary values representing a relative weight of the process. Thus, the same value means the same amount of CPU for every process, a higher value will increase the percentage of available time a process is allowed, and a lower value will decrease it. It defaults to a rather arbitrary, round number of 1024. It's worth noting, that it's only enforced when there isn't enough CPU time for everyone, otherwise it has no effect. It's essentially a soft limit.

Now, let's see what Docker sets up for a new container. Let's start a container, by running the following command in a terminal window:

```
docker run -ti --rm ubuntu:focal-20200423
```

Once inside of the container, let's start a long-running process, so that we can identify it easily later. Run the following command from inside of the container to start a `sleep` process (doing nothing but existing) for 3600 seconds:

```
sleep 3600
```

While that container is running, let's use another terminal window to check again the cgroupfs folder that Docker maintains. Run the following command in that second terminal window:

```
ls -l /sys/fs/cgroup/cpu/docker
```

You will see a familiar output, just like the following one. Note, that there is a new folder there with a name corresponding to the container ID (in bold font):

```
total 0
drwxr-xr-x 2 root root 0 May  3 22:21
       87a692e9f2b3bac1514428954fd2b8b80c681012d92d5ae095a10f81fb010450
-rw-r--r-- 1 root root 0 May  3 12:44 cgroup.clone_children
-rw-r--r-- 1 root root 0 May  3 12:44 cgroup.procs
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.stat
-rw-r--r-- 1 root root 0 May  3 12:44 cpuacct.usage
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_all
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_percpu
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_percpu_sys
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_percpu_user
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_sys
-r--r--r-- 1 root root 0 May  3 12:44 cpuacct.usage_user
-rw-r--r-- 1 root root 0 May  3 12:44 cpu.cfs_period_us
-rw-r--r-- 1 root root 0 May  3 12:44 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 May  3 12:44 cpu.shares
-r--r--r-- 1 root root 0 May  3 12:44 cpu.stat
-rw-r--r-- 1 root root 0 May  3 12:44 notify_on_release
-rw-r--r-- 1 root root 0 May  3 12:44 tasks
```

To make things easier, let's just store that long container ID in an environment variable. Do that by running the following command:

```
export CONTAINER_ID=87a692e9f2b3bac1514428954fd2b8b80c681012d92d5ae095a10f81fb010450
```

Now, list the contents of that new folder by running the following command:

```
ls -l /sys/fs/cgroup/cpu/docker/$CONTAINER_ID
```

You will see an output similar to the following one, with the now familiar structure. This time I would like you to pay attention to `cgroup.procs` (in bold font), which holds a list of PIDs of processes within this cgroup.

```
total 0
-rw-r--r-- 1 root root 0 May  3 22:43 cgroup.clone_children
-rw-r--r-- 1 root root 0 May  3 22:21 cgroup.procs
-r--r--r-- 1 root root 0 May  3 22:43 cpuacct.stat
-rw-r--r-- 1 root root 0 May  3 22:43 cpuacct.usage
-r--r--r-- 1 root root 0 May  3 22:43 cpuacct.usage_all
-r--r--r-- 1 root root 0 May  3 22:43 cpuacct.usage_percpu
-r--r--r-- 1 root root 0 May  3 22:43 cpuacct.usage_percpu_sys
-r--r--r-- 1 root root 0 May  3 22:43 cpuacct.usage_percpu_user
-r--r--r-- 1 root root 0 May  3 22:43 cpuacct.usage_sys
-r--r--r-- 1 root root 0 May  3 22:43 cpuacct.usage_user
-rw-r--r-- 1 root root 0 May  3 22:43 cpu.cfs_period_us
-rw-r--r-- 1 root root 0 May  3 22:43 cpu.cfs_quota_us
-rw-r--r-- 1 root root 0 May  3 22:43 cpu.shares
-r--r--r-- 1 root root 0 May  3 22:43 cpu.stat
-rw-r--r-- 1 root root 0 May  3 22:43 notify_on_release
-rw-r--r-- 1 root root 0 May  3 22:43 tasks
```

Let's investigate what the processes contained in that `cgroup.procs` file are. You can do that by running the following command in a terminal window:

```
ps -p $(cat /sys/fs/cgroup/cpu/docker/$CONTAINER_ID/cgroup.procs)
```

You will see the container's bash session, as well as the sleep we started earlier, just like the following:

```
  PID TTY      STAT   TIME COMMAND
28960 pts/0    Ss     0:00 /bin/bash
29199 pts/0    S+     0:00 sleep 3600
```

Let's also check what the default values our container got started with are. In the same subdirectory, you will see the following default values. They mean no hard limit and the default weight:

- `cpu.cfs_period_us` - set to 100000
- `cpu.cfs_quota_us` - set to -1
- `cpu.shares` - set to 1024

Similarly, we can peek into what default values are set in terms of memory usage. To do that, let's explore the memory part of the tree, by running the following command:

```
ls -l /sys/fs/cgroup/memory/docker/$CONTAINER_ID/
```

It will print a list similar to the following. Note the `memory.limit_in_bytes` (which sets the hard limit of RAM accessible to the process) and `memory.usage_in_bytes` (which shows the current RAM utilization):

```
total 0
-rw-r--r-- 1 root root 0 May  3 23:04 cgroup.clone_children
--w--w--w- 1 root root 0 May  3 23:04 cgroup.event_control
-rw-r--r-- 1 root root 0 May  3 22:21 cgroup.procs
-rw-r--r-- 1 root root 0 May  3 23:04 memory.failcnt
--w------- 1 root root 0 May  3 23:04 memory.force_empty
-rw-r--r-- 1 root root 0 May  3 23:04 memory.kmem.failcnt
-rw-r--r-- 1 root root 0 May  3 23:04 memory.kmem.limit_in_bytes
-rw-r--r-- 1 root root 0 May  3 23:04 memory.kmem.max_usage_in_bytes
-r--r--r-- 1 root root 0 May  3 23:04 memory.kmem.slabinfo
-rw-r--r-- 1 root root 0 May  3 23:04 memory.kmem.tcp.failcnt
-rw-r--r-- 1 root root 0 May  3 23:04 memory.kmem.tcp.limit_in_bytes
-rw-r--r-- 1 root root 0 May  3 23:04 memory.kmem.tcp.max_usage_in_bytes
-r--r--r-- 1 root root 0 May  3 23:04 memory.kmem.tcp.usage_in_bytes
-r--r--r-- 1 root root 0 May  3 23:04 memory.kmem.usage_in_bytes
-rw-r--r-- 1 root root 0 May  3 23:04 memory.limit_in_bytes
-rw-r--r-- 1 root root 0 May  3 23:04 memory.max_usage_in_bytes
-rw-r--r-- 1 root root 0 May  3 23:04 memory.move_charge_at_immigrate
-r--r--r-- 1 root root 0 May  3 23:04 memory.numa_stat
-rw-r--r-- 1 root root 0 May  3 23:04 memory.oom_control
---------- 1 root root 0 May  3 23:04 memory.pressure_level
-rw-r--r-- 1 root root 0 May  3 23:04 memory.soft_limit_in_bytes
-r--r--r-- 1 root root 0 May  3 23:04 memory.stat
-rw-r--r-- 1 root root 0 May  3 23:04 memory.swappiness
-r--r--r-- 1 root root 0 May  3 23:04 memory.usage_in_bytes
-rw-r--r-- 1 root root 0 May  3 23:04 memory.use_hierarchy
-rw-r--r-- 1 root root 0 May  3 23:04 notify_on_release
-rw-r--r-- 1 root root 0 May  3 23:04 tasks
```

If you check the contents of these two files, you will see the following values:

- `memory.limit_in_bytes` set to 9223372036854771712, which seems to be a max number for a 64-bit int, minus a page size, or effectively representing infinity
- `memory.usage_in_bytes` which happens to read 1445888 for me (or ~1.4MB)

Note, that although `memory.usage_in_bytes` is read-only, you can modify `memory.limit_in_bytes` by simply writing to it. For example, to impose a 20MB memory limit on our container, run the following command:

```
echo 20971520 | sudo tee /sys/fs/cgroup/memory/docker/$CONTAINER_ID/memory.limit_in_bytes
```

This covers what you need to know about the cgroups for now. You can exit the container we were running by pressing Ctrl-D. For more detailed information about cgroups you can always run `man cgroups`. Let's put our new knowledge to use and run some experiments!

5.3.9 Experiment 3: Using all the CPU I can find!

Docker offers two ways of controlling the amount of CPU a container gets to use, which are analogous to what we covered in the previous section. First, the `--cpus` flag, which controls the hard limit. Setting it to `--cpus=1.5` is equivalent to setting the period to 100,000 and

the quota to 150,000. Second, through the `--cpu-shares`, we can give our process a relative weight.

Let's test the first one with the following experiment:

1. Observability: observe the amount of CPU used by `stress` command, using top or mpstat
2. Steady state: CPU utilization close to 0
3. Hypothesis: if we run `stress` in CPU mode, in a container started with --cpus=0.5 it will use no more than 0.5 processor on average
4. Run the experiment!

To implement that, let's start by building a container with the stress command inside of it. I've prepared a simple Dockerfile for you, that you can see by running the following command in a terminal window:

```
cat ~/src/examples/poking-docker/experiment3/Dockerfile
```

You will see the following output, a very basic Dockerfile with a single command in it:

```
FROM ubuntu:focal-20200423

RUN apt-get update && apt-get install -y stress
```

Let's build a new image called stressful using that Dockerfile. You can do that by running the following command in a terminal window:

```
cd ~/src/examples/poking-docker/experiment3/
docker build -t stressful .
```

After a few seconds, you should be able to see the new image in the list of Docker images. You can see it by running the following command:

```
docker images
```

You will see the new image (in bold font) in the output, similar to the following one:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
stressful           latest              9853a9f38f1c        5 seconds ago       95.9MB
(...)
```

Now, let's set up our working space. To make things easy, let's try to have two terminal windows open side by side. In the first one, let's start the container in which we will use the stress command. You can do that by running the following command:

```
docker run \
--cpus=0.5 \                                    #A
-ti \                                           #B
--rm \                                          #C
--name experiment3 \                            #D
stressful                                       #E
```

#A limit the container to use half a CPU
#B keep STDIN open and allocate a pseudo-TTY to allow us to type commands
#C remove the container after we're done with it

#D name the container experiment3 so that it's easier to find later on
#E run the new image we just built, with the stress command in it

In the second terminal window, let's start monitoring the CPU usage of the system. Run the following command in the second window:

```
mpstat -u -P ALL 2
```

You should start seeing updates similar to the following, every 2 seconds. Note, that my VM is running with two CPUs, and so should yours if you're running the default values. Also, the %idle is around 99.75%.

```
Linux 4.15.0-99-generic (linux)    05/04/2020        _x86_64_ (2 CPU)

12:22:22 AM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice
       %idle
12:22:24 AM  all    0.25    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
       99.75
12:22:24 AM    0    0.50    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
       99.50
12:22:24 AM    1    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
       100.00
```

Showtime! In the first terminal, start the stress command:

```
stress --cpu 1 --timeout 30
```

In the second window running `mpstat`, you should start seeing one CPU at about 50% and the other one close to 0, resulting in total utilization of about 24.5%, similar to the following output:

```
12:27:21 AM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice
       %idle
12:27:23 AM  all   24.56    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
       75.44
12:27:23 AM    0    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
       100.00
12:27:23 AM    1   48.98    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
       51.02
```

To confirm it in a different way, we can inspect the contents of `cpu.stat` file in cgroupfs for that particular container.

```
CONTAINER_ID=$(docker inspect -f '{{ .Id }}' experiment3)
cat /sys/fs/cgroup/cpu/docker/$CONTAINER_ID/cpu.stat
```

You will see an output similar to the following one. Of particular interest, you will see an increasing `throttled_time`, which is a number of microseconds where processes in the cgroup were throttled and `nr_throttled` which is the number of periods in which throttling took place.

```
nr_periods 311                    #A
nr_throttled 304                  #B
throttled_time 15096182921        #C
```

#A number of elapsed CPU time periods

#B number of periods during which throttling took place (period size set with cpu.cfs_period_us)
#C total number of nanoseconds of CPU time throttled

That's another way of verifying that our setup worked. And work it did! Congratulations! The experiment worked, Docker did its job. If you used a higher value for the `--cpu` flag of the stress command, you would see the load spread across both CPUs, while still resulting in the same overall average. And if you check the cgroupfs metadata, you will see that Docker did indeed result in setting the `cpu.cfs_period_us` to 100000, `cpu.cfs_quota_us` to 50000 and `cpu.shares` to 1024. When you're done, you can exit the container, by pressing Ctrl-D.

I wonder if it'll go as smoothly with limiting the RAM. Shall we find out?

5.3.10 Experiment 4: Using too much RAM

To limit the amount of RAM a container is allowed to use, you can use Docker's `--memory` flag. It accepts b (bytes), k (kilobytes), m (megabytes), and g (gigabytes) as suffixes. As an effective chaos engineering practitioner, we want to know what happens when a process reaches that limit.

Let's test it with the following experiment:

1. Observability: observe the amount of RAM used by `stress` command, using `top`; monitor for OOM in `dmesg`
2. Steady state: no logs of killing in `dmesg`
3. Hypothesis: if we run `stress` in RAM mode, trying to consume 512MB, in a container started with `--memory=128m`, it will use no more than 128MB of RAM
4. Run the experiment!

Let's set up our working space again with two terminal windows open side by side. In the first one, let's start a container with the same image we did for the previous experiment, but this time-limiting the memory, not the CPU. You can do that by running the following command:

```
docker run \
--memory=128m \                                    #A
-ti \                                              #B
--name experiment4 \                               #C
--rm \                                             #D
stressful                                          #E
```

#A limit the container to a max of 128MB of RAM
#B keep STDIN open and allocate a pseudo-TTY to allow us to type commands
#C name our container experiment 4
#C remove the container after we're done with it
#D run the same stress image we built for experiment 3

In the second terminal window, let's first check the dmesg logs to see that there is nothing about OOM killing (if you've forgotten all about the OOM, it's the Linux kernel feature which kills processes to recover RAM - we covered it in chapter 2). Run the following command in the second terminal window:

```
dmesg | egrep "Kill|oom"
```

Depending on the state of your VM machine, you might not get any results, or if you do, mark the timestamp, so that you can differentiate them from fresher logs. Now, let's start monitoring the RAM usage of the system. Run the following command in the second window:

```
top
```

You will start seeing updates of the top command. Observe and note the steady state levels of RAM utilization.

With that, the scene is set! Let's start the experiment, by running the following command in the first terminal window, from within the container. It will run RAM workers, each allocating 512MB of memory (bold font):

```
stress \
--vm 1 \                                                    #A
--vm-bytes 512M \                                           #B
--timeout 30                                                #C
```

#A run 1 worker allocating memory
#C allocate 512 MB
#C run for 30 seconds

While that's running, you will see something interesting in the top command, similar to the following output. Notice that the container is using 528152KiB of virtual memory, and 127400KB of reserved memory, just under the 128MB limit we gave to the container.

```
Tasks: 211 total,   1 running, 173 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.1 sy,  0.0 ni, 99.6 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4039228 total,  1235760 free,  1216416 used,  1587052 buff/cache
KiB Swap:  1539924 total,  1014380 free,   525544 used.  2526044 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
32012 root      20   0  528152 127400    336 D  25.0  3.2   0:05.28 stress
(...)
```

After 30 seconds, the stress command will finish and print the following output. It happily concluded its run.

```
stress: info: [537] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: info: [537] successful run completed in 30s
```

Well, that's a fail for our experiment--and a learning opportunity! Things get even weirder if you rerun the stress command, but this time with `--vm 3`, to run three workers, each trying to allocate 512MB. In the output of top (the second window), you will notice that all three of them have 512MB of virtual memory allocated to them, but their total reserved memory adds up to about 115MB, below our limit.

```
Tasks: 211 total,   1 running, 175 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.2 us,  0.1 sy,  0.0 ni, 99.6 id,  0.1 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4039228 total,  1224208 free,  1227832 used,  1587188 buff/cache
KiB Swap:  1539924 total,    80468 free,  1459456 used.  2514632 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
32040 root      20   0  528152  32432    336 D   6.2  0.8   0:02.22 stress
32041 root      20   0  528152  23556    336 D   6.2  0.6   0:02.40 stress
32042 root      20   0  528152  59480    336 D   6.2  1.5   0:02.25 stress
```

It looks like the kernel is doing something smart, because stress doesn't actually do anything with the allocated memory, so our initial idea of the experiment won't work. What can we do instead to see the kernel limit the amount of memory our container can use? Well, we could always use a good old fork bomb. It's for science!

Let's monitor the memory usage of the container. To do this, let's leverage the cgroupfs once again, this time to read the number of bytes of used memory, by running in a third terminal window the following command:

```
export CONTAINER_ID=$(docker inspect -f '{{ .Id }}' experiment4)
watch -n 1 sudo cat /sys/fs/cgroup/memory/docker/$CONTAINER_ID/memory.usage_in_bytes
```

And in the first terminal (inside of our container) let's drop the fork bomb, by running the following command. All it's doing is calling itself recursively to exhaust the available resources:

```
boom () {
  boom | boom &
}; boom
```

Now, in the third terminal, you will see that the number of bytes used is oscillating somewhere just above 128MB, slightly more than the limit that we gave to the container. In the second window, running top, you're likely to see something similar to the following output. Note the very high CPU system time percentage (in bold font).

```
Tasks: 1173 total, 131 running, 746 sleeping,   0 stopped, 260 zombie
%Cpu(s):  6.3 us, 89.3 sy,  0.0 ni,  0.0 id,  0.8 wa,  0.0 hi,  3.6 si,  0.0 st
```

In the first window, inside of the container, you will see bash failing to allocate memory:

```
bash: fork: Cannot allocate memory
```

If the container hasn't been killed by the OOM, you can stop it by running the following command in a terminal window:

```
docker stop experiment4
```

Finally, let's check the dmesg for OOM logs by running the following command:

```
dmesg | grep Kill
```

You will see an output similar to the following one. The kernel notices the cgroup is out of memory, and kicks in to kill some of the processes within it. But because our fork bomb managed to start a few thousand processes, it actually takes a non-negligible CPU power for the OOM to do its thing.

```
[133039.835606] Memory cgroup out of memory: Kill process 1929 (bash) score 2 or sacrifice
       child
[133039.835700] Killed process 10298 (bash) total-vm:4244kB, anon-rss:0kB, file-rss:1596kB,
       shmem-rss:0kB
```

Once again a failed experiment teaches us more than a successful one. What did we learn? A few interesting bits of information:

- Just allocating the memory doesn't trigger OOM, and we can successfully allocate

much more memory than the cgroup allows for

- When using a fork bomb, the total of the memory used by our forks was slightly higher than the limit we allocated to the container, which is useful when doing capacity planning
- The cost of running OOM dealing with a fork bomb is non-negligeable and can actually be pretty high. If you've done your math when allocating resources, it might be worth considering disabling OOM for the container through the `--oom-kill-disable` flag.

Now, armed with that new knowledge, let's revisit for the third - and final - time our bare-bones container(-ish) implementation.

5.3.11 Implementing a simple container(-ish) part 3 - cgroups

In part 2 of the mini-series on a DIY container, we reused the script that prepared a filesystem, and we started chroot from within a new namespace. Now, in order to limit the amount of resources our container-ish can use, we can leverage the cgroups we just learned about. To keep things simple, let's focus on just two cgroup types: memory and CPU. To refresh your memory on how this fits in the big picture, take a look at figure 5.10. It shows where cgroups fit with the other underlying technology in Linux kernel that Docker leverages.

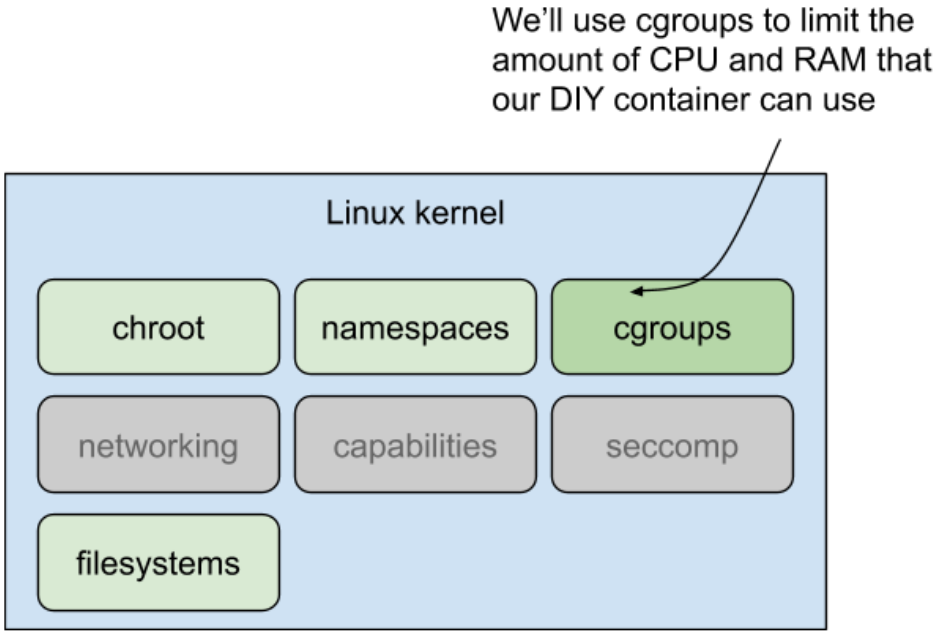


Figure 5.10 DIY container part 3 - cgroups

Now, let's put to use everything we've learned in the previous section. To create a new cgroup, all we need to do is create a new folder in the corresponding `cgroupfs` filesystem. To

configure the cgroup, we'll put the values we want in the files we've looked at in the previous section. And to add a new process to that filesystem, we'll add our bash process to it by writing to the `tasks` file. All the children of that process will then automatically be included in there. And voila!

I've prepared a script that does that. You can see it by running the following command in a terminal window inside of your VM:

```
cat ~/src/examples/poking-docker/container-ish-2.sh
```

You will see the following output. We reuse, once again, the filesystem prep script from part 1 of this series, and create and configure two new cgroups of type cpu and memory. Finally, we start the new process using unshare and chroot, exactly the same way we did in part 2.

```
#! /bin/bash
set +x

CURRENT_DIRECTORY="$(dirname "${0}")"
CPU_LIMIT=${1:-50000}
RAM_LIMIT=${2:-5242880}

echo "Step A: generate a unique ID (uuid)"
UUID=$(date | sha256sum | cut -f1 -d" ")                                    #A

echo "Step B: create cpu and memory cgroups"
sudo mkdir /sys/fs/cgroup/{cpu,memory}/$UUID                                #B
echo $RAM_LIMIT | sudo tee /sys/fs/cgroup/memory/$UUID/memory.limit_in_bytes  #C
echo 100000 | sudo tee /sys/fs/cgroup/cpu/$UUID/cpu.cfs_period_us
echo $CPU_LIMIT | sudo tee /sys/fs/cgroup/cpu/$UUID/cpu.cfs_quota_us

echo "Step C: prepare the folder structure to be our chroot"
bash $CURRENT_DIRECTORY/new-filesystem.sh $UUID > /dev/null && cd $UUID     #D

echo "Step D: put the current process (PID $$) into the cgroups"
echo $$ | sudo tee /sys/fs/cgroup/{cpu,memory}/$UUID/tasks                  #E

echo "Step E: start our namespaced chroot container-ish: $UUID"
sudo unshare \                                                              #F
    --fork \
    --pid \
    chroot . \
    /bin/bash -c "mkdir -p /proc && /bin/mount -t proc proc /proc && exec /bin/bash"
```

#A generate a nice looking UUID
#B create cpu and memory cgroups using the UUID as the name
#C write the values we want to limit RAM and CPU usage
#D prepare a filesystem to chroot into
#E add the current process to the cgroup
#F start a bash session using a new pid namespace and chroot

You can now start our container-ish, by running the following command in a terminal window:

```
~/src/examples/poking-docker/container-ish-2.sh
```

You will see the following output, and will be presented with an interactive bash session. Note the container UUID (in bold font):

```
Step A: generate a unique ID (uuid)
Step B: create cpu and memory cgroups
5242880
100000
50000
Step C: prepare the folder structure to be our chroot
Step D: put the current process (PID 10568) into the cgroups
10568
Step E: start our namespaced chroot container-ish:
       169f4eb0dbd1c45fb2d353122431823f5b7b82795d06db0acf51ec476ff8b52d
Welcome to the kind-of-container!
bash-4.4#
```

Leave this session running, and open another terminal window. In that window, let's investigate the cgroups our processes are running in. Run the following command in that second terminal window:

```
ps -ao pid,command -f
```

You will see an output similar to the following one (I abbreviated it to only show the part we're interested in). Note the PID of the bash session "inside" our container(-ish):

```
  PID COMMAND
(...)
 4628 bash
10568  \_ /bin/bash /home/chaos/src/examples/poking-docker/container-ish-2.sh
10709      \_ sudo unshare --fork --pid chroot . /bin/bash -c mkdir -p /proc && /bin/mount
       -t
10717          \_ unshare --fork --pid chroot . /bin/bash -c mkdir -p /proc && /bin/mount -
       t
10718              \_ /bin/bash
```

With that PID, we can finally confirm the cgroups that processes ended up in. To do that, we can use the good old ps command. Run the following command in the second terminal window:

```
ps \
-p 10718 \                #A
-o pid,cgroup \           #B
-ww                       #C
```

#A show the process with the requested PID
#B print pid and cgroups
#C don't shorten the output to fit the width of the terminal, print all

You will see an output just like the following. Note the `cpu,cpuacct,` and `memory` cgroups (in bold font), which should match the UUID you saw in the output when your container(-ish) started. In other aspects, it's using the default cgroups.

```
  PID CGROUP
10718 12:pids:/user.slice/user-
       1000.slice/user@1000.service,10:blkio:/user.slice,9:memory:/169f4eb0dbd1c45fb2d35312
       2431823f5b7b82795d06db0acf51ec476ff8b52d,6:devices:/user.slice,4:cpu,cpuacct:/169f4e
       b0dbd1c45fb2d353122431823f5b7b82795d06db0acf51ec476ff8b52d,1:name=systemd:/user.slic
       e/user-1000.slice/user@1000.service/gnome-terminal-
       server.service,0::/user.slice/user-1000.slice/user@1000.service/gnome-terminal-
       server.service
```

I invite you to play around with the container and see for yourself how well the process is contained. With this short script slowly built over three parts of the series, we've contained the process in a few important aspects:

1. The filesystem access
2. PID namespace separation
3. CPU and RAM limits

To aid the visual memory, take a look at figure 5.11. It shows the elements we have covered (chroot, filesystems, namespaces, cgroups) and underlines the ones which remain to be covered (networking, capabilities and seccomp).

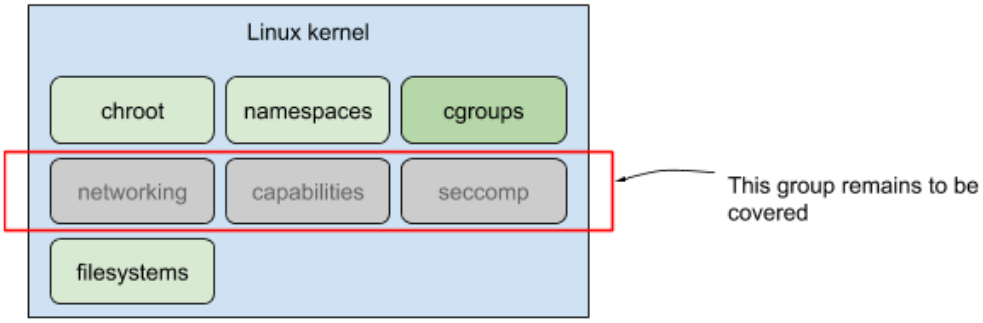


Figure 5.11 Coverage status after the DIY container part 3

It's beginning to look more like a real container, but with one large caveat: its networking access is still exactly the same as for any other process running on the host and we haven't covered any security features at all. Let's look into how Docker does networking next.

5.3.12 Docker and networking

Docker allows you to explicitly manage networking, through the use of the `docker network` subcommand. By default, Docker comes with three networking options for you to choose from when you're starting a container. Let's list the existing networks by running the following command in a terminal window:

```
docker network ls
```

As you can see, the output lists three options: bridge, host, and none (in bold font). For now, you can safely ignore the scope column:

```
NETWORK ID          NAME                DRIVER              SCOPE
130e904f5364        bridge              bridge              local
2ac4140a7b9d        host                host                local
278d7624eb4b        none                null                local
```

Let's start with the easy one: `none`. If you start a container with `--network none`, no networking will be set up. This is useful if you want to isolate your container from the

network and make sure it can't be contacted. Note, that this is a runtime option: it doesn't affect how an image is built. You can build an image by downloading packages from the internet, but then run the finished product without access to any network. It uses a `null` driver.

The second is also straightforward: `host`. If you start a container with `--network host`, the container will use the host's networking setup directly, without any special treatment or isolation. The ports you try to use from inside the container will be the same as if you did it from the outside. The driver for this mode is also called `host`.

Finally, the `bridge` mode is where it gets interesting. In networking, a *bridge* is an interface, which connects multiple networks and forwards traffic between the interfaces it's connected to. You can think of it as a network switch. Docker leverages a bridge interface to provide network connectivity to containers through use of virtual interfaces. It works like this:

- Docker creates a *bridge* interface called `docker0` and connects it to the host's logical interface
- For each container, Docker creates a `net` namespace, which allows it to create network interfaces only accessible to processes in that namespace
- Inside of that namespace, Docker creates the following:
    - a virtual interface connected to the `docker0` bridge
    - a local loopback device

When a process from within a container tries to connect to the outside world, the packets go through its virtual network interface and then the bridge, which routes it where it should go. This architecture is summarized in figure 5.12.

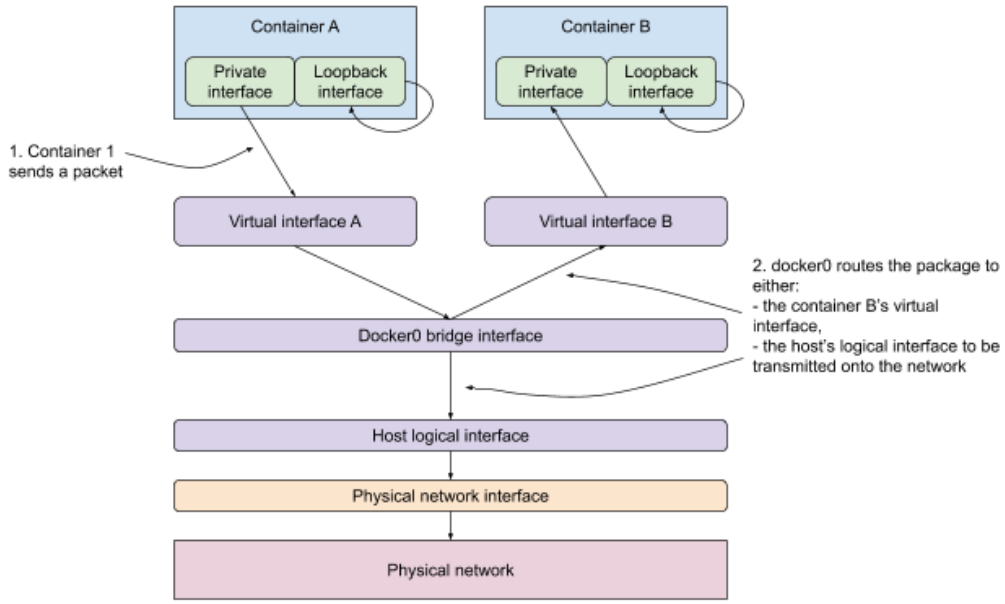


Figure 5.12 Docker networking running two containers in bridge mode

You can see the default docker bridge device in your VM by running the following command in a terminal window:

```
ip addr
```

You will see an output similar to the following one (abbreviated for clarity). Note the local loopback device (`lo`), the ethernet device (`eth0`) and the docker bridge (`docker0`):

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
(...)
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default
       qlen 1000
    link/ether 08:00:27:bd:ac:bf brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute eth0
       valid_lft 84320sec preferred_lft 84320sec
(...)
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group
       default
    link/ether 02:42:cd:4c:98:33 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
```

So far, all of the containers we have started were running on the default network settings. Let's now go ahead and create a new network and inspect what happens. Creating a new Docker network is simple. To create a funky new network, run the following command in a terminal window:

```
docker network create \
  --driver bridge \                        #A
  --attachable \                           #B
  --subnet 10.123.123.0/24 \               #C
  --ip-range 10.123.123.0/25 \             #D
  chaos                                    #E
```

#A use the bridge driver to allow connectivity to the host's network
#B allow for containers to manually attach to this network
#C pick a funky subnet
#D only give containers IP from this sub-range of that funky subnet
#E give it a name

Once that's done, you can confirm the new network is there by running the following command again:

```
docker network ls
```

You will see an output just like the following, including our new network called "chaos" (bold font):

```
NETWORK ID          NAME                DRIVER              SCOPE
130e904f5364        bridge              bridge              local
b1ac9b3f5294        chaos               bridge              local
2ac4140a7b9d        host                host                local
278d7624eb4b        none                null                local
```

Let's now rerun the ip command to list all available network interfaces:

```
ip addr
```

In the following abbreviated output, you'll notice the new interface `br-b1ac9b3f5294` (bold font), which has our funky IP range configured:

```
(...)
4: br-b1ac9b3f5294: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
       group default
    link/ether 02:42:d8:f2:62:fb brd ff:ff:ff:ff:ff:ff
    inet 10.123.123.1/24 brd 10.123.123.255 scope global br-b1ac9b3f5294
       valid_lft forever preferred_lft forever
```

Let's now start a container using that new network. You can do that by running the following command in a terminal window:

```
docker run \
    --name explorer \
    -ti \
    --rm \
    --network chaos \                      #A
    ubuntu:focal-20200423
```

#A note that we're using our brand new network

The image we're running is pretty slim, so in order to look inside, we'll need to install the ip command. You can do that by running the following command from inside of that container:

```
apt-get update
```

```
apt install -y iproute2
```

Now, let's investigate! From inside the container, run the following ip command to see what interfaces are available:

```
ip addr
```

You will see an output just like the following one. Note the interface with our funky range (bold font).

```
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
5: eth0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
       default
    link/ether 02:42:0a:7b:7b:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.123.123.2/24 brd 10.123.123.255 scope global eth0
       valid_lft forever preferred_lft forever
```

You can confirm you've gotten an IP address from within that funky range by running the following command inside of the container:

```
hostname -I
```

Sure enough, it's what you'd expect it to be, just like the following:

```
10.123.123.2
```

Now, let's see how that plays with the net namespaces. You will remember from the previous sections that we can list namespaces using `lsns`. Let's list the net namespaces by running the following command in a second terminal window on the host (not in the container we're running):

```
sudo lsns -t net
```

You will see the following output. I happen to have three net namespaces running.

```
        NS TYPE NPROCS   PID USER  COMMAND
4026531993 net     208     1 root  /sbin/init
4026532172 net       1 12543 rtkit /usr/lib/rtkit/rtkit-daemon
4026532245 net       1 20829 root  /bin/bash
```

But which one is our container's? Let's leverage what we learned about the namespaces to track our container's net namespace by its PID. Run the following command in the second terminal window (not inside the container):

```
CONTAINER_PID=$(docker inspect -f '{{ .State.Pid }}' explorer)
sudo readlink /proc/$CONTAINER_PID/ns/net
```

You will see an output similar to the following one. In my example, the namespace is `4026532245`.

```
net:[4026532245]
```

Now, for the grand finale, let's enter that namespace. In the section on namespaces we used nsenter with the `--target` flag using a process' PID. We could do it here, but I'd also like to show you another way of targeting a namespace. To directly use the namespace file, run the following command in the second terminal window (outside of the container):

```
CONTAINER_PID=$(docker inspect -f '{{ .State.Pid }}' explorer)
sudo nsenter --net=/proc/$CONTAINER_PID/ns/net
```

You will notice that your prompt has changed: you are now root inside of the net namespace `4026532245`. Let's confirm that we are seeing the same set of network devices we saw from inside of the container. Run the following command in this new prompt

```
ip addr
```

You will see the same output you saw from inside of the container, just like in the following output:

```
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
5: eth0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
       default
    link/ether 02:42:0a:7b:7b:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.123.123.2/24 brd 10.123.123.255 scope global eth0
       valid_lft forever preferred_lft forever
```

When you're done playing, you can type exit or press Ctrl-D to exit the shell session and therefore the namespace. Well done, we've just covered the basics you need to know about networking - the fourth pillar of how Docker implements the containers. Last stop on this journey - capabilities and other security mechanisms.

5.3.13 Capabilities and seccomp

The final pillar of what makes Docker is the use of *capabilities* and *seccomp*. For the final time, let me refresh your memory of where they fit on figure 5.13.

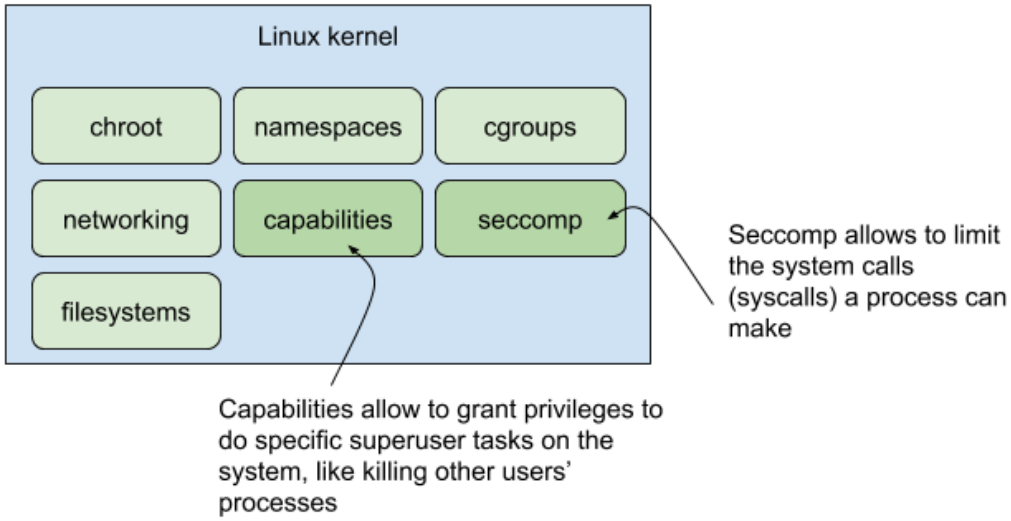


Figure 5.13 Capabilities and seccomp

We'll cover them briefly, because they're necessary for the complete image of how Linux containers are implemented with Docker, but we wouldn't do it justice to try to get into how they work under the hood in a single section. I'll leave that part as an exercise to the reader.

5.3.13.1 CAPABILITIES

Let's start with *capabilities*. It's a Linux kernel feature that splits the superuser privileges (which skip all checks) into smaller, more granular units of permissions, each unit called - you guessed it - a capability. So instead of binary "all" or "nothing," you can grant users permissions to do specific tasks. For example, any user with capability CAP_KILL bypasses permission checks for sending signals to processes. The same way, any user with CAP_SYS_TIME can change the system clock.

By default, Docker grants every container a default set of capabilities. To find out what they are, let's start a container, and use the `getpcaps` command to list the capabilities we have. Run the following command in a terminal window to start a fresh container with all the default settings:

```
docker run \
    --name cap_explorer \
    -ti --rm \
    ubuntu:focal-20200423
```

While that container is running, we can check its capabilities in another window by finding out its PID, and using the getpcaps command. You can do that by running the following command:

```
CONTAINER_PID=$(docker inspect -f '{{ .State.Pid }}' cap_explorer)
getpcaps $CONTAINER_PID
```

You will see an output similar to the following one, listing all the capabilities a Docker container gets by default. Notice the `cap_sys_chroot` capability (bold font):

```
Capabilities for `4380': =
       cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_
       setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,ca
       p_setfcap+eip
```

To verify it works, let's have some Inception-style fun by chrooting inside the container's chroot! You can do that by running the following commands inside of your container:

```
NEW_FS_FOLDER=new_fs
mkdir $NEW_FS_FOLDER
cp -v --parents `which bash` $NEW_FS_FOLDER                                    #A
ldd `which bash` | egrep -o '(/usr)?/lib.*\.[0-9][0-9]?' \                     #B
| xargs -I {} cp -v --parents {} $NEW_FS_FOLDER                                #C
chroot $NEW_FS_FOLDER `which bash`                                             #D
```

#A copy bash binary to the subfolder
#B find out all the libraries bash needs
#C copy the libraries over into their respective locations
#D run the actual chroot from the new subfolder and start bash

You will land in a new bash session (with not much to do, because we've copied only the bash binary itself). Now, to the twist: when starting a new container with docker run, you can use `--cap-add` and `--cap-drop` flags to add or remove any particular capability, respectively. A special keyword `ALL` allows for adding or dropping all available privileges.

Let's now kill the container (press Ctrl-D) and restart it with `--cap-drop ALL` flag, using the following command:

```
docker run \
    --name cap_explorer \
    -ti --rm \
    --cap-drop ALL \
    ubuntu:focal-20200423
```

While that container is running, we can check its capabilities in another window, by finding out its PID, and using the getpcaps command. You can do that by running the following command:

```
CONTAINER_PID=$(docker inspect -f '{{ .State.Pid }}' cap_explorer)
getpcaps $CONTAINER_PID
```

You will see an output similar to the following one, this time listing no capabilities at all.

```
Capabilities for `4813': =
```

From inside the new container, retry the chroot snippet, by running the following commands again:

```
NEW_FS_FOLDER=new_fs
mkdir $NEW_FS_FOLDER
cp -v --parents `which bash` $NEW_FS_FOLDER
ldd `which bash` | egrep -o '(/usr)?/lib.*\.[0-9][0-9]?' | xargs -I {} cp -v --parents {}
       $NEW_FS_FOLDER
```

```
chroot $NEW_FS_FOLDER `which bash`
```

This time you will see the following error:

```
chroot: cannot change root directory to 'new_fs': Operation not permitted
```

Docker leverages that (and so should you) to limit the actions the container can perform. It's always a good idea to only give the container what it really needs in terms of capabilities. And you have to admit - Docker makes it pretty easy. Now, let's take a look at seccomp.

5.3.13.2 SECCOMP

Seccomp is a Linux kernel feature that allows you to filter which syscalls a process can make. Interestingly, under the hood, it uses Berkeley Packet Filter (BPF; for more information, see chapter 3) to implement the filtering. Docker leverages seccomp to limit the default set of syscalls that are allowed for containers (see more details about that set at https://docs.docker.com/engine/security/seccomp/).

Docker's seccomp profiles are stored in JSON files, which describe a series of rules to evaluate which syscalls to allow. You can see the Docker's default profile at https://github.com/moby/moby/blob/master/profiles/seccomp/default.json. To give you a preview of what a profile looks like, here's an extract from the Docker's default:

```
{
       "defaultAction": "SCMP_ACT_ERRNO",                              #A
...
       "syscalls": [
         {
                  "names": [                                           #B
                           "accept",
                           "accept4",
...
                           "write",
                           "writev"
                  ],
                  "action": "SCMP_ACT_ALLOW",                          #C
...
         },
...
       ]
}
```

#A by default, block all calls
#B for the syscalls with the following list of names
#C allow them to proceed

To use a different profile than the default, use the `--security-opt seccomp=/my/profile.json` flag when starting a new container. That's all we're going to cover about seccomp in the context of Docker. Right now I just need you to know that it exists, that it limits the syscalls which are allowed, and that you can leverage that without using Docker, because it's a Linux kernel feature. Let's go ahead and review what we've seen under Docker's hood.

5.3.14 Docker demystified

By now, you understand that containers are implemented with a collection of loosely connected technologies and that to know what to expect from the dish you need to know the ingredients. We've covered chroot, namespaces, cgroups, networking, and briefly, capabilities, seccomp, and filesystems. Figure 5.14 shows once again what each of these technologies are for, to drive the point home. This section showed you that Docker, as well as the Linux features that do the heavy lifting, are not that scary once you've checked what's under the hood. They are very useful technologies and they are fun to use! Understanding them is crucial to designing chaos engineering experiments in any system involving Linux containers.

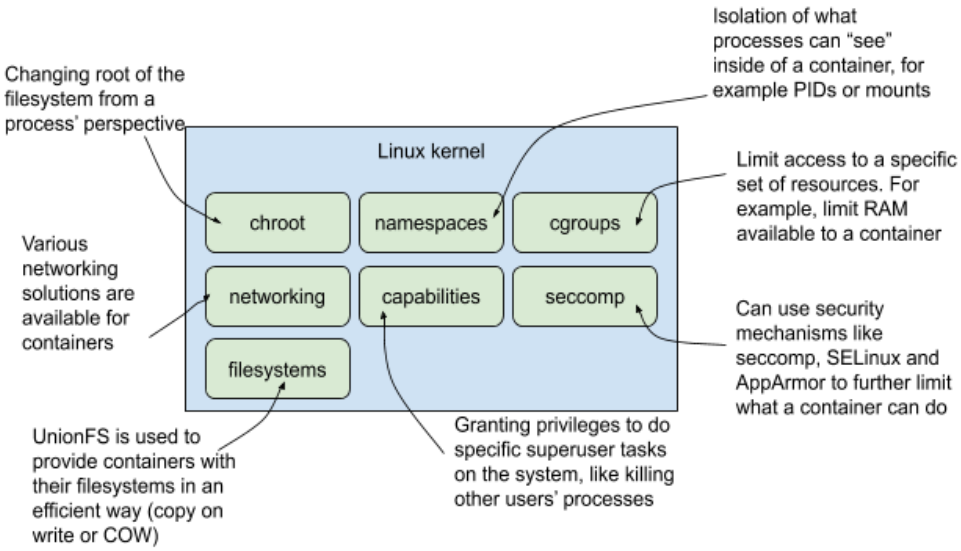


Figure 5.14 High-level overview of Docker interacting with the kernel

Given the current state of the ecosystem, the containers seem to be here to stay. To learn more about these technologies, I suggest starting with the man pages. Both `man namespaces` and `man cgroups` are pretty well written and accessible. Online documentation of Docker (https://docs.docker.com/) also provides a lot of useful information on Docker and also the underlying kernel features.

I'm confident that you will be able to face whatever containerized challenges life throws at you when practicing chaos engineering. Now we're ready to go and fix our dockerized Meower USA app being slow.

5.4 Fixing my (dockerized) app being slow

Let's refresh our memory on how the app is deployed. Figure 5.15 shows a simplified overview of the app's architecture, from which I've removed the third-party load balancer; I'm only showing a single instance of Ghost, connecting to the MySQL database.

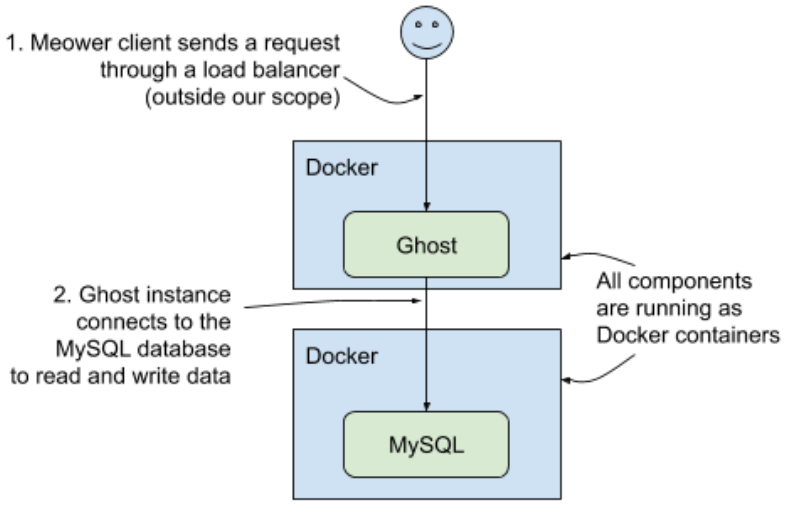


Figure 5.15 Simplified overview of Meower USA technical architecture

It's a very simple setup--purposefully so, so that we can focus on the new element in the equation: Docker. Let's bring this up in our VM.

5.4.1 Booting up Meower

Now that we're comfortable running Docker commands, let's start up the Meower stack in the VM. We are going to use the functionality of Docker which allows you to describe a set of containers that need to be deployed together - `docker stack deploy` (see https://docs.docker.com/engine/reference/commandline/stack_deploy/ for more information.) This command uses simple-to-understand YAML files to describe sets of containers. This allows for a portable description of an application. You can see the description for the Meower stack by running the following command in a terminal in your VM:

```
cat ~/src/examples/poking-docker/meower-stack.yml
```

You will see the following output. It describes two containers, one for MySQL, and another one for Ghost. It also configures Ghost to use the MySQL database and takes care of things such as (very insecure) passwords.

```
version: '3.1'
services:
  ghost:
```

```
    image: ghost:3.14.0-alpine                                    #A
    ports:
      - 8368:2368                                                 #B
    environment:
      database__client: mysql
      database__connection__host: db
      database__connection__user: root
      database__connection__password: notverysafe                 #C
      database__connection__database: ghost
      server__host: "0.0.0.0"
      server__port: "2368"
  db:
    image: mysql:5.7                                              #D
    environment:
      MYSQL_ROOT_PASSWORD: notverysafe                            #E
```

#A run the ghost container in a specific version
#B expose port 8368 on the host to route to port 2368 in the ghost container
#C specify the database password for ghost to use
#D run the mysql container
#E specify the same password for the mysql container to use

Let's start it! Run the following commands in a terminal window:

```
docker swarm init                                                 #A
docker stack deploy \
-c ~/src/examples/poking-docker/meower-stack.yml \                #B
meower                                                            #C
```

#A we need to initialize our host to be able to run docker stack commands
#B use the stack file we saw earlier
#C give the stack a name

When that's done, we can confirm the stack was created by running the following command in a terminal window:

```
docker stack ls
```

You will see the following output, showing a single stack, meower, with two services in it.

```
NAME                SERVICES            ORCHESTRATOR
meower              2                   Swarm
```

To confirm what Docker containers it started, run the following command in a terminal window:

```
docker ps
```

You will see an output similar to the following one. As expected, we can see two containers, one for MySQL and one for the Ghost application.

```
CONTAINER ID        IMAGE                 COMMAND                  CREATED
       STATUS              PORTS               NAMES
72535692a9d7        ghost:3.14.0-alpine   "docker-entrypoint.s…"   39 seconds ago      Up
       32 seconds          2368/tcp            meower_ghost.1.4me3qjpcks6o8hvc19yp26svi
7d32d97aad37        mysql:5.7             "docker-entrypoint.s…"   51 seconds ago      Up
       48 seconds          3306/tcp, 33060/tcp   meower_db.1.ol7vjhnnwhdx34ihpx54sfuia
```

To confirm that it worked, browse to http://127.0.0.1:8080/. If you feel like configuring the Ghost instance, feel free to go to http://127.0.0.1:8080/ghost/ but for our purposes it's fine to leave it unconfigured.

With the setup out of the way, we can now focus on the question that brought us here in the first place: why is the app slow?

5.4.2 Why is the app slow?

So why might the app be slow? Given what we've learned so far in this chapter, there are at least two plausible explanations for the slowness of the Meower application.

One of the reasons might be that the process is starved for CPU time. It sounds obvious, but I've seen it happen a lot, when someone... else... typed one zero too few or too many. Fortunately, now you know that it's easy to check the `cpu.stat` of the underlying cgroup to see if any throttling took place at all, and take it from there.

Another reason, which we explored in chapter 4 with Wordpress, is that the application is more fragile to the networking slowness of its database than we expected. It's a common gotcha to make assumptions based on the information from test environments and local databases, and then be surprised when networking slows down in the real world.

I'm confident that you can handle the first possibility with ease. I suggest then that we explore the second one now in the context of Docker, and using a positively more modern stack than that of chapter 4. Hakuna Matata!

5.4.3 Experiment 5: Network slowness for containers with Pumba

Let's conduct an experiment in which we add a certain amount of latency to the communications between Ghost and MySQL, and see how that affects the response time of the website. To do that, we can once again rely on ab to generate traffic and produce metrics about the website response time and error rate. Here's the four steps to one such experiment:

1. Observability: use ab to generate a certain amount of load, monitor for average response time and error rate
2. Steady state: there are no errors and we average X ms per request
3. Hypothesis: if we introduce 100ms latency to network connectivity between Ghost and MySQL, we should see the average website latency go up by 100ms
4. Run the experiment!

So the only question remaining now is: what's the easiest way to inject latency into Docker containers?

5.4.2.1 Pumba - Docker chaos engineering tool

Pumba (https://github.com/alexei-led/pumba) is a really neat tool that helps conducting chaos experiments on Docker containers. It can kill containers, emulate network failures (using tc under the hood), and run stress tests (using stress-ng - https://kernel.ubuntu.com/~cking/stress-ng/) from inside a particular container's cgroup.

NOTE Pumba is preinstalled in the VM; for installation on your host, see appendix A.

It's really convenient to use, because it operates on container names and saves a lot of typing. The syntax is straightforward. Take a look at this excerpt from running `pumba help` in a terminal window:

```
USAGE:
   pumba [global options] command [command options] containers (name, list of names, RE2
      regex)

COMMANDS:
     kill     kill specified containers
     netem    emulate the properties of wide area networks
     pause    pause all processes
     stop     stop containers
     rm       remove containers
     help, h  Shows a list of commands or help for one command
```

To introduce latency to a container's egress, we're interested in the netem subcommand. Under the hood, it uses the same tc command that we leveraged in chapter 4, section 4.2.2.2, but it's much easier to use. There is one gotcha though: the way it works by default is through executing a tc command from inside a container. That means that tc needs to be available, which is unlikely for anything other than a testing container.

Fortunately, there is a convenient workaround. Docker allows you to start a container in a way that the networking configuration is shared with another, pre-existing container. By doing that, it is possible to start a container that has the tc command available, run it from there, and affect both containers' networking. Pumba conveniently allows for that through `--tc-image` flag, which allows us to specify the image to use to create a new container (you can use `gaiadocker/iproute2` as an example container that has tc installed). Putting it all together, we can add latency to a specific container called "my_container" by running the following command in the terminal:

```
pumba netem \
--duration 60s \                    #A
--tc-image gaiadocker/iproute2 \    #B
delay \                                     #C
--time 100 \                                #D
"my_container"                              #E
```

#A duration of the experiment - how long the delay should be in there
#B specify the image to run that has the tc command available
#C use the delay subcommand
#D specify the delay (ms)
#E specify the name of the container to affect

Armed with that, we are ready to run the experiment!

5.4.2.1 CHAOS EXPERIMENT IMPLEMENTATION

First things first, let's establish the steady state. To do that, let's run `ab`. We will need to be careful to run with the same settings later on to compare apples to apples. Let's run for 30 seconds to give it long enough to produce a meaningful number of responses, but not long

enough for us to waste time. And let's start with concurrency of 1, because in our setting, we're using the same CPUs to produce and serve the traffic, so it's a good idea to keep the number of variables to a minimum. You can do that by running the following command in your terminal:

```
ab -t 30 -c 1 -l http://127.0.0.1:8080/
```

You will see an output similar to the following one. I abbreviated it for clarity. Note the time per request at around 26ms (in bold font) and failed requests at 0 (also bold font):

```
(...)
Complete requests:      1140
Failed requests:        0
(...)
Time per request:       26.328 [ms] (mean)
(...)
```

Now, let's run the actual experiment. Open another terminal window. Let's find the name of the Docker container running MySQL by running the following command in this second terminal window:

```
docker ps
```

You will see an output similar to the following. Note the name of the MySQL container (bold font):

```
docker ps
CONTAINER ID        IMAGE                 COMMAND                  CREATED
       STATUS              PORTS                 NAMES
394666793a39        ghost:3.14.0-alpine   "docker-entrypoint.s…"   2 hours ago         Up 2
       hours               2368/tcp              meower_ghost.1.svumole20gz4bkt7iccnbj8hn
a0b83af5b4f5        mysql:5.7             "docker-entrypoint.s…"   2 hours ago         Up 2
       hours               3306/tcp, 33060/tcp   meower_db.1.v3jamilxm6wmptphbgqb8bung
```

Conveniently, Pumba allows us to use regular expressions, by prepending the expression with **re2:**. So, to add a 100 ms of latency to our MySQL container for 60 seconds, let's run the following command, still in the second terminal window (bold font for the regular expression prefix). Note, that to simplify the analysis, we're disabling both random jitter and correlation between the events, to just simply add the same delay to each call:

```
pumba netem \
--duration 60s \                    #A
--tc-image gaiadocker/iproute2 \    #B
delay \                             #C
--time 100 \                        #D
--jitter 0 \                        #E
--correlation 0 \                   #F
"re2:meower_db"                     #G
```

#A duration of the experiment - how long the delay should be in there
#B specify the image to run that has the tc command available
#C use the delay subcommand
#D specify the delay (ms)
#E disable random jitter
#F disable correlation between the events

#G specify the name of the container to affect using regular expressions

Now, while the delay is in place (we have 60 seconds!) switch back to the first terminal window, and rerun the same ab command as before:

```
ab -t 30 -c 1 -l http://127.0.0.1:8080/
```

The output you'll see will be rather different from the previous one and similar to the following (abbreviated for brevity, failed requests and time per request in bold font):

```
(...)
Complete requests:      62
Failed requests:        0
(...)
Time per request:       490.128 [ms] (mean)
(...)
```

Ouch. A "mere" 100ms added latency to the MySQL database changes the average response time of Meower USA from 26ms to 490ms, or a factor of more than 18. If this sounds suspicious to you, that's the reaction I'm hoping for!. To confirm our findings, let's rerun the same experiment, but this time let's use 1ms as the delay, the lowest that the tool will allow us. To add the delay, run the following command in the second terminal window:

```
pumba netem \
--duration 60s \
--tc-image gaiadocker/iproute2 \
delay \
--time 1 \                                  #A
--jitter 0 \
--correlation 0 \
"re2:meower_db"
```

#A this time use a delay of just 1ms

In the first terminal, while that's running, rerun the ab command once again with the following command:

```
ab -t 30 -c 1 -l http://127.0.0.1:8080/
```

It will print the output you're pretty familiar with by now, just like the following (once again, abbreviated). You will notice that the result is a few milliseconds greater than our steady state.

```
(...)
Complete requests:      830
Failed requests:        0
(...)
Time per request:       36.212 [ms] (mean)
(...)
```

Back-of-a-napkin maths warning: that result effectively puts an upper bound on the average amount of overhead our delay injector adds itself (36ms-26ms=10ms per request). Assuming the worst-case scenario, where the database sent a single packet that was delayed by 1ms, that's a theoretic average overhead of 9ms. The average time per request during our experiment was 490ms, or 464ms (490-26) larger than the steady state. Even assuming

that worst case scenario, 9ms overhead, the result would not be significantly different (9/490 ~= 2%).

Long story short: these results are plausible, and that concludes our chaos experiments with a failure. The initial hypothesis was way off. Now, with the data, we have a much better idea of where the slowness might be coming from, and we can debug this further and hopefully fix the issue.

Just one last hint before we leave. List all containers, including the ones that are finished, by running the following command in a terminal window:

```
docker ps --all
```

You will see an output similar to the following one. Notice the pairs of containers started with the image `gaiadocker/iproute2` we specified earlier on with the `--tc-image` flag:

```
CONTAINER ID        IMAGE                 COMMAND                  CREATED
       STATUS                          PORTS                NAMES
9544354cdf9c        gaiadocker/iproute2   "tc qdisc del dev et…"   26 minutes ago
       Exited (0) 26 minutes ago                            stoic_wozniak
8c975f610a29        gaiadocker/iproute2   "tc qdisc add dev et…"   27 minutes ago
       Exited (0) 27 minutes ago                            quirky_shtern
(...)
```

These are the short-lived containers, which executed the tc command from inside the same networking configuration as our target container. You can even inspect one of them, by running a command similar to the following:

```
docker inspect 9544354cdf9c
```

You will see a long JSON file, similar to the following output (abbreviated). Within this file notice two members, Entrypoint and Cmd. They list the entry point binary and its arguments, respectively.

```
(...)
            "Cmd": [
                "qdisc",
                "del",
                "dev",
                "eth0",
                "root",
                "netem"
            ],
(...)
            "Entrypoint": [
                "tc"
            ],
(...)
```

So there we go, another chaos experiment under your belt and another tool in your toolbox. Let's finish by taking a tour of other mention-worthy aspects of chaos engineering relevant to Docker that we haven't covered.

5.5 Other parts of the puzzle

I want to mention some other topics that we haven't covered in detail in this chapter that are worth considering when designing your own chaos experiments. The list is potentially infinite, but let me list just a few common issues.

5.5.1 Docker daemon restarts

In its current model, a restart of Docker daemon means a restart of all applications running on Docker on that host. This might sound obvious and trivial, but it can be a very real problem. Imagine a host running a few hundred containers and Docker crashing:

- How long is it going to take for all the applications to get started again?
- Do some containers depend on others, so the order of them starting is important?
- How do containers react to this situation, in which resources are used to start other containers (thundering herd problem)?
- Are you running infrastructure processes (say overlay network) on Docker? What happens if that container doesn't start before the other ones?
- If it crashes at the wrong moment, can Docker recover from any inconsistent state? Does any state get corrupted?
- Does your load balancer know when a service is really ready, rather than just starting, to know when to serve it traffic?

A simple chaos experiment restarting Docker mid-flight might help you answer all of these questions and many more.

5.5.2 Storage for image layers

Similarly, the storage problems have a much larger scope for failure than we covered. We saw earlier that a simple experiment showed that a default Docker installation on Ubuntu 18.04 doesn't allow for restricting the size of storage a container can use. But in real life a lot more than a single container unable to write to disk can go wrong. For example, consider the following:

- What happens if an application doesn't know how to handle lack of space, crashes, and Docker is unable to restart it due to the lack of space?
- Will Docker have enough storage to download the layers necessary to start a new container you need to start (difficult to predict the total amount of decompressed storage needed).
- How much storage does Docker itself need to start if it crashes when a disk is full?

Again, this might sound basic, but a lot of damage can be caused by a single faulty loop writing too much data to the disk, and running processes in containers might give a false sense of safety in this respect.

5.5.3 Advanced networking

We covered the basics of Docker networking, as well using Pumba to issue tc commands to add delay to interfaces inside of containers, but that's just a tip of the iceberg of what can go

wrong. Although the defaults are not hard to wrap your head around, the complexity can grow very quickly. Docker is often used in conjunction with other networking elements such as overlay networks (for example Flannel https://github.com/coreos/flannel), cloud-aware networking solutions (such as Calico https://www.projectcalico.org/), and service meshes (such as Istio https://istio.io/docs/concepts/what-is-istio/), which further add to the standard tools (like iptables https://en.wikipedia.org/wiki/Iptables and IPVS https://en.wikipedia.org/wiki/IP_Virtual_Server) to further increase the complexity.

We will touch upon some of these in the context of Kubernetes, but understanding how your networking stack works (and breaks) will always be important to anyone practicing chaos engineering.

5.5.4 Security

Finally, the security aspect of things. While it's typically a job of a dedicated team, it's worth exploring these problems using chaos engineering techniques. We briefly mentioned seccomp, SELinux, and AppArmor. Each of them provide layers of security, which can be tested against with an experiment. Unfortunately, these are beyond the scope of this chapter, but there is still typically a lot of low-hanging fruit to look into. For example, all of the following situations can (and do) lead to security issues, and can usually be easily fixed:

- Containers running with `--privileged` flag, often without a good reason
- Running as root inside of the container (default pretty much everywhere)
- Unused capabilities given to containers
- Using random Docker images from the internet, often without peeking inside
- Running ancient versions of Docker images containing known security flaws
- Running ancient versions of Docker itself containing known security flaws

Chaos engineering can help design and run experiments and see how exposed you are to the different threats out there. And if you tune in, you will notice that exploits do appear on a more or less regular basis (for example https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/).

5.6 Summary

- Docker builds on several decades of technology and leverages various Linux kernel functionalities (like chroot, namespaces, cgroups, and others) to make for a simple user experience
- The same tools designed to operate on namespaces and cgroups apply equally to Docker containers
- For effective chaos engineering in a containerized world, it is necessary to have an understanding of how they work (and they're not that scary after you've seen them from up close)
- Pumba (https://github.com/alexei-led/pumba) is a convenient tool for injecting network problems, running stress tests from within a cgroup and killing containers
- Chaos engineering should be applied to both applications running on Docker and to the Docker itself to make both more resilient to failures

6

Who you gonna call? Syscall-busters!

This chapter covers

- **Observing syscalls of a running process using strace and BPF**
- **Working with black-box software, understanding what it does without reading the source code**
- **Designing chaos experiments at the syscall level**
- **Blocking syscalls using strace and seccomp**

It's time to take a deep dive - all the way to the OS - to learn how to do chaos engineering at the syscall level. I want to show you that even in a simple system, like a single process running on a host, you can create plenty of value by applying chaos engineering and learning just how resilient that system is to failure. And oh - it's good fun too!

In this chapter, we'll start with a brief refresher on syscalls. We'll then see how to do the following:

- Understand what a process does without looking at its source code
- List and block the syscalls that it can make
- Experimentally test our assumptions about how it deals with failure.

If I do my job well, you'll finish this chapter with a realization that it's hard to find a piece of software that can't benefit from some chaos engineering, even if it's closed source. Whoa, did I just say "closed source?" The same guy who always goes on about how great open source software is and who maintains some himself? Why would you do closed source? Well, sometimes it all starts with a promotion.

6.1 Scenario - congratulations on your promotion!

Do you remember your last promotion? Perhaps a few nice words, some handshakes, and ego-friendly emails from your boss. And then, invariably, a bunch of surprises you hadn't

thought of when you agreed to take on the new opportunity. There is a certain something that somehow always appears in these conversations, but only after the deal is done: the maintenance of legacy systems.

Legacy systems, like potatoes, come in all shapes and sizes. And just like potatoes, you often won't realize just how convoluted their shape really is until you dig them out of the ground. Things can get messy if you don't know what you're doing! What counts as legacy in one company might be considered pretty progressive in a different setting. Sometimes, there are good reasons to keep the same code base for a long time (for example, the requirements haven't changed, it runs fine on modern hardware, and there is a talent pool) and sometimes software is kept in an archaic state for all the wrong reasons (sunk-cost fallacy, vendor lockdown, good old bad planning, and so on). Even modern code can be considered legacy if it's not well maintained. But there is a particular type of legacy system that I'd like you to look at in this chapter - the kind that works, but no one really knows how. Let's take a look at an example of such a system.

6.1.1 System X: if everyone is using it, but no one maintains it, is it abandonware?

If you've been around for a while, you can probably name a few legacy systems that only certain people really understood inside out, but a lot of people use. Well, let's imagine a situation where the last person knowing a certain system quits, and your promotion includes figuring out what to do with that system to maintain it. It's officially your problem now. Let's call that problem "System X."

First things first, you check the documentation. Oops, there isn't any! Through a series of interviews of the more senior people you find the executable binary and the source code. And thanks to the tribal knowledge, you know that the binary provides an HTTP interface that everyone is using. Figure 6.1 summarizes this rather enigmatic description of the system.

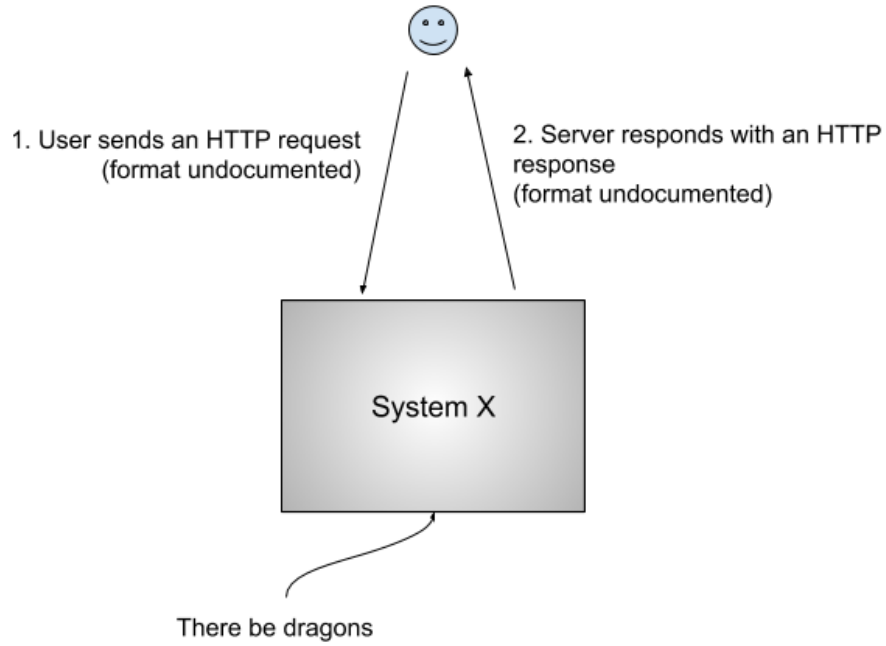


Figure 6.1 The (known part of the) architecture of the legacy System X

Let's take a glance at the source code structure. If you're inside of the VM shipped with this book, you can find it by going to the following folder in a terminal window:

```
cd ~/src/examples/who-you-gonna-call/src/
```

(Alternatively you can browse the code online on GitHub at https://github.com/seeker89/chaos-engineering-book/tree/master/examples/who-you-gonna-call/src). It's a simulated legacy application, written in C. To keep this as realistic as possible, don't dig too deep into how the application is written (it should appear to you awfully complicated for what it's doing). If you're really curious, this source code is generated through `generate_legacy.py` script in the same folder, but I recommend you read it only after we're done with this chapter. I'm not going to walk you through what it's doing, but let's just get a rough idea of how much code goes into the final product. To find all the files and sum up the lines of code, run the following command in a terminal window:

```
find ~/src/examples/who-you-gonna-call/src/ -name "*.c" -o -name "*.h" | sort | xargs wc -l
```

You will see an output similar to the following (abbreviated). Note the total of 3128 lines of code (bold font):

```
   26 ./legacy/abandonware_0.c
  (...)
   26 ./legacy/web_scale_0.c
(...)
```

```
   79 ./main.c
 3128 total
```

Fortunately, it also comes with a Makefile, which allows us to build the binary. Run the following command in a terminal window, from the same directory, to build the binary called `legacy_server`. It will compile the application for you:

```
make
```

After it's done compiling, you will be left with a new executable file, `legacy_server`. You can now start it, by running the following command in a terminal window:

```
./legacy_server
```

It will print a single line to inform you that it started listening on port 8080, like in the following output:

```
Listening on port 8080, PID: 1649
```

You can now confirm that it's working by opening a browser and going to http://127.0.0.1:8080/ You will see the web interface of the legacy System X. It's not a system without which the world would stop spinning, but it's definitely an important aspect of the company culture. Make sure you investigate it thoroughly.

Now, the big question is: given that the legacy System X is a big, black box, how can I sleep well at night, not knowing how it might break? Well, as the title of this book might give away, a little bit of chaos engineering can help!

The purpose of this chapter is to show you how to inject failure on the boundary between the application and the system (something even the most basic of programs will need to do) and see how the application copes when it receives errors from the system. That boundary is defined by a set of syscalls. To make sure we're all on the same page, let's start with a quick refresher on syscalls.

6.2 A brief refresher on syscalls

System calls (more commonly abbreviated to *syscalls*) are the application programming interface (API) of an operating system (OS), such as Unix, Linux, or Windows. For a program running on an OS, syscalls are the way of communicating with the kernel of that OS. If you've ever written so much as a "hello world" program, that program is using a syscall to print the message to your console. What do syscalls do? They give programs access to resources managed by the kernel. Here's a few basic examples:

- `open` - open a file
- `read` - read from a file (or file-like, for instance a socket)
- `write` - write to a file (or file-like)
- `exec` - replace the currently running process with another one, read from an executable file
- `kill` - send a signal to a running process

In a typical modern operating system like Linux, any code executed on a machine runs in either of the following:

- kernel space
- user space (also called userland)

Inside the kernel space, like the name suggests, only the kernel (with its subsystems and most drivers) code is allowed and access to the underlying hardware is granted. Anything else runs inside of the user space, without direct access to the hardware. So if you run a program as a user, it will be executed inside of the user space; when it needs to access the hardware, it will make a syscall, which will be interpreted, validated, and executed by the kernel. The actual hardware access will be done by kernel, and the results made available to the program in the user space. Figure 6.2 sums this process up.

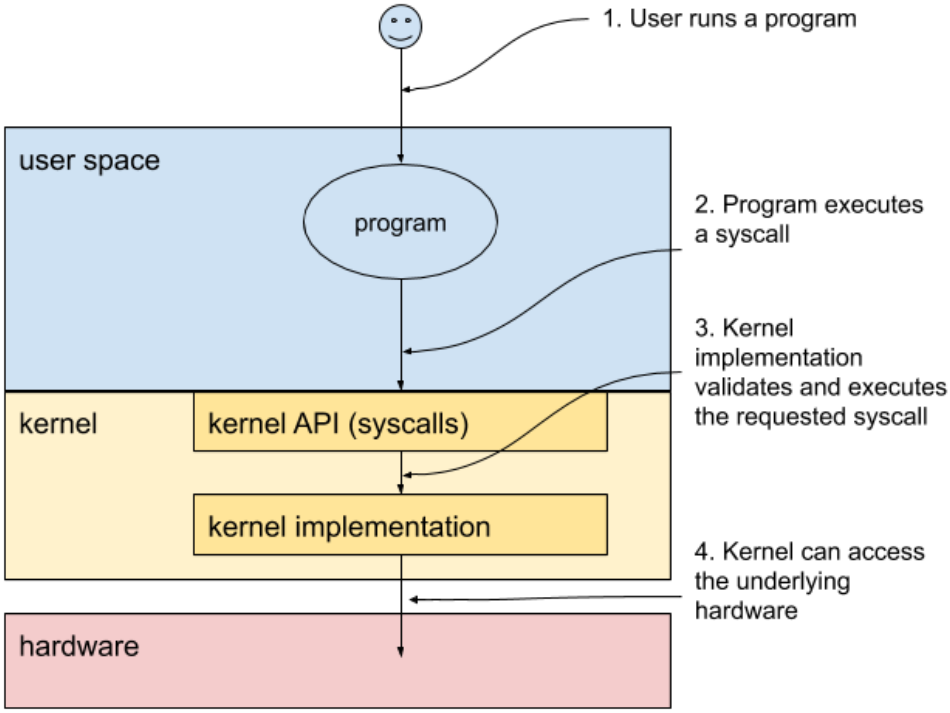


Figure 6.2 Division between kernel space, userland and hardware

Why can't I write a program that directly uses the hardware? Well, the answer is that there is nothing stopping you from writing code directly for particular hardware, but in the modern times it's not very practical. Apart from specialized use cases, like embedded systems or unikernels (https://en.wikipedia.org/wiki/Unikernel; we touched upon this in chapter 5), it just makes more sense to program against a well-defined and documented API, like the

Linux syscalls. All the usual arguments in favor of a well-defined API apply here. Here are a few advantages to this setup:

- *Portability* - an application written against the Linux kernel API will run on any hardware architecture supported by Linux
- *Security* - the kernel will verify the syscalls are legal, and will prevent accidental damage to the hardware
- *Don't reinvent the wheel* - a lot of solutions to common problems (for example virtual memory, filesystems, etc) have already been implemented and thoroughly tested
- *Rich features* - Linux comes with plenty of advanced features, which let the application developer focus on the application itself, rather than having to worry about the low-level, mundane stuff. For example: user management and privileges, drivers for a lot of common hardware or advanced memory management.
- *Speed and reliability* - chances are, that the Linux kernel implementation of a particular feature, tested daily on millions of machines all over the world will be of better quality that one that you'd need to write yourself to support your program

> **NOTE:** Linux is POSIX-compliant (Portable Operating System Interface - https://en.wikipedia.org/wiki/POSIX). This means that a lot of its API is standardized, so you will find the same (or very similar) syscalls in other Unix-like operating systems, for example the BSD family. In this chapter, we will focus on Linux as the most popular representative of this group.

The downside is that there is a small amount of overhead compared with directly accessing the hardware, which is easily outweighed by the upsides for the majority of use cases. Now that you have a high-level idea of what syscalls are for, let's find out which ones are available to you!

6.2.1 Finding out about syscalls

To find out about all the syscalls available in your Linux distribution, we are going to use the `man` command. `man` has the concept of sections, numbered from 1 to 9, that can cover items with the same name in different sections. To find out what the sections are, run the following command in a terminal window:

```
man man
```

You will see an output similar to the following one (abbreviated). Note that section 2 cover syscalls (bold font):

```
(...) A section, if provided, will direct man to look only in that section of the manual.
       The default action is to search in all of the available sections following a pre-
       defined order ("1 n  l  8  3  2 3posix  3pm  3perl  3am  5 4 9 6 7" by default,
       unless overridden by the SECTION directive in /etc/manpath.config), and to show only
       the first page found, even if page exists in several sections.

       The table below shows the section numbers of the manual followed by the types of
       pages they contain.

       1   Executable programs or shell commands
```

```
       2   System calls (functions provided by the kernel)
       3   Library calls (functions within program libraries)
       4   Special files (usually found in /dev)
       5   File formats and conventions eg /etc/passwd
       6   Games
       7   Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
       8   System administration commands (usually only for root)
       9   Kernel routines [Non standard]
```

Therefore to list the available syscalls, run the following command:

```
man 2 syscalls
```

You will see a list of syscalls, along with the version of kernel they were introduced in and notes, just like the following (abbreviated). The numbers in parentheses are the section numbers you can use with man:

```
       System call                 Kernel        Notes
       ──────────────────────────────────────────────────────────────────────
(...)
       chroot(2)                   1.0
(...)
       read(2)                     1.0
(...)
       write(2)                    1.0
```

Let's pick the read syscall as an example. To get more information about that syscall, run the man command in a terminal window, using section two (as instructed by the number in parenthesis):

```
man 2 read
```

You will see the following output (abbreviated again for brevity). Note, that the synopsis contains a code sample in C (bold font), as well as a description of what the arguments and return values mean. This code sample (in C) describes the signature of the syscall in question, and we'll be learning more about that later on:

```
READ(2)                    Linux Programmer's Manual                    READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descriptor fd into the buffer
       starting at buf.
```

Using the man command in section 2, you can learn about any and every syscall available on your machine. It will show you the signature, a description, possible error values and any interesting caveats. From the perspective of chaos engineering, if we want to inject failure into the syscalls a program is making, we first need to build a reasonable understanding of

what purpose they serve. So now we know how to look them up. But how would you go about actually making a syscall? The answer to that question is most commonly `glibc` (https://www.gnu.org/software/libc/libc.html), and using one of the function-wrappers it provides for almost every syscall. Let's take a closer look at how it works.

6.2.2 Standard C library and glibc

A standard C library provides (among other things) an implementation of all the functions whose signatures you can see in section 2 of man pages. These signatures are stored in `unistd.h`, and we have already seen it before. Let's look at a man page of `read(2)` once again, by running the following command:

```
man 2 read
```

You will see the following output in the synopsis section. Notice that the code sample in synopsis includes a header file called `unistd.h`, like in the following output (include in bold font):

```
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);
```

How do we learn more about it? Once again, man pages to the rescue. Run the following statement in a terminal window:

```
man unistd.h
```

In the output of that command, you will learn about all of the functions that should be implemented by a standard C library, just like in the following output. Note the signature of the read function (bold font):

```
(...)
NAME
       unistd.h – standard symbolic constants and types
(...)
   Declarations
       The following shall be declared as functions and may also be defined as macros.
       Function prototypes shall be provided.
(...)
           ssize_t      read(int, void *, size_t);
(...)
```

This is the POSIX standard of what the signature of the syscall wrapper for `read` should look like. Which begs the question: when you write a C program and use one of the wrappers, where is the implementation coming from? `glibc` (https://www.gnu.org/software/libc/libc.html) stands for The GNU C Library and is the most common C library implementation for Linux. It's been around for more than three decades and a lot of software relies on it, despite being criticized for being bloated (http://ecos.sourceware.org/ml/libc-alpha/2002-01/msg00079.html). Note-worthy alternatives include `musl libc` (https://musl.libc.org/) and `diet libc`

(https://www.fefe.de/dietlibc/), both of which focus on reducing the footprint. To learn more, check out `libc(7)` man pages.

In theory, what these wrappers provided by glibc do is just invoke the syscall in question in the kernel and call it a day. In practice, there is a sizable portion of the wrappers which adds code to make the syscalls easier to use. In fact, it's easy to check. In glibc source code, there is a list of "pass-through" syscalls, for which the C code is automatically generated using a script. For example, for version 2.23, you can see the list at https://github.molgen.mpg.de/git-mirror/glibc/blob/glibc-2.23/sysdeps/unix/syscalls.list. That this list contains only 100 of the 380 or so, meaning that almost three quarters of them contain some auxiliary code.

A common example is the `exit(3)` glibc syscall, which adds the possibility to call any functions pre-registered using `atexit(3)` before executing the actual `_exit(2)` syscall to terminate the process. So it's worth remembering that there isn't necessarily a one-to-one mapping between the functions in the C library and the syscalls they implement.

Finally, you will notice, that the argument names might differ between the documentation of glibc and man pages in section 2. That doesn't matter in C, but you can use section 3 of man pages (for example `man 3 read`) to display the signatures from the C library, instead of `unistd.h`.

With this new information, it's time to upgrade our figure 6.2. Figure 6.3 contains the updated version, with the addition of libc for a more complete image. The user runs a program, the program executes a libc syscall wrapper, which in turns makes the syscall. The kernel then executes the requested syscall and accesses the hardware.

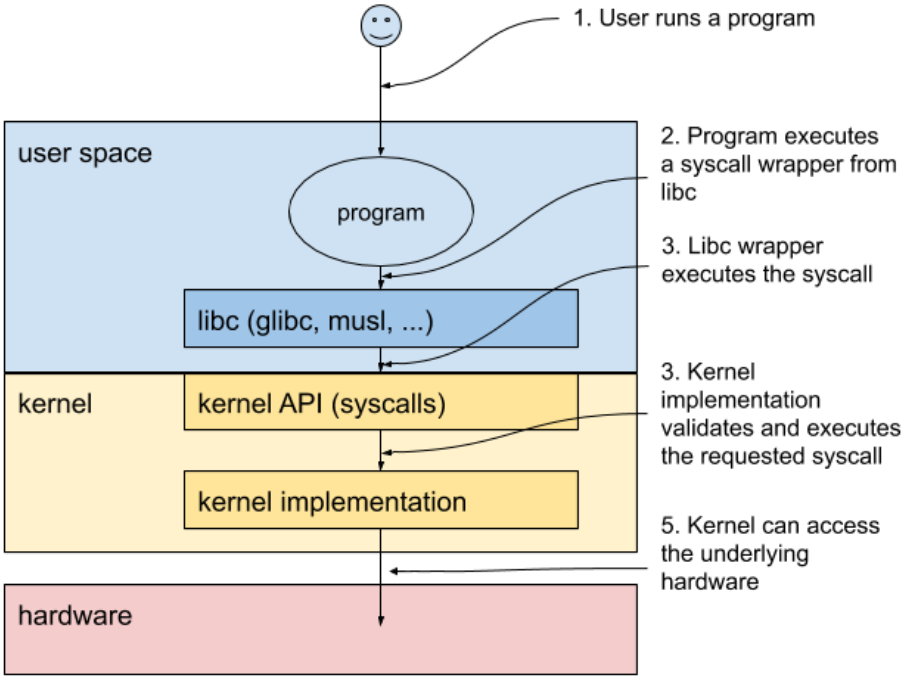


Figure 6.3 User space, libc, kernel space and hardware

A final thought I'd like to plant in your brain is that libc doesn't mean it's only relevant when writing software in C. In fact, it's likely to be relevant to you regardless of what programming language you use, and that's why using a Linux distribution relying on musl libc (like Alpine Linux) might sometimes bite you in the neck when you least expect it (for example see https://github.com/gliderlabs/docker-alpine/issues/367).

With that, I think that we've covered all the necessary theory, and it's time to get our chaos-engineering-wielding hands dirty! We know what syscalls are, how to look up their documentation and what happens when a program makes once. The next question becomes: "apart from reading through the entirety of the source code, how do I know what syscalls a process is making?" Let's cover two ways of achieving that - strace and BPF.

6.3 How to observe a process' syscalls?

For the purpose of chaos engineering, we need to first build a good understanding of what a process does before we can go and design experiments around it. Let's dive in and see what syscalls are being made by using the `strace` (https://strace.io/) command. Let's go through a concrete example of what strace output looks like.

6.3.1 strace and sleep

Let's start with the simplest example I can think of - let's trace what syscalls are made when we run `sleep 1`, a command that does nothing but sleep for one second. To do that, we can just prepend the command we want to run with *strace*. Run the following command in a terminal window (note that we'll need sudo privileges to use strace):

```
sudo strace sleep 1
```

You will see an output similar to the following. Don't worry: it might look daunting at first, but I want you to get through this once, so that you see it's not that complicated. The command we've just run starts a program we requested (sleep) and prints a line per syscall that is made by that program. In each line, it prints the syscall name, the arguments, and the returned value after the "=" sign. There are 12 unique syscalls executed, and `nanosleep` (providing the actual sleep) is the last one on the list. Let's walk through this output bit by bit (I used a bold font for the first instance of a syscall in the output to make it easier to focus on the new syscalls each time).

We start with `execve`, which replaces the current process with another process from an executable file. Its three arguments are the path to the new binary, a list of command line arguments, and the process environment, respectively. This is how the new program is started. It's then followed by `brk` syscall, which reads (when the argument is NULL, like it is in our example) or sets the end of the process's data segment.

```
execve("/bin/sleep", ["sleep", "1"], 0x7fff3f195908 /* 17 vars */) = 0
brk(NULL)                               = 0x557cd8060000
```

To check user's permissions to a file, the `access` syscall is used. You can ignore the call to `/etc/ld.so.nohwcap`. If present, `/etc/ld.so.preload` is used to read the list of shared libraries to preload. Use `man 8 ld.so` for more details on these files. In our case, both calls return value -1, meaning that the files don't exist.

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
```

Next, `openat` is used to open a file (the "at" postfix means the variant that handles relative paths, that the regular `open` doesn't do) and returns a file descriptor number 3. `fstat` is then used to get the file status, using that same file descriptor.

```
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=80887, ...}) = 0
```

Next, `mmap` syscall creates a map of the same file descriptor 3 into virtual memory of the process and the file descriptor is closed using the `close` syscall. mmap is an advanced topic which is not relevant to our goal here, you can read more about how it works at https://en.wikipedia.org/wiki/Mmap

```
mmap(NULL, 80887, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ffb65187000
close(3)                                = 0
```

For some reason, the program tries to access `/etc/ld.so.nohwcap` once again, failing again.

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
```

Next, it opens the libc shared object file at `/lib/x86_64-linux-gnu/libc.so.6`, with file descriptor 3 being reused.

```
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
```

Next, it reads from the libc shared object file (file descriptor 3) to a buffer using a `read` syscall. The display here is a little bit confusing, because the second parameter is the buffer to which the read syscall will write, so displaying its contents doesn't make much sense. The returned value is the number of bytes read, in this case 832. fstat is used once again to get the file status.

```
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030544, ...}) = 0
```

Then it gets a little bit fuzzy. `mmap` is used again to map some virtual memory, including some of the libc shared object file (file descriptor 3). `mprotect` syscall is used to protect some portion of that mapped memory from reading. The `PROT_NONE` flag means that the program can't access that memory at all. Finally, file descriptor 3 is closed with a `close` syscall. For our purposes, you can consider this boilerplate:

```
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7ffb65185000
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ffb64b83000
mprotect(0x7ffb64d6a000, 2097152, PROT_NONE) = 0
mmap(0x7ffb64f6a000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,
       0x1e7000) = 0x7ffb64f6a000
mmap(0x7ffb64f70000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1,
       0) = 0x7ffb64f70000
close(3)                                = 0
```

Next `arch_prctl` is used to set some architecture-specific process state (we can ignore it), `mprotect` is used to make some virtual memory read-only (flag `PROT_READ`) and munmap is used to remove the mapping of the address `0x7ffb65187000`, the one which was mapped to the file `/etc/ld.so.cache` earlier on. All of these operations return value 0 (success).

```
arch_prctl(ARCH_SET_FS, 0x7ffb65186540) = 0
mprotect(0x7ffb64f6a000, 16384, PROT_READ) = 0
mprotect(0x557cd6c5e000, 4096, PROT_READ) = 0
mprotect(0x7ffb6519b000, 4096, PROT_READ) = 0
munmap(0x7ffb65187000, 80887)           = 0
```

Next, the program first reads, and then tries to move the end of the process's data segment, effectively increasing the memory allocated to the process, using `brk`.

```
brk(NULL)                               = 0x557cd8060000
brk(0x557cd8081000)                     = 0x557cd8081000
```

Next, it opens `/usr/lib/locale/locale-archive`, checks its stats, maps it to the virtual memory and closes it.

```
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=3004464, ...}) = 0
```

```
mmap(NULL, 3004464, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ffb648a5000
close(3)                                = 0
```

Then (finally!) we get to the actual meat of things, which is a single `nanosleep` syscall, passing 1 second as an argument.

```
nanosleep({tv_sec=1, tv_nsec=0}, NULL)  = 0
```

Finally, it closes file descriptors 1 (standard output, or stdout) and 2 (standard error, or stderr), just before the program terminates.

```
close(1)                                = 0
close(2)                                = 0
```

And we're through! As you can see, for this simple program, it spent much longer doing things we didn't explicitly ask it to do, rather than what we asked (sleep). If you want to learn more about any of these syscalls, remember that you can just run `man 2 syscall-name` in a terminal window. Just one more thing I wanted to show you is the count summary that strace can produce. If you rerun the strace command, but this time add -C and -S count flags, it will produce a summary sorted by the count of each syscall. Run the following command in a terminal window:

```
sudo strace \
-C \                        #A
-S calls \                  #B
sleep 1
```

#A produce a summary of syscalls
#B sort that summary by the count

After the previous output, you will see a summary similar to the following (our single call to nanosleep in bold):

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         6           mmap
  0.00    0.000000           0         5           close
  0.00    0.000000           0         4           mprotect
  0.00    0.000000           0         3           fstat
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         3         3 access
  0.00    0.000000           0         3           openat
  0.00    0.000000           0         1           read
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         1           nanosleep
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                    32         3 total
```

Which once again shows us that the syscall we actually cared about is only 1 in 32. Equipped with this new toy, let's go and take a look at what our legacy System X does under the hood!

6.3.2 strace and System X

Let's use strace on the legacy System X binary to see what syscalls it makes. You now know how to start a new process with strace, let's also learn how to attach to one that's already running. We're going to use two terminal windows. In the first window, start the legacy_server binary we compiled earlier on. To do that, run the following command:

```
~/src/examples/who-you-gonna-call/src/legacy_server
```

You will see an output similar to the following, printing the port number it listens on and its PID. Note the PID, you can use it to attach to the process with strace (bold font).

```
Listening on port 8080, PID: 6757
```

In a second terminal window, let's use strace to attach to that PID. Run the following command to attach to your the legacy system:

```
sudo strace -C \
-p $(pidof legacy_server)                    #A
```

#A flag -p attaches to an existing process with the given PID

Now, back in the browser, go to (or refresh) http://127.0.0.1:8080/. After you've done that, go back to the second terminal window (the one with strace) and see the output. You will see something similar to the following (abbreviated). This gives us a pretty good idea of what the program is doing. It accepts a connection with `accept`, writes a bunch of data with `write` and closes the connection with `close` (all three in bold font).

```
accept(3, {sa_family=AF_INET, sin_port=htons(53698), sin_addr=inet_addr("127.0.0.1")},
       [16]) = 4
read(4, "GET / HTTP/1.1\r\nHost: 127.0.0.1:"..., 2048) = 333
write(4, "HTTP/1.0 200 OK\r\nContent-Type: t"..., 122) = 122
write(4, "<", 1)                            = 1
write(4, "!", 1)                            = 1
write(4, "d", 1)
(...)
fsync(4)                                    = -1 EINVAL (Invalid argument)
close(4)                                    = 0
```

You might have noticed that it has a bug: it tries to `fsync` (synchronize a file's in-core state with storage device) a file, and it gets back an error `EINVAL (Invalid argument)`. You can now press Ctrl-C to detach strace, and print the summary, like the following one. We can also see that it does a whole lot of writes (292 to be precise), almost all of which write only a single character. More than 98% of the time is spent writing data (in bold font).

```
 <detached ...>
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 98.34    0.002903          10       292           write
  0.68    0.000020          20         1           close
  0.61    0.000018          18         1           accept
  0.34    0.000010          10         1           read
  0.03    0.000001           1         1         1 fsync
------ ----------- ----------- --------- --------- ----------------
```

```
100.00    0.002952                   296         1 total
```

Notice that by attaching strace to a running process, we're only sampling the syscalls that process made while we were attached to it - which makes it easier to work through, but will miss any potentially important initial setup the program might have done.

So far, so good! Using strace has been very straightforward. Unfortunately, it also has its downsides, amongst which the biggest one is the overhead. Let's zoom in on that.

6.3.3 strace's problem - overhead

The dark side of strace is the performance hit that it adds to the traced process. It's not really a secret: this comes directly from man strace(1) pages:

```
BUGS
       A traced process runs slowly.
```

Here's a good example I'm borrowing from Brendan Gregg's blog post that I recommend reading (it comes with a bunch of useful, accurately-titled one-liners and it's overall hilarious): http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html.

`dd` is a simple Linux utility that copies a certain amount of bytes from one file to another, using chunks of desired size. Its simplicity makes it a good candidate for testing the speed of syscalls - it does very little other than make read syscalls followed by write syscalls. Thus, by reading from an infinite source, like the `/dev/zero` (returns zeros for every read) and writing to `/dev/null` (discards the written bytes), we can stress-test the speed of `read` and `write` syscalls.

Let's do just that. First, let's see how quickly it can go without strace attached to it. Let's make 500k operations (an arbitrary number that should be big enough to last a few hundred milliseconds, but small enough to not bore you to death) and writes of size one byte (the smallest amount we can write, for a maximum number of operations), by running the following command in a terminal window:

```
dd if=/dev/zero of=/dev/null bs=1 count=500k
```

You will see an output similar to the following, taking about half a second (bold font) to perform that operation:

```
512000+0 records in
512000+0 records out
512000 bytes (512 kB, 500 KiB) copied, 0.509962 s, 1.0 MB/s
```

Now, let's rerun the same command, but trace it with strace. And let's use the `-e` flag to filter only `accept` syscall, which dd doesn't even use (to show that just the action of attaching strace is already adding the overhead, even if it's on an unrelated syscall). Run the following command in a terminal window:

```
strace \
-e accept \                                              #A
dd if=/dev/zero of=/dev/null bs=1 count=500k
```

#A only print the accept syscalls (which dd doesn't make)

You will see an output similar to the following. In my example, it took 58.5 seconds (bold font), or a more than 100-fold slowdown, compared to the values we saw without strace:

```
512000+0 records in
512000+0 records out
512000 bytes (512 kB, 500 KiB) copied, 58.4923 s, 8.8 kB/s
+++ exited with 0 +++
```

That means that it might be fine to use strace in a test environment, like we're doing now, but attaching it to a process running in production can have very serious consequences. It also means that if we were looking into the performance of a program traced with strace, all our numbers would be off.

All of that limits the use cases for strace, but fortunately there are other options. Let's look at an alternative - the Berkeley Packet Filter (BPF).

> **PTRACE SYSCALL** As a side note to what we've covered about strace, I bet you're wondering what the underlying mechanism is that allows it to control and manipulate other processes. The answer is the `ptrace` syscall. You don't need to know how it works to get the value out of using strace, but for the curious reader, check out the man page of ptrace(2). There is also a good intro on Wikipedia: https://en.wikipedia.org/wiki/Ptrace

6.3.4 BPF

BPF was initially designed to filter network packets. It has since been extended (extended Berkeley Packet Filter, or eBPF) to become a generic Linux kernel execution engine, which allows for writing programs with guarantees of both safety *and* performance. When talking about BPF, most people refer to the extended version. In the context of chaos engineering, BPF will often come in handy to produce metrics for our experiments.

One of the most exciting things about BPF is that it allows for writing very efficient programs executed during certain events in the Linux kernel. Together with the limits enforced on these programs to limit the time they can take and the memory they can access, as well as built-in efficient aggregation primitives, BPF is an amazing tool to gain visibility into what's going on at the kernel level. What is exciting for our chaos engineering needs is that unlike strace, it is often possible to achieve that insight (for example trace all the syscalls) with a minimal overhead.

The downside of BPF is that the learning curve is pretty steep. In order to write a meaningful program looking into the Linux kernel internals, it's routinely necessary to look into how things are implemented in the kernel itself. Although the time investment pays off, it can be a little daunting at first. Fortunately, there are now a few projects that make that introduction much easier, like BCC (BPF Compiler Collection). Let's take a look at how BCC can help in the practice of chaos engineering.

6.3.4.1 BPF AND BCC

BCC (https://github.com/iovisor/bcc) is a framework that makes it easier to write and run BPF programs, providing wrappers in Python and Lua and many useful tools and examples. Reading through these tools and examples is currently the best way of starting with BPF that I can think of.

We covered a few of the BCC tools in chapter 3 (biotop, tcptop, oomkill), and now I'd like to bring another one to your attention: `syscount`. Your VM comes with the tools pre-installed, but installing them on Ubuntu is as easy as running the following command from a terminal (check appendix A for more information):

```
sudo apt-get install bpfcc-tools linux-headers-$(uname -r)
```

In the previous section, we used strace to produce a list of syscalls a program made. It worked well, but it had one serious problem - strace introduced a large overhead to the program it was tracing. Let me show you how we can get the same list without the overhead, by leveraging BPF and BCC through the tool syscount.

Let's start by getting used to using syscount. In its simplest form, it will count all syscalls of all the processes currently running, and will print the top 10 when you press Ctrl-C. Run the following command in a terminal window to do that (remember that on Ubuntu, the BCC tools are postfixed with `-bpfcc`):

```
sudo syscount-bpfcc
```

After a few seconds, press Ctrl-C to stop the process and you will see an output just like the following. You will recognize some of the syscalls on the list, like write and read (bold font). It's a list counting all syscalls made by all of the processes on the host during the time syscount was running:

```
Tracing syscalls, printing top 10... Ctrl+C to quit.
^C[20:12:40]
SYSCALL                   COUNT
recvmsg                   42057
futex                     35200
poll                      12730
epoll_wait                 6816
write                      6005
read                       5971
writev                     4200
setitimer                  2957
mprotect                   2748
sendmsg                    2631
```

Now, let's verify this claim about low overhead. Remember how in the previous section just using strace on the process slowed it down by a factor of 100, even though we were targeting a syscall that program wasn't making? Let's compare how BPF fares. To do that, let's open two terminals. In the first one, we'll run the syscount command again, and in the other one we'll rerun the same dd one-liner we used. Ready? Start by running the syscount in the first terminal, by typing:

```
sudo syscount-bpfcc
```

Then, from a second terminal window, let's run the dd again:

```
dd if=/dev/zero of=/dev/null bs=1 count=500k
```

When it's done, you will see an output just like the following in the second terminal. Notice that the total time of executing the half million read and write syscalls each took slightly longer than previously (0.509s), 0.54s in my example:

```
512000+0 records in
512000+0 records out
512000 bytes (512 kB, 500 KiB) copied, 0.541597 s, 945 kB/s
```

0.541597s vs 0.509962s is about 6% overhead, and that's for a close-to-worst-case scenario, where dd doesn't do much more than read and write. And we've been tracing everything that's happening on the kernel, not just a single PID.

Now that we've confirmed the overhead is much more acceptable for BPF, compared to strace, let's go back to our chaos engineering use case: learning how to get a list of syscalls made by a process. Let's see how to use syscount to show the top syscalls for a specific PID, using the `-p` flag. To do that, let's once again use two terminal windows. In the first one, start the legacy_server by running the following command:

```
~/src/examples/who-you-gonna-call/src/legacy_server
```

In a second terminal window, start the syscount command, but this time with the -p flag, by running the following command:

```
sudo syscount-bpfcc \
-p $(pidof legacy_server)                    #A
```

#A Trace only the calls for pid of our legacy server

You will see an output just like in table 6.1. Note that it matches the summary of the output we've gotten from strace, with the 292 calls to write, although it provides fewer details.

Table 6.1 Output of syscount-bpfcc side-by-side with the output of strace

syscount-bpfcc	strace
Tracing syscalls, printing top 10... Ctrl+C to quit. ^C[20:39:19] SYSCALL COUNT write 292 accept 1 read 1 close 1	% time seconds usecs/call calls errors syscall ------ ----------- ----------- --------- --------- ---------- 98.34 0.002903 10 292 write 0.68 0.000020 20 1 close 0.61 0.000018 18 1 accept 0.34 0.000010 10 1 read 0.03 0.000001 1 1 1

```
fsync
1
```

```
fsync
------ ----------- ----------- --------- --------- -
---------------
100.00    0.002952                   296         1
total
```

And voila! Using this technique, you can now list syscalls that a process makes, without the overhead the strace introduces. Note, that `syscount-bpfcc` only gives you a count, without the details that strace was printing for each syscall, but it will be sufficient if we only need a rough idea of what a process is doing. As always, when designing your chaos experiment, pick the right tool for the job.

I'd love to talk to you more about BPF (and I'm sure we will, if we bump into each other at the next conference), but it's time to move on. If you feel like you need more BPF in your life, have a read through the source code of syscount. It's only a single *`less $(which syscount-bpfcc)`* (or https://github.com/iovisor/bcc/blob/master/tools/syscount.py) away! In the meantime, let's make a few other honorable mentions to alternative tools you might be able to use to get similar results.

6.3.5 Other options

I wanted to make you aware of other related technologies that are available to use to gain a similar level of visibility. Unfortunately, we won't get into the details, but it's worth having them on your radar. Let's take a look.

6.3.5.1 SystemTap

SystemTap (https://sourceware.org/systemtap/) is a tool for dynamically instrumenting running Linux systems. It uses a domain-specific language (that looks much like awk or C, read more at https://sourceware.org/systemtap/man/stap.1.html) to describe various kinds of probes. The probes are then compiled and inserted into a running kernel. The original paper describing the motivations and architecture can be found at https://sourceware.org/systemtap/archpaper.pdf. There is an overlap between SystemTap and BPF, and the work to use BPF as a backend for SystemTap is called stapbpf.

6.3.5.2 ftrace

Ftrace (https://www.kernel.org/doc/Documentation/trace/ftrace.txt) is another framework for tracing the Linux kernel. It allows for tracing many events happening in the kernel, both statically and dynamically defined. It requires a kernel built with ftracer support and has been part of the kernel code base since 2008.

With that, we're ready to design some chaos experiments!

6.4 Blocking syscalls for fun and profit part 1 - strace

Let's put our chaos engineering hat on and design an experiment which will tell us how our legacy application fares when it gets errors trying to make syscalls. So far we've looked a little bit under the hood of what the black box System X binary is doing, all without reading the source code. We established that during an HTTP request from a browser, it makes a small number of syscalls, like in the following output:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 98.34    0.002903          10       292           write
  0.68    0.000020          20         1           close
  0.61    0.000018          18         1           accept
  0.34    0.000010          10         1           read
  0.03    0.000001           1         1         1 fsync
------ ----------- ----------- --------- --------- ----------------
100.00    0.002952                   296         1 total
```

To warm up, let's start with something simple: let's pick the `close` syscall, which is called only a single time in our initial research, and see whether System X handles a situation in which `close` returns an error. What could possibly go wrong? Let's find out.

6.4.1 Experiment 1: breaking the close syscall

As always, we start with the observability. Luckily, we can once again use the ab command, which will allow us to generate some traffic and summarize statistics about the latencies, throughput, and the number of failed requests. And because we have no information about it, apart from the fact that the system has been running live for years and years, let's assume there will be no requests if we introduce failure on the close syscall. Therefore, we can devise the four simple steps to run a chaos experiment like the following:

1. Observability: use ab to generate traffic, read the number of failures and latencies
2. Steady state: read ab numbers for System X under normal conditions
3. Hypothesis: if we make calls to close fail for the System X binary, it will handle it gracefully, transparently to the end user
4. Run the experiment!

You're familiar with that ab command, you know how to trace a process with strace, so the question now becomes, "How do we introduce failure into a syscall for the System X binary?" Fortunately, strace makes it easy through the use of the `-e` flag. Let's learn how to use the `-e` flag, by looking into the help of strace. To do that, run thestrace command with the `-h` flag:

```
strace -h
```

You will see the following output (abbreviated). In particular, notice the fault option (bold font).

```
(...)
  -e expr        a qualifying expression: option=[!]all or option=[!]val1[,val2]...
     options:    trace, abbrev, verbose, raw, signal, read, write, fault
```

```
(...)
```

By default, running with a flag `-e fault=<syscall name>` returns an error (-1) on every call to the desired syscall. To inject failure into the `close` syscall, we can use the `-e fault=close` flag. This is the most popular form. But there is another, more flexible flag that we can use, although weirdly it's not mentioned by `strace -h`, and that's `-e inject`. To learn about it, we need to read the man pages for strace, by running the following command:

```
man strace
```

You will see much more detail on how to use strace. In particular, note the section describing the `-e inject` option (in bold font) and its syntax:

```
(...)
       -e inject=set[:error=errno|:retval=value][:signal=sig][:when=expr]
              Perform syscall tampering for the specified set of syscalls.

(...)
```

In fact, the flag is pretty powerful, and supports the following arguments:

- `fault=<syscall>` - injects a fault into a particular syscall
- `error=<error name>` - specifies a particular error to return
- `retval=<return code>` - overrides the actual syscall return value and send the specified one instead
- `signal=sig` - sends a particular signal to the traced process
- `when=<expression>` - controls which calls are affected and can take three forms:
    - `when=<n>` - only tampers with the nth syscall
    - `when=<n>+` - tampers with the nth and all the subsequent calls
    - `when=<n>+<step>` - tempers with the nth, and every one in step occurrences after that

For example, the following flag fails every write syscall, starting with the second one, by injecting EACCES error (permission denied) as the return value:

```
-e inject=write:error=EACCES:when=2+
```

The following flag, on the other hand, overrides whatever the result of the first syscall to fsync is (even if it was an error response) and returns a value of 0 instead:

```
-e inject=fsync:retval=0:when=1
```

All of this together gives us a fairly fine-grained control over what happens to the process on the syscall level. The price? Well, once again, the overhead. We need to keep in mind that to compare apples to apples, we'll also need to establish our steady state, including the overhead of strace. But as long as we do that, we should be ready to implement the experiment. Let's do it!

6.4.2 Experiment 1: steady state

First, let's establish the steady state. We'll use three terminal windows: System X in the first one, strace in the second, and ab in the third. Let's start the `legacy_server` (System X binary) in the first window by running the following command:

```
~/src/examples/who-you-gonna-call/src/legacy_server
```

Next, let's attach strace to the `legacy_server` in the second terminal window, for now without any failures, and tracing only the close syscalls. Do that by running the following command:

```
sudo strace \
-p $(pidof legacy_server) \
-e close                                    #A
```

#A display only the close syscall

Finally, let's start ab in the third window, by running the following command. We'll use a concurrency of 1 to keep things simple, and run for up to 30 seconds:

```
ab -c1 -t30 http://127.0.0.1:8080/
```

In the same third window, you will see results similar to the following. Note that of the ~3000 complete requests, none were failed, and we achieved about 101 requests per second (all three in bold font):

```
(...)
Time taken for tests:   30.003 seconds
Complete requests:      3042
Failed requests:        0
(...)
Requests per second:    101.39 [#/sec] (mean)
(...)
```

So that's our steady state: no failures and about 100 requests per second. To be sure, you could run the ab a few times and see how much the values vary between runs. Now, to the fun part - implementation time!

6.4.3 Experiment 1: implementation

Let's see what happens, when the legacy System X gets errors on `close` syscall. To do that, let's keep the same setup with three terminal windows, but in the second one, close strace (press Ctrl-C) and restart it with `-e inject` option, by running the following command:

```
sudo strace \
-p $(pidof legacy_server) \
-e close \
-e inject=close:error=EIO                   #A
```

#A add failure to the close syscall, use error EIO

Now, in the third terminal window, let's start ab again with the same command by running:

```
ab -c1 -t30 http://127.0.0.1:8080/
```

This time, the output will be different, like the following. Our ab isn't even able to finish its run: it's getting an error (bold font):

```
(...)
Benchmarking 127.0.0.1 (be patient)
apr_socket_recv: Connection refused (111)
Total of 1 requests completed
```

If you switch back to the second window with strace, you will see that it injected the error we asked for, and that the application then exited with error code 1, just like in the following output. It also exited at the very first call to close (number of calls and errors in bold font):

```
close(4)                                = -1 EIO (Input/output error) (INJECTED)
+++ exited with 1 +++
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         1         1 close
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000                     1         1 total
```

And back in the first window, the application printed an error message and crashed with the following output:

```
legacy_server: error closing socket: Input/output error
```

What does it mean? Well, our experiment hypothesis was wrong. Let's analyse these findings.

6.4.4 Experiment 1: analysis

What we learned is that the application doesn't handle failure gracefully when making the `close` syscall - it exits with an error code of 1, signalling a generic error. We still haven't looked into the source code, so we can't be sure why its authors decided to implement it that way, but using this very simple experiment, we have already found a fragile point. How fragile? Let's see what the man pages tell us about the `close` syscall by running the following command in a terminal:

```
man 2 close
```

If you scroll to the ERRORS section of the output, you will see the following output:

```
ERRORS
       EBADF  fd isn't a valid open file descriptor.

       EINTR  The close() call was interrupted by a signal; see signal(7).

       EIO    An I/O error occurred.

       ENOSPC, EDQUOT
              On NFS, these errors are not normally reported against the first write which
       exceeds the available storage space, but instead  against
              a subsequent write(2), fsync(2), or close(2).
```

This information can be summarized to four possibilities:

1. the argument is not an open file descriptor
2. the call was interrupted by a signal
3. I/O error occurred
4. NFS write error is reported against a subsequent close, instead of a write

Again, without even reading through the source code we can make an educated guess that at least option 2 is possible, because any process could be interrupted by a signal. And now we know that this kind of interruption might cause the legacy System X to go down. Fortunately we can test it by injecting that specific error code to see if the program handles it correctly.

Now that we know about this, we could try to find the place in the source code which handles this part and make it more resilient to failure. That would definitely help the newly promoted you sleep better. But let's not elect to rest on our laurels quite yet. I wonder what happens when failure occurs on one of the more busy syscalls - for example, `write`?

6.4.5 Experiment 2: breaking the write syscall

Recalling our handy table of syscalls, the legacy System X spent most of its time making write syscalls (in bold font), as shown in the following output:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 98.34    0.002903          10       292           write
  0.68    0.000020          20         1           close
  0.61    0.000018          18         1           accept
  0.34    0.000010          10         1           read
  0.03    0.000001           1         1         1 fsync
------ ----------- ----------- --------- --------- ----------------
100.00    0.002952                   296         1 total
```

Surely, for a piece of software that might predate our tenure at the company, there must be some kind of resilience and fault tolerance built-in, right? Well, let's find out! Much like the previous experiment, let's use ab and strace, but let's fail only every other call to write. Our experiment then becomes:

1. Observability: use ab to generate traffic, read the number of failures and latencies for System X
2. Steady state: read ab numbers under normal conditions
3. Hypothesis: if we make every other call to `write` fail for the System X binary, it will handle it gracefully, transparently to the end user
4. Run the experiment!

If this sounds like a plan, let's go and do it.

6.4.6 Experiment 2: steady state

Again, let's start by establishing the steady state. We'll use three terminal windows again: System X in the first one, strace in the second, and ab in the third. Let's start the `legacy_server` (System X binary) in the first window by running the following command:

```
~/src/examples/who-you-gonna-call/src/legacy_server
```

Next, let's attach strace to the `legacy_server` in the second terminal window, for now without any failures, and only tracing the write syscalls. Do that by running the following command:

```
sudo strace \
-p $(pidof legacy_server) \
-e write                                    #A
```

#A only display the write syscall

Finally, let's start ab in the third window, by running the following command. We'll use a concurrency of 1 to keep things simple, and run for up to 30 seconds:

```
ab -c1 -t30 http://127.0.0.1:8080/
```

In the same third window, you will see results similar to the following. Similar to the previous experiment, there should be no failures, but the throughput will be lower (bold font), due to more print operations at the terminal:

```
(...)
Complete requests:      1587
Failed requests:        0
(...)
```

Our steady state is similar to the one from the previous experiment; that shouldn't be a surprise. Let's now get to the fun part - the actual implementation of the failure injection for the experiment 2.

6.4.7 Experiment 2: implementation

The fun should start, when the legacy System X gets errors on `write` syscall. To do that, let's keep the same setup with three terminal windows. And just like the last time, in the second one, close strace (press Ctrl-C) and restart it with `-e inject` option to add the failure we designed (fail every other write syscall), by running the following command:

```
sudo strace \
-p $(pidof legacy_server) \
       -C \                                         #A
-e inject=write:error=EIO:when=1+2                  #B
```

#A display a summary at the end of the session
#B add failure to the close syscall, use error EIO, fail on every other call starting with the first one

Now, in the third terminal window, let's start ab again with the same command by running:

```
ab -c1 -t30 http://127.0.0.1:8080/
```

This time, we're in for a pleasant surprise. You will see an output similar to the following. Despite every other syscall failing, overall there are still no failed requests (bold font). But the throughput is roughly halved, at 570 requests in my example (also bold font):

```
(...)
Time taken for tests:   30.034 seconds
Complete requests:      570
```

```
Failed requests:        0
(...)
```

In the second window you can now kill strace by pressing Ctrl-C. Take a look at the output. You will see a lot of lines similar to the following output. You can clearly see that the program retries failed writes, because each write is done twice, first receiving the error we inject, and then succeeding:

```
(...)
write(4, "l", 1)                        = -1 EIO (Input/output error) (INJECTED)
write(4, "l", 1)                        = 1
write(4, ">", 1)                        = -1 EIO (Input/output error) (INJECTED)
write(4, ">", 1)                        = 1
(...)
```

It means that the program implements some kind of algorithm to account for failed write syscalls, which is good news - one step closer to getting paged less at night. We can also see the cost of the additional operations: the throughput is roughly 50% of what it was without the retries. In real life, it's unlikely that every other `write` would fail, but even in this nightmare-ish scenario System X turns out to not be as easy to break as it was with the `close` syscall.

And that concludes our experiment 2 - this time our hypothesis was correct. High five! You've learned how to discover what syscalls are made by a process and how to tamper with them to implement experiments using strace. And in our case, focused on whether System X keeps working, rather than how quickly it responds, it all worked out.

But we still have one skeleton in the closet - the overhead of strace. What can we do, if we want to block some syscalls, but can't accept the massive slowdown while doing the experiment? Before we wrap up this chapter, I'd like to point out an alternative solution for syscalls blocking - using seccomp.

6.5 Blocking syscalls for fun and profit part 2 - seccomp

You'll remember seccomp from the chapter on Docker as a layer of hardening the containers by restricting the syscalls that they can make. I would like to show you how you can leverage seccomp to implement experiments similar to what we've done with strace by blocking certain syscalls. We'll do it the easy way and the hard way, each covering a different use case. The easy way will be quick, but not very flexible. The hard way will be more flexible, but will require more work. Let's start with the easy way.

6.5.1 Seccomp the easy way - Docker

An easy way to block some syscalls is to leverage a custom seccomp profile when starting a container. Probably the easiest way of achieving this is to download the default seccomp policy (https://github.com/moby/moby/blob/master/profiles/seccomp/default.json) and remove the syscall that you'd like to disable.

The profile has the following structure. It's a list of allowed calls; by default, all calls are blocked and return error when called (the `SCMP_ACT_ERRNO` default action) . Then a long list of names is explicitly allowed:

```
{
       "defaultAction": "SCMP_ACT_ERRNO",                                #A
...
       "syscalls": [
         {
                 "names": [                                              #B
                         "accept",
                         "accept4",
...
                         "write",
                         "writev"
                 ],
                 "action": "SCMP_ACT_ALLOW",                             #C
...
         },
...
       ]
}
```

#A by default, block all calls
#B for the syscalls with the following list of names
#C allow them to proceed

Our System X binary uses `getpid` syscall; let's try to block that. In order to construct a profile with `getpid` excluded, run the following commands in a terminal window. It will store the new profile in `profile.json` (or if you don't have Internet access right now, you can find it in `~/src/examples/who-you-gonna-call/profile.json` in the VM):

```
cd ~/src/examples/who-you-gonna-call/src
curl https://raw.githubusercontent.com/moby/moby/master/profiles/seccomp/default.json \
| grep -v getpid > profile.json
```

I have also prepared a very simple Dockerfile for you to package the System X binary into a container. You can see it by running the following command in the terminal:

```
cat ~/src/examples/who-you-gonna-call/src/Dockerfile
```

You will see the following output. We use the latest Ubuntu base image, and just copy the binary from the host.

```
FROM ubuntu:focal-20200423
COPY ./legacy_server /legacy_server
ENTRYPOINT [ "/legacy_server" ]
```

With that, we can build a Docker image with our legacy software and start it. Do that by running the following commands from the same terminal window. It will build and run a new image called "legacy," use the profile we just created, and expose the port 8080 on the host:

```
cd ~/src/examples/who-you-gonna-call/src
make
docker build -t legacy .
docker run \
--rm \
-ti \
--name legacy \
--security-opt seccomp=./profile.json \             #A
```

```
-p 8080:8080 \                                    #B
legacy
```

#A use the seccomp profile we just created
#B expose the container's port 8080 on the host

You will see the process starting, but notice the PID equal to -1 (bold font). This is the seccomp blocking the `getpid` syscall, and returning an error code -1, just like we asked it to do:

```
Listening on port 8080, PID: -1
```

And voila! We achieved blocking a particular syscall. That's the easy way there for you! Unfortunately, doing it this way gives us less flexibility than strace, we can't pick every other call and we can't attach to a running process. We also need Docker to actually run it, which further limits the use cases where this approach will be suitable.

On the bright side, we achieved blocking the syscall without incurring the harsh penalty introduced by strace. But don't just take my word for it - let's find out how it compares. While the container is running, let's re-run the same ab one-liner we used to establish the steady state in our previous experiments:

```
ab -c1 -t30 http://127.0.0.1:8080/
```

You will see a much more pleasant output, similar to the following. Note, that at 36k requests (bold font), we are at least 10 times faster than we were tracing the `close` syscall (when we achieved 3042 requests per second):

```
(...)
Time taken for tests:   30.001 seconds
Complete requests:      36107
Failed requests:        0
Total transferred:      14912191 bytes
HTML transferred:       10507137 bytes
Requests per second:    1203.53 [#/sec] (mean)
Time per request:       0.831 [ms] (mean)
(...)
```

So there you have it: seccomp the easy way, leveraging Docker. But what if the easy way is not flexible enough? Or you can't or don't want to use Docker? If you need more flexibility, let's look at the level below - libseccomp, or seccomp the hard way.

6.5.2 Seccomp the hard way - libseccomp

Libseccomp (https://github.com/seccomp/libseccomp) is a higher-level, platform-independent library for managing seccomp in Linux kernel that abstracts away the low-level syscalls and exposes easy-to-use functions for the developers. It is leveraged by Docker to implement its seccomp profiles. The best place to start to learn how to use it is the tests (https://github.com/seccomp/libseccomp/tree/master/tests) and man pages, such as `seccomp_init`(3), `seccomp_rule_add`(3) and `seccomp_load`(3). In this section, I would like to show you a short example of how you too can leverage libseccomp with just a few lines of C.

First, we'll need to install the dependencies from package `libseccomp-dev` on Ubuntu/Debian or `libseccomp-devel` on RHEL/Centos. On Ubuntu, you can do that by running the following command (this step is already done for you, if you're using the VM this book comes with):

```
sudo apt-get install libseccomp-dev
```

This will allow us to include the `<seccomp.h>` header in our programs to link against the seccomp library (we'll do both in a second). Let me show you how you can leverage libseccomp to limit the syscalls your program can make. I prepared a small example, which does a minimal amount of setup to change its permissions during the execution time to only allow a small number of syscalls to go through. To see the example, run the following command from a terminal window:

```
cat ~/src/examples/who-you-gonna-call/seccomp.c
```

You will see a very simplistic C program. It uses four functions from libseccomp to limit the syscalls we're allowed to make:

- `seccomp_init` - initialize the seccomp state and prepare it for usage; returns a context
- `seccomp_rule_add` - adds a new filtering rule to a context
- `seccomp_load` - loads the actual context into the kernel
- `seccomp_release` - releases the filter context and frees memory, when you're done with the context

You will see the following output (the four functions in bold font). We start by initializing the context to block all syscalls and then explicitly allow two of them: `write` and `exit`. Then we load the context, execute one `getpid` syscall and one `write`, and release the context:

```
#include <stdio.h>
#include <unistd.h>
#include <seccomp.h>
#include <errno.h>

int main(void)
{
    scmp_filter_ctx ctx;
    int rc; // note, that we totally avoid any error handling here...

    // disable everything by default, by returning EPERM (not allowed)
    ctx = seccomp_init(SCMP_ACT_ERRNO(EPERM));                                    #A
    // allow write...
    rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(write), 0);               #B
    // and exit - otherwise it would segfault on exit
    rc = seccomp_rule_add(ctx, SCMP_ACT_ALLOW, SCMP_SYS(exit), 0);
    // load the profile
    rc = seccomp_load(ctx);                                                       #C

    // write should succeed, but the pid will not
    fprintf(stdout, "getpid() == %d\n", getpid());                                #D

    // release the seccomp context
```

```
    seccomp_release(ctx);                                                    #E
}
```

#A initialize the context by defaulting to returning the EPERM error
#B allow write
#C allow exit
#D load the context we just configured into kernel
#E release the context

Let's compile and start it by running the following commands in the same terminal window:

```
cd ~/src/examples/who-you-gonna-call
cc seccomp.c \
-lseccomp \                                  #A
-o seccomp-example                           #B
./seccomp-example
```

#A we need to include the seccomp library, using the -l flag
#B call the output executable "seccomp-example"

You will see the following output. The fact that we see the output at all proves that `write` syscall was allowed. The program also finished without crashing, meaning that `exit` worked too. But as you can see, the result of `getpid` was -1 (bold font), just like we wanted:

```
getpid() == -1
```

And that's the hard way for you, which -- thanks to libseccomp -- is not that hard after all all. You can now leverage this mechanism to block or allow syscalls as you see fit and you can use it to implement chaos experiments. If you'd like to dig deeper into seccomp, I suggest checking the following resources:

- https://lwn.net/Articles/656307/
- http://man7.org/conf/lpc2015/limiting_kernel_attack_surface_with_seccomp-LPC_2015-Kerrisk.pdf
- https://www.paul-moore.com/docs/devconf-syscall_filtering-pmoore-012014-r1.pdf

And with that, it's time to wrap it up!

6.6 Summary

- System calls (syscalls) are a way of communication between user-land programs and the operating system, allowing them to indirectly access the system resources
- Chaos engineering can produce value even for simple systems consisting of a single process, by testing their resilience to errors when making syscalls
- Strace is a flexible and easy to use tool that allows for detecting and manipulating of syscalls made by any program on the host, but it incurs a non-negligible overhead
- BPF, made easier to use by projects like BCC, allows for much-lower-overhead insight into the running system, including listing syscalls made by processes
- Seccomp can be leveraged to implement chaos experiments designed to block processes from making syscalls and libseccomp makes it much easier to use seccomp

7

Injecting failure into the JVM

This chapter covers

- **Designing chaos experiments for applications written in Java**
- **Injecting failure into a JVM using the `java.lang.instrument` interface ("javaagent")**
- **Using free, open source tools to implement chaos experiments**

Java is one of the most popular programming languages on planet Earth -- in fact, it is consistently placed in the top two or three of many popularity rankings.[1] When practicing chaos engineering, you are very likely to work with systems written in Java. In this chapter, I'm going to focus on preparing you for that moment.

You'll start by looking at an existing Java application to come up with ideas for chaos experiments. Then you'll leverage a unique feature of the Java Virtual Machine (JVM) to inject failure into an existing code base (without modifying the source code) to implement our experiments. Finally, you'll cover some existing tools that will allow you to make the whole process easier, as well as some further reading.

By the end of this chapter, you will have learned how to apply chaos engineering practices to any Java program you run into and understand the underlying mechanisms that make it possible to rewrite Java code on the fly. First stop: a scenario to put things in context.

7.1 Scenario

Your previous chapter's success in rendering the legacy System X less scary and more maintainable hasn't gone unnoticed. In fact, it's been a subject of many watercooler chats on every floor of the office and a source of many approving nods from strangers in the elevator. One interesting side-effect is that people started reaching out, asking for your help to make

[1] take for example https://octoverse.github.com/#top-languages or https://www.tiobe.com/tiobe-index/, two popular rankings

their projects more resilient to failure. Charming at first, it quickly turned into a "please pick a number and wait in the waiting room until your number appears on the screen" situation. Inevitably, a priority queue had to be introduced for the most important projects to be handled quickly.

One of these high-profile projects was called FBEE. At this stage, no one knew for sure what the acronym stood for, but everyone understood it was an enterprise-grade software solution, very expensive, and perhaps a tad over-engineered. Helping make FBEE more resilient felt like the right thing to do, so you accepted the challenge. Let's see what's what.

7.1.1 FizzBuzzEnterpriseEdition

With a little bit of digging, you find out that FBEE actually stands for *FizzBuzzEnterpriseEdition*, and it certainly lives up to its name. It started as a simple programming game used to interview developer candidates and it evolved over time. The game itself is very simple, and goes like this: for each number between 1 and 100, do the following:

- if the number is divisible by 3, print "Fizz"
- if the number if divisible by 5, print "Buzz"
- if the number is divisible by both 3 and 5, print "FizzBuzz"
- otherwise, print the number itself

Over time, however, some people felt that this simple algorithm wasn't enough to test out the enterprise-level programming skills, and decided to provide a reference implementation that was *really* solid. Hence, FizzBuzzEnterpriseEdition in its current form started to exist! Let's have a closer look at the application and how it works

7.1.1.1 Looking around FizzBuzzEnterpriseEdition

If you're following along with the VM provided with this book, a Java JDK (openjdk) is preinstalled and the FizzBuzzEnterpriseEdition source code, as well as jar files, ready to use (otherwise refer to appendix A for installation instructions). In the VM, open a terminal window, and type the following command to go to the directory that contains the application:

```
cd ~/src/examples/jvm
```

In that directory, you'll see `FizzBuzzEnterpriseEdition/lib` subfolder that contains a bunch of jar files that together make the program. You can see the jar files by running the following command from the same directory:

```
ls -al ./FizzBuzzEnterpriseEdition/lib/
```

You will see the following output. Note the main jar file, called `FizzBuzzEnterpriseEdition.jar`, which contains the FizzBuzzEnterpriseEdition main function (bold font), as well as some dependencies:

```
-rw-r--r-- 1 chaos chaos   4467 Jun  2 08:01 aopalliance-1.0.jar
-rw-r--r-- 1 chaos chaos  62050 Jun  2 08:01 commons-logging-1.1.3.jar
-rw-r--r-- 1 chaos chaos  76724 Jun  2 08:01 FizzBuzzEnterpriseEdition.jar
-rw-r--r-- 1 chaos chaos 338500 Jun  2 08:01 spring-aop-3.2.13.RELEASE.jar
-rw-r--r-- 1 chaos chaos 614483 Jun  2 08:01 spring-beans-3.2.13.RELEASE.jar
```

```
-rw-r--r-- 1 chaos chaos 868187 Jun  2 08:01 spring-context-3.2.13.RELEASE.jar
-rw-r--r-- 1 chaos chaos 885410 Jun  2 08:01 spring-core-3.2.13.RELEASE.jar
-rw-r--r-- 1 chaos chaos 196545 Jun  2 08:01 spring-expression-3.2.13.RELEASE.jar
```

If you're curious how it works, you can browse through the source code, but that's not necessary for our needs. In fact, in the practice of chaos engineering you're most likely to be working with someone else's code, and because it's often not feasible to get intimate with the entire codebase due to its size, it would be more realistic if you didn't look into that quite yet. The main function of the application is in `com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.Main`. With that information, we can now go ahead and start the application. Run the following command in a terminal window, still from the same directory:

```
java \
-classpath "./FizzBuzzEnterpriseEdition/lib/*" \                              #A
com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.Main       #B
```

#A allow java to find the jar files of the application, by passing the directory with * wildcard
#B specify the path of the main function

After a short moment, you will see the following output (abbreviated). Apart from the expected lines with numbers and words "fizz" and "buzz", you'll also notice a few very verbose log messages (they're safe to ignore):

```
(...)
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
(...)
```

That's great news, because it looks like `FizzBuzzEnterpriseEdition` is working as expected! It appears to correctly solve the problem at hand, and it would surely convey the message that we're doing serious business here to any new hires, killing two birds with one stone.

But the fact that it works in one use case doesn't tell you anything about how resilient the application is to failure, which is the very reason why you accepted to look at this to begin with. You guessed it -- chaos engineering to the rescue! Let's take a look at how to design an experiment that exposes this piece of software to failure to test how well it handles it.

7.2 Chaos engineering and Java

In order to design a meaningful chaos experiment, you need to start by making an educated guess about what kind of failure might affect our application. Fortunately, over the course of the previous chapters, you've built a little arsenal of tools and techniques that can help with that. For example, you could treat this program as a black box, and apply the techniques you covered in an earlier chapter to see what syscalls it's making, and then design some experiments around blocking some of these syscalls.

You could also leverage the tools from the BCC project you saw earlier (https://github.com/iovisor/bcc), like `javacalls`, to gain an insight into which methods are being called and devise an experiment around the most prominent ones. Or you could package the application in a Docker container and leverage what you learned in chapter 5. The point is, that for the most part, the things you learned before will be applicable to a Java application as well.

But there is more, because Java and the JVM offer some unique and interesting features that you can leverage for the practice of chaos engineering. I'll focus on those in this chapter. So instead of using one of the techniques you've learned before, let's approach it differently. Let's modify an existing method on the fly to throw an exception so that you can verify your assumptions about what happens to the system as whole. Let me show you what I mean by that.

7.2.1 Experiment 1 - idea

The technique I want to teach you in this chapter boils down to these three steps:

1. Identify the class and method that might throw an exception in a real-world scenario
2. Design an experiment that modifies that method on the fly to actually throw the exception in question
3. Verify the application behaves the way you expect it to behave (handles the exception) in the presence of the exception

Steps 2 and 3 both depend on where you decide to inject the exception, so you're going to need to address that first. Let's find a good spot for the exception in the FizzBuzzEnterpriseEdition code now.

7.2.1.1 Finding the right exception to throw

Finding the right place to inject failure requires building understanding of how (a subset) of the application works, and it's one of the things that make chaos engineering both exciting (you get to learn about a lot of different software) and challenging (you get to learn about a lot of different software) at the same time.

It is possible to automate some of this discovery (see the section 7.4 Further reading), but the reality is that you will need to (quickly) build understanding of how things work. You learned some techniques that can help with that in the previous chapters (for example looking under the hood by observing syscalls, or the BCC tools that can give you visibility into methods being called) and the right tool for the job will depend on the application itself,

its complexity level and the sheer amount of code it's built from. One very simple, yet useful technique is to search for the exceptions thrown.

As a reminder, in Java, every method needs to declare any exceptions that its code might throw, through the use of the `throws` keyword. For example, a made-up method that might throw an `IOException` could look like the following:

```
public static void mightThrow(String someArgument) throws IOException {
  // definition here
}
```

That means that you can find all of the places in the source code where an exception might be thrown by simply searching for that keyword. From inside the VM, run the following commands in a terminal window to do just that:

```
cd
       ~/src/examples/jvm/src/src/main/java/com/seriouscompany/business/java/fizzbuzz/packa
       genamingpackage/            #A
grep \
       -n \                        #B
       -r \                        #C
       ") throws" .
```

#A navigate to the folder to avoid dealing with super-long paths in the output
#B print the line numbers
#B recursively search in subfolders

You will see the following output, listing three locations with the "throws" keyword (in bold font). The last one is an interface so let's ignore that one for now. Let's focus on the first two locations:

```
./impl/strategies/SystemOutFizzBuzzOutputStrategy.java:21:   public void output(final
       String output) throws IOException {

./impl/ApplicationContextHolder.java:41:    public void setApplicationContext(final
       ApplicationContext applicationContext) throws BeansException {

./interfaces/strategies/FizzBuzzOutputStrategy.java:14:      public void output(String
       output) throws IOException;
```

Let's take a look at the first file from that list, the `SystemOutFizzBuzzOutputStrategy.java`, by running the following command in a terminal window:

```
cat
       ~/src/examples/jvm/src/src/main/java/com/seriouscompany/business/java/fizzbuzz/packa
       genamingpackage/impl/strategies/SystemOutFizzBuzzOutputStrategy.java
```

You will see the following output (abbreviated), with a single method called output, capable of throwing `IOException`. The method is very simple, printing to and flushing the standard output. This is the class and method that's used internally when you ran the application and saw all of the output in the console:

```
(...)
public class SystemOutFizzBuzzOutputStrategy implements FizzBuzzOutputStrategy {
```

```
(...)
       @Override
       public void output(final String output) throws IOException {
         System.out.write(output.getBytes());
         System.out.flush();
       }
}
```

This looks like a good starting point for an educational experiment:

- It's reasonably uncomplicated
- It's used when you simply run the program
- It has the potential to crash the program, if the error handling is not done properly.

It's a decent candidate, so let's decide to use it as a target for the experiment. You can go ahead and design the experiment. Let's do just that.

7.2.2 Experiment 1 - plan

Without looking at the rest of the source code, you can design a chaos experiment that injects an `IOException` into the `output` method of the `SystemOutFizzBuzzOutputStrategy` class, to verify that the application as a whole can withstand that. If the error-handling logic is on point, it wouldn't be unreasonable to expect it to retry the failed write and at the very least to log an error message and signal a failed run. You can leverage the return code to know whether the application finished successfully.

Putting this all together into our usual four-step template, this is the plan of the experiment:

1. Observability: the return code and the standard output of the application
2. Steady state: the application runs successfully and prints the correct output
3. Hypothesis: if an `IOException` exception is thrown in the `output` method of the `SystemOutFizzBuzzOutputStrategy` class, the application returns an error code after its run
4. Run the experiment!

The plan sounds straightforward, but in order to implement it, you need to know how to modify a method on the fly. This is made possible by a feature of the JVM often referred to as `javaagent`, that allows us to write a class that can rewrite the bytecode of any other Java class that is being loaded into the JVM. Bytecode? Don't worry, you'll cover that in a moment.

This is an advanced topic that might be new to even a seasoned Java developer. It is of particular interest in the practice of chaos engineering - it allows you to inject failure into someone else's code to implement various chaos experiments. It's also easy to mess things up, because this technique gives you access to pretty much all and any code executed in the JVM, including built-in classes. It is therefore very important to make sure that you understand what you're doing, and I'm going to take my time to guide you through it.

So in order to be able to implement this experiment, I want to give you all the tools you need:

1. A quick refresher of what bytecode is, and how to peek into it, before you start modifying it.
2. An easy way to see the bytecode generated from Java code.
3. An overview of `java.lang.instrument` interface, and how to use it to implement a class that can modify other classes.
4. A walkthrough of how to implement our experiment with no external dependencies.
5. Finally, once you understand how modifying code on the fly works under the hood, some higher-level tools that can do some of the work for you.

Let's start at the beginning - by acquanting you to the bytecode.

7.2.3 Brief introduction to JVM bytecode

One of the key design goals of Java was to make it portable -- write once, run anywhere (WORA) principle. To that end, Java applications run inside of a Java Virtual Machine (JVM). When you run an application, it's first compiled from the source code (.java) into Java bytecode (.class), that can then be executed by any compatible implementation of the JVM, on any platform that supports it. The bytecode is independent of the underlying hardware. This process is summed up in figure 7.1.

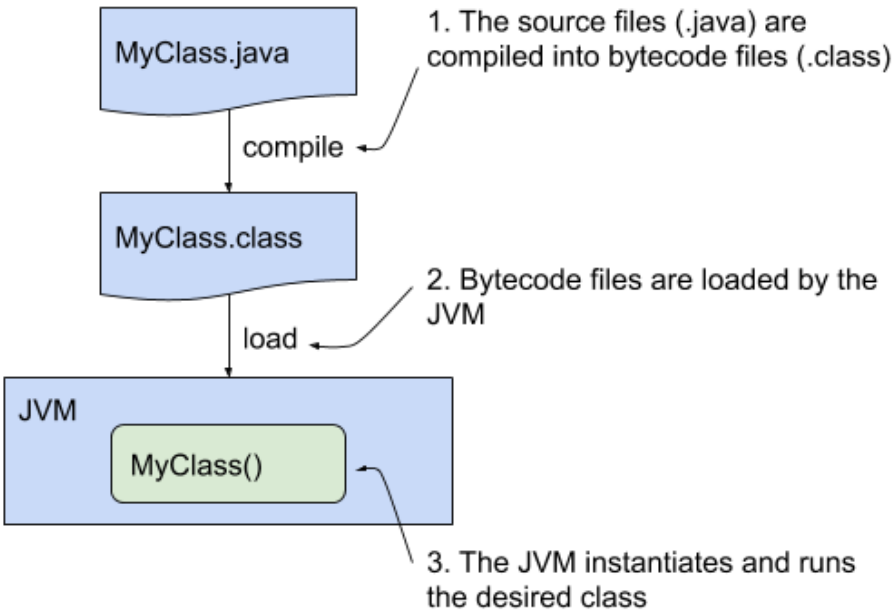


Figure 7.1 High-level overview of running Java code

What does a JVM look like? You can see the formal specs for all Java versions for free at https://docs.oracle.com/javase/specs/ - and they are pretty good. Take a look at the Java 8

JVM specification (that's the version you're running in the VM that is shipped with this book) at https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf. It describes the format of a .class file, the instruction set of the VM (similar to an instruction set of a physical processor), as well as the structure of the JVM itself. It's good to know that you can always look things up in the formal specification. But nothing teaches better than doing things ourselves, so let's get our hands dirty and look at what this process is like in practice. You want to go modifying other people's bytecode, so before you do that, let's first peek into what the bytecode actually looks like.

7.2.3.1 READING THE BYTECODE

OK, so you want to modify someone else's code on the fly to inject failure for our chaos experiment. If you're serious about it (and want to be responsible), you'll need to get familiar with what bytecode actually looks like. Let's go through the whole process of compiling, running and looking into the bytecode of a simple class. To make things easy to start, I prepared a little sample application that you can work on. Let's start by opening a terminal window in our VM, and going to the location of the example, by running the following command:

```
cd ~/src/examples/jvm/
```

From within that directory, you will find a subfolder structure (`./org/my`) with an example program (`Example1.java`). The directory structure is important, as this needs to match the package name, so let's stick to the same folder for the rest of this chapter. You can see the contents of the example program by running the command:

```
cat ./org/my/Example1.java
```

You will see the following "hello world" program, a class called `Example1`. Note that it contains a main method that does a single call to `println` (both in bold font) to print a simple message to the standard output:

```
package org.my;

class Example1
{
    public static void main(String[] args)
    {
        System.out.println("Hello chaos!");
    }
}
```

Before you can run it, it needs to be compiled into bytecode. You can do that using the `javac` command line tool. In our very simple example, you just need to specify the file path. Compile it by running the following command:

```
javac ./org/my/Example1.java
```

No output means that there were no errors.

> **TIP** If you'd like to learn more about what the compiler did there, run the same command with `-verbose` flag added to it. Where did the bytecode file go? It will be sitting next to the source file, with the file name corresponding to the name of the class itself.

Let's take a look at that subfolder again, by running the following command:

```
ls -l ./org/my/
```

You will see an output just like the following; note the new file, `Example1.class`, the result of you compiling the java file (bold font):

```
(...)
-rw-r--r-- 1 chaos chaos 422 Jun  4 08:44 Example1.class
-rw-r--r-- 1 chaos chaos 128 Jun  3 10:43 Example1.java
(...)
```

To run it, you can use the *java* command, and just specify the fully-qualified class name (with the package prefix). Do that by running the following command (remember, you still need to be in the same directory):

```
java org.my.Example1
```

You will see an output of the "hello world" program, just like the following:

```
Hello chaos!
```

It runs, which is nice, but I bet this is all old news to you. Even if you are not very familiar with Java before, the steps you took look pretty much like any other compiled language. What you might have not seen before is the bytecode it produces. Fortunately for us, JDK ships with another tool, `javap`, which allows us to print the bytecode contents of the class in a human-readable form. To do it to our `org.my.Example1` class, run the following command:

```
javap -c org.my.Example1
```

You will see an output, just like the following (abbreviated to just show the main method), describing what JVM machine instructions were generated for our Example1 class. You will see four different instructions:

```
Compiled from "Example1.java"
class org.my.Example1 {
(...)

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2    // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3    // String Hello chaos!
       5: invokevirtual #4    // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

Let's take a look at a single instruction to understand it's format. For example this one:

```
3: ldc           #3    // String Hello chaos!
```

The format is the following:

- relative address
- colon
- name of the instruction (you can look them up in the JVM spec document)
- argument
- comment about what the argument actually is (human-readable format)

Translating the instructions making up the `main` method into English, you have a `getstatic` instruction, that gets a static field `out`[2] of type `java.io.PrintStream`[3] from class `java.lang.System`[4], then an `ldc` instruction that loads a constant string "Hello chaos!" and pushes it onto what's called *operand stack*. This is followed by the `invokevirtual` instruction, which invokes instance method `.println` and pops the value previously pushed to the operand stack. Finally, `return` instruction ends the function call. And voila! That's what is written in the `Example1.java` file, as far as the JVM is concerned.

This might feel a bit dry. Why is it important from the perspective of chaos engineering? Because this is what you're going to be modifying to inject failure in our chaos experiments.

You can look up all the details about all these instructions from the docs I mentioned earlier (https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf) but that's not necessary right now. As a practitioner of chaos engineering, I just want you to know that you can easily access the bytecode, see it in a human-readable(-ish) form, and look up any definitions you might want to understand more in detail.

There are plenty of other interesting things about the JVM, but for this chapter, I just need to make you feel comfortable with some basic bytecode. This sneak peek of the JVM bytecode gives us just enough information you need to understand the next step - instrumenting the bytecode on the fly. Let's take a look at that now.

7.2.3.2 Using -javaagent to instrument the JVM

OK, so you're on our quest to implement the chaos experiment you've designed, and to do that you need to know how to modify the code on the fly. You can do that by leveraging a mechanism directly provided by the JVM.

This is going to get a little bit technical, so let me just say this: you will cover higher-level tools that make it easier (see section 7.3), but it's important to learn what the JVM actually offers, in order to understand the limitations of this approach. Skipping straight to the higher-level stuff would be a little bit like driving a car without understanding how the gearbox works. It might be fine for most people, but it won't cut it for a race driver. When doing chaos engineering, I need you to be a race driver.

With that preamble out of the way, let's dive in and take a look at what JVM has to offer. Java comes with instrumentation and code transformation capabilities built-in, by the means of the `java.lang.instrument` package that has been available since JDK version 1.5 (https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html). People often refer to it as `javaagent`, because that's the name of the command-line

[2] https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out
[3] https://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html
[4] https://docs.oracle.com/javase/8/docs/api/java/lang/System.html

argument that you use to attach the instrumentation. The package defines two interfaces, both of which are needed for you to inject failure into a class:

1. `ClassFileTransformer` - classes implementing this interface can be registered to transform class files of a JVM; it requires a single method called `transform`
2. `Instrumentation` - allows for registering instances implementing the `ClassFileTransformer` interface with the JVM to receive classes for modification before they're used

Together, it makes it possible to inject code into the class, just like you need for the experiment. It allows you to register a class (implementing `ClassFileTransformer`) that will receive the byte code of all other classes before they are used, and will be able to transform them. This is summarized in figure 7.2.

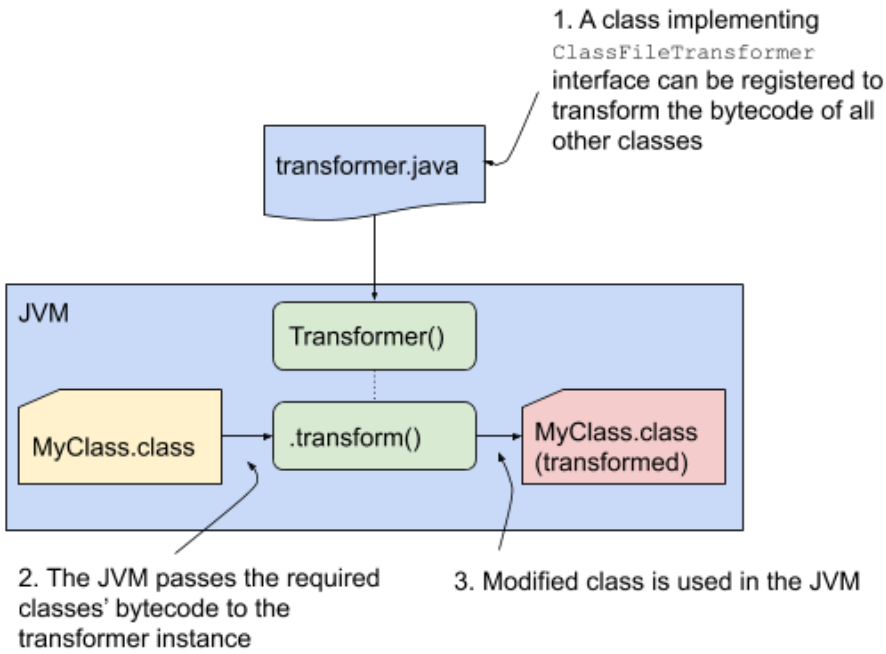


Figure 7.2 Instrumenting JVM with the java.lang.instrument package

Now, I know that this is a lot of new information, so I suggest to do it in two steps:

1. First, let's go through setting everything up with the javaagent, but hold off from modifying any code
2. Second, add the actual code to modify the bytecode of the classes you're interested in separately

In order to implement the first part, you just need to follow the steps that the architects of the `java.lang.instrument` package came up with. To make your life easier, let me summarize it for you. It all boils down to these four steps:

1. Write a class implementing the `ClassFileTransformer` interface - let's call it `ClassPrinter`
2. Implement another class with the special method called `premain`, that will register an instance of `ClassPrinter`, so that the JVM knows to use it - let's call it `Agent`
3. Package the `Agent` and `ClassPrinter` classes into a jar file, with an extra attribute `Premain-Class` pointing to the class with the premain method (`Agent`)
4. Run Java with an extra argument `-javaagent:/path/to/agent.jar` pointing to the jar file created in the previous step

Let's do that! I've prepared for you the three files that you're going to need. First, the `ClassPrinter` class, that you can see by running the following command in a terminal window:

```
cat ~/src/examples/jvm/org/agent/ClassPrinter.java
```

You will see the contents of a class with a single method, `transform`, that is needed to satisfy the `ClassFileTransformer` interface (both in bold font). You'll notice that the method has a bunch of arguments that are required by the interface. In the use case of our chaos experiment, you're only going to need two of them (both in bold font below):

- `className` (the name of the class to transform)
- `classfileBuffer` (the actual binary content of the class file)

For now, like I suggested earlier, let's skip the modification part and instead just print the name and size for each class that the JVM will call it with, and return the class file buffer unchanged. This will effectively list all of the classes loaded by the JVM, in the order that they are loaded, showing us that the javaagent mechanism worked:

```
package org.agent;
import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;
class ClassPrinter implements ClassFileTransformer {
    public byte[] transform(ClassLoader loader,
                            String className,                          #A
                            Class<?> classBeingRedefined,
                            ProtectionDomain protectionDomain,
                            byte[] classfileBuffer)                    #B
        throws IllegalClassFormatException {
    System.out.println("Found class: " + className                     #C
       + " (" + classfileBuffer.length + " bytes)");
    return classfileBuffer;                                            #D
  }
}
```

#A the name of the class brought by the JVM for transformation
#B the binary content of the class file for the class
#C just print the name of the class and its binary size

#D return the class unchanged

Now, you need to actually register that class so that the JVM uses it for instrumentation. This is straightforward too, and I prepared a sample class that does that for you. You can see it by running the following command in a terminal window:

```
cat ~/src/examples/jvm/org/agent/Agent.java
```

You will see the following Java class. Note, that it imports the `Instrumentation` package, and implements the special `premain` method (in bold font), which will be called by the JVM before the main method is executed. It uses the `addTransformer` method to register an instance of `ClassPrinter` class (also in bold font). This is how you actually make the JVM take an instance of our class and allow it to modify the bytecode of all other classes:

```
package org.agent;

import java.lang.instrument.Instrumentation;

class Agent {
  public static void premain(String args,                          #A
                      Instrumentation instrumentation){     #B
    ClassPrinter transformer = new ClassPrinter();
    instrumentation.addTransformer(transformer);                   #C
  }
}
```

#A the premain method needs to have this special signature
#B an object implementing the Instrumentation interface will be passed by the JVM when the method is called
#C use addTransformer method to register an instance of our ClassPrinter class

And finally, the piece de resistance, is a special attribute `Premain-Class` that needs to be set when packaging these two classes into a jar file. The value of the attribute needs to point to the name of the class with the premain method (`org.agent.Agent` in our case) so that the JVM knows which class to call. The easiest way to do that is to create a manifest file. I prepared one for you. To see it, run the following command in a terminal window:

```
cat ~/src/examples/jvm/org/agent/manifest.mf
```

You will see the following output. Note the `Premain-Class` attribute, specifying the fully-qualified class name of our `Agent` class, the one with the `premain` method. Once again, this is how you tell the JVM to use this particular class to attach the instrumentation.

```
Manifest-Version: 1.0
Premain-Class: org.agent.Agent
```

And that's all the ingredients you need. The last step is to package it all together in a format that's required by the JVM as the `-javaagent` argument - a simple jar file with all the necessary classes and the special attribute you just covered. Let's now compile the two classes and build our jar file into agent1.jar, by running the following commands:

```
cd ~/src/examples/jvm
javac org/agent/Agent.java
javac org/agent/ClassPrinter.java
```

```
jar vcmf org/agent/manifest.mf agent1.jar org/agent
```

Once that's ready, you're all done. You can go ahead and leverage the `-javaagent` argument of the `java` command, to use our new instrumentation. Do that by running the following command in a terminal window:

```
cd ~/src/examples/jvm
java \
       -javaagent:./agent1.jar \                                    #A
       org.my.Example1                                              #B
```

#A use the -javaagent argument to specify the path to our instrumentation jar file
#B run the Example1 class you had looked at before

You will see the following output (abbreviated), with our instrumentation listing all the classes passed to it. There are a bunch of built-in classes, and then the name of our target class, `org/my/Example1` (bold font). Eventually you can see the familiar "hello chaos" output of the main method of that target class (also bold font):

```
(...)
Found class: sun/launcher/LauncherHelper (14761 bytes)
Found class: java/util/concurrent/ConcurrentHashMap$ForwardingNode (1618 bytes)
Found class: org/my/Example1 (429 bytes)
Found class: sun/launcher/LauncherHelper$FXHelper (3224 bytes)
Found class: java/lang/Class$MethodArray (3642 bytes)
Found class: java/lang/Void (454 bytes)
Hello chaos!
(...)
```

So it worked, very nice! You have just instrumented your JVM, and didn't even break a sweat in the process. You're getting really close to being able to implement our chaos experiment now, and I'm sure you can't wait to finish the job. Let's go and do it!

7.2.4 Experiment 1 - implementation

You are one step away from being able to implement our chaos experiment - you know how to attach our instrumentation to a JVM and get all the classes with their bytecode passed to us. Now you just need to figure out how to modify the bytecode to include the failure you need for the experiment. You want to inject code automatically into the class you're targeting, to simulate it throwing an exception. As a reminder, this is the class:

```
(...)
public class SystemOutFizzBuzzOutputStrategy implements FizzBuzzOutputStrategy {
(...)
       @Override
       public void output(final String output) throws IOException {
         System.out.write(output.getBytes());
         System.out.flush();
       }
}
```

For our experiment, it doesn't really matter where the exception is thrown in the body of this method, so you can as well add it at the beginning. But how do you know what bytecode

instructions to add? Well, a simple way to figure that out is to copy some existing bytecode. Let's take a look at how to do that now.

7.2.4.1 WHAT INSTRUCTIONS SHOULD YOU INJECT?

Because the javaagent mechanism operates on bytecode, you need to know what bytecode instructions you want to inject. Fortunately, you now know how to look under the hood of a .class file, and you can leverage that to write the code you want to inject in Java, and then see what bytecode it produces. To do that, I prepared a very simple class throwing an exception. Run the following command inside of a terminal window in your VM to see it:

```
cat ~/src/examples/jvm/org/my/Example2.java
```

You will see the following code. It has two methods: a static `throwIOException` that does nothing but throw an `IOException`, and a `main` that calls that same `throwIOException` method (both in bold font):

```
package org.my;
import java.io.IOException;
class Example2
{
    public static void main(String[] args) throws IOException
    {
        Example2.throwIOException();
    }

    public static void throwIOException() throws IOException
    {
        throw new IOException("Oops");
    }
}
```

The reason I added this extra method is to make things easier for us - calling a static method with no arguments is really simple in the bytecode. But don't take my word for it. Let's check that for ourselves, by compiling the class and printing its bytecode. You can do that by running the following commands in the same terminal:

```
cd ~/src/examples/jvm/
javac org/my/Example2.java
javap -c org.my.Example2
```

You will see the following bytecode (abbreviated to only show the main method). Notice it's a single `invokestatic` JVM instruction, specifying the method to call, as well as no arguments and no return value (which is represented by `()V` in the comment). This is good news, because you're only going to need to add a single instruction injected into our target method:

```
(...)
  public static void main(java.lang.String[]) throws java.io.IOException;
    Code:
       0: invokestatic  #2                  // Method throwIOException:()V
       3: return
(...)
```

So to make our target method `SystemOutFizzBuzzOutputStrategy.output` throw an exception, you can add a single `invokestatic` instruction to the beginning of it, pointing to any static method throwing the exception you want, and you're done! Let's finally take a look at how to put all of this together.

7.2.4.2 Injecting code into JVM on the fly

You know what instructions you want to inject, where to inject them and how to use the instrumentation to achieve that. The last question now is how to actually modify that bytecode that the JVM will pass to our class. You could go back to the JVM specs, open the chapter on the class file format and implement code to parse and modify the instructions. Fortunately, there is no need to reinvent the wheel. The following are a few frameworks and libraries that you can use to do that:

- https://asm.ow2.io/
- https://www.javassist.org/
- https://bytebuddy.net/
- https://commons.apache.org/proper/commons-bcel/
- https://github.com/cglib/cglib

In the spirit of simplicity, I'll show you how to rewrite a method using the ASM library (https://asm.ow2.io/), but you could probably pick any one of these frameworks. The point here is not to teach you how to become an expert at modifying Java classes. It's to give you just enough understanding of how that process works so that you can design meaningful chaos experiments. In real life, in your experiments you're probably going to use one of the higher-level tools that you'll cover in the "existing tools" section a little later in the chapter -- but it is important to understand how to implement a complete example from scratch. Do you remember the race driver and gearbox analogy? When doing chaos engineering, you need to know the limitations of your methods, and it's harder to do when using tools that do things you don't understand. Let's dig in.

Groovy and Kotlin

If you were ever wondering how Groovy (http://www.groovy-lang.org/) and Kotlin (https://kotlinlang.org/) languages were implemented to run in the JVM - the answer is that they are using ASM to generate the bytecode. And so do the higher-level libraries like Byte Buddy (https://bytebuddy.net/)

Remember how earlier I suggested splitting the implementation into two steps, the first being the `org.agent` package you used for printing classes passed to our instrumentation by the JVM? Let's take the second step now, and build on top of that to add the bytecode rewriting part.

I prepared another package, `org.agent2`, that implements the modification that you want to make using ASM. Note, that ASM already shipped with `openjdk`, so there is no need to install it. ASM is a large library with good documentation, but for our purposes you will use

a very small subset of what it can do. To see it, run the following command from the terminal inside of the VM:

```
cd ~/src/examples/jvm/
cat org/agent2/ClassInjector.java
```

You will see the following class, `org.agent2.ClassInjector`. It is Java after all, so it's a little bit verbose. It implements the same `transform` method that is needed to be registered for instrumenting the bytecode of classes inside of the JVM, just like we saw before. It also implements another method, a static `throwIOException`, that prints a message to stderr and throws an exception. The `transform` method looks for the (very long) name of the class, and only does any rewriting if the class name matches. It uses an ASM library `ClassReader` instance to read the bytecode of the class into an internal representation as an instance of the `ClassNode` class. That `ClassNode` instance allows us to do the following:

- iterate through the methods
- select the one called output
- inject a single `invokestatic` instruction as the first instruction, calling to our `throwIOException` static method

This is depicted in figure 7.3.

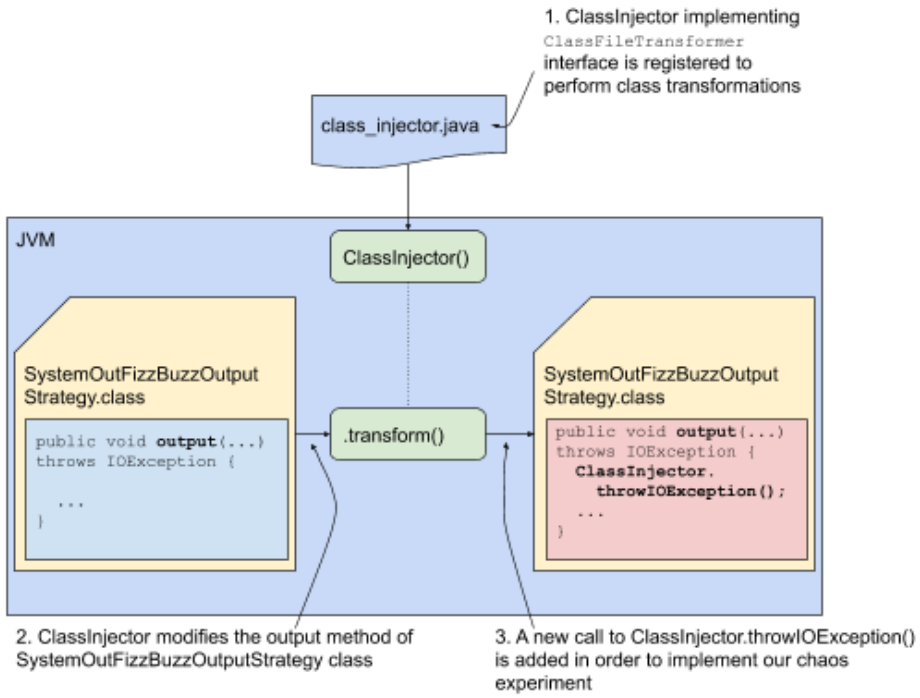


Figure 7.3 Instrumenting the JVM with the java.lang.instrument package

Take a look at the `ClassInjector` class (listing 7.x):

Listing 7.1 ClassInjector.java

```
package org.agent2;

import java.io.IOException;
import java.util.List;
import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.IllegalClassFormatException;
import java.security.ProtectionDomain;

import jdk.internal.org.objectweb.asm.ClassReader;
import jdk.internal.org.objectweb.asm.ClassWriter;
import jdk.internal.org.objectweb.asm.tree.*;
import jdk.internal.org.objectweb.asm.Opcodes;
public class ClassInjector implements ClassFileTransformer {
    public String targetClassName =
       "com/seriouscompany/business/java/fizzbuzz/packagenamingpackage/impl/strategies/Syst
       emOutFizzBuzzOutputStrategy";

    public byte[] transform(ClassLoader loader, String className,
       Class<?> classBeingRedefined, ProtectionDomain protectionDomain,
       byte[] classfileBuffer) throws IllegalClassFormatException {            #A

    if (className.equals(this.targetClassName)){

      ClassNode classNode = new ClassNode();
      new ClassReader(classfileBuffer).accept(classNode, 0);                   #B
      classNode.methods.stream()
        .filter(method -> method.name.equals("output"))                        #C
        .forEach(method -> {
          InsnList instructions = new InsnList();
          instructions.add(new MethodInsnNode(                                 #D
              Opcodes.INVOKESTATIC,
              "org/agent2/ClassInjector",
              "throwIOException",
              "()V",
              false // not a method
          ));
          method.maxStack += 1;                                                #E
          method.instructions.insertBefore(
         method.instructions.getFirst(), instructions);                        #F
        });
        final ClassWriter classWriter = new ClassWriter(0);
        classNode.accept(classWriter);
        return classWriter.toByteArray();                                      #G
    }
    return classfileBuffer;
  }
  public static void throwIOException() throws IOException
  {
      System.err.println("[CHAOS] BOOM! Throwing");
      throw new IOException("CHAOS");
  }
}
```

#A the same transform method needed to implement the ClassFileTransformer interface

#B ClassReader reads and parses the bytecode into an internal representation of type ClassNode
#C filter only the method called 'output'
#D create a new instruction of type invokestatic, calling a static method throwIOException on the org/agent2/ClassInjector class with no arguments and no return value
#E in order to allow an extra instruction on the stack, you need to increase its size
#F insert the instructions at the beginning of the method
#G generate the resulting bytecode using a ClassWriter class

Once again, to satisfy the requirements of the format accepted by the `javaagent` argument in order for the JVM to use this class as instrumentation, you're going to need:

- a class with a method called premain, that creates and registers an instance of `ClassInjector` class
- a manifest including the special attribute `Premain-Class`, pointing to the class with premain method
- a jar file packaging it all together, so that we can pass in the `javaagent` argument

I wrote a simple premain class `org.agent2.Agent` for you, that you can see by running the following command from the same folder:

```
cat org/agent2/Agent.java
```

You will see the following class, implementing the premain method and using the same `addTransformer` method you used earlier to register an instance of `ClassInjector` class with the JVM. Once again, this is how you tell the JVM to pass all the classes being loaded to `ClassInjector` for modifications:

```
package org.agent2;
import java.lang.instrument.Instrumentation;
class Agent {
  public static void premain(String args, Instrumentation instrumentation){
    ClassInjector transformer = new ClassInjector();
    instrumentation.addTransformer(transformer);
  }
}
```

And I also prepared a manifest, very similar to the previous one, so that you can build the jar the way it's required by the javaagent argument. You can see it by running the following command from the same directory:

```
cat org/agent2/manifest.mf
```

The output you will see is the following. The only difference to the previous manifest, is that it points to the new agent class (bold font):

```
Manifest-Version: 1.0
Premain-Class: org.agent2.Agent
```

The last part of the puzzle is that in order to have access to the internal.jdk packages, you need to add the `-XDignore.symbol.file` flag when compiling our classes. With that, you're ready to prepare a new agent jar, and let's call it agent2.jar. Create it by running the following commands, still from the same directory:

```
javac -XDignore.symbol.file org/agent2/Agent.java
javac -XDignore.symbol.file org/agent2/ClassInjector.java
jar vcmf org/agent2/manifest.mf agent2.jar org/agent2
```

The resulting agent2.jar file will be created in the current directory and can be used to implement our experiment. Ready? Let's run it.

7.2.4.3 RUN THE EXPERIMENT

Finally, you have everything set up to run your experiment and see what happens. As a reminder, this is our experiment plan:

1. Observability: the return code and the standard output of the application
2. Steady state: the application runs successfully and prints the correct output
3. Hypothesis: if an `IOException` exception is thrown in the `output` method of the `SystemOutFizzBuzzOutputStrategy` class, the application returns an error code after its run
4. Run the experiment!

First, let's establish the steady state, by running the application unmodified and inspecting the output and the return code. You can do that by running the following command in a terminal window:

```
java \
-classpath "./FizzBuzzEnterpriseEdition/lib/*" \                                #A
com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.Main \      #B
2> /dev/null                                                                     #C
```

#A allow java to find the jar files of the application, by passing the directory with * wildcard
#B specify the path of the main function
#C remove the noisy logging messages

After a few seconds, you will see the following output (abbreviated). The output is correct:

```
1
2
Fizz
4
Buzz
(...)
```

Let's verify the return code, by running the following command in the same terminal window:

```
echo $?
```

The output will be `0`, indicating a successful run. So the steady state is satisfied: you've got the correct output and a successful run. Let's now run the experiment! To run the same application, but this time using your instrumentation, run the following command:

```
java \
-javaagent:./agent2.jar                                                          #A
-classpath "./FizzBuzzEnterpriseEdition/lib/*" \
com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.Main \
2> /dev/null
```

#A add the java agent instrumentation jar you've just built

This time, there will be no output, which is understandable, because you modified the function doing the printing to always throw an exception. Let's verify the other assumption of our hypothesis - namely, that the application handles it well by indicating an error as a return value. To check the return code, rerun the same command in the same terminal:

```
echo $?
```

The output is still `0`, failing our experiment and showing a problem with the application. Turns out that the hypothesis about `FizzBuzzEnterpriseEdition` was wrong. Despite not printing anything, it doesn't indicate an error as its return code. Houston, we've got a problem!

This has been a lot of learning, so I'd like you to appreciate what you just did:

- you took an existing application you weren't familiar with
- you found a place that throws an exception and designed a chaos experiment to test whether an exception thrown in that place is handled by the application in a reasonable way
- you prepared and applied JVM instrumentation, with no magical tools and external dependencies
- you prepared and applied automatic bytecode modifications, with no external dependencies other than the ASM library already provided by the openjdk
- you ran the experiment, modified the code on the fly, and demonstrated scientifically that the application was not handling the failure well

But once again, it's OK to be wrong. Experiments like this are supposed to help you find problems with software, like you just did. And it would make for a pretty boring chapter, if you did all that work, and it turned out to be working just fine, wouldn't it?

The important thing here is that you added another tool in your toolbox and demystified another technology stack. Hopefully it will come in handy sooner rather than later.

Now that you understand how the underlying mechanisms work, you're allowed to cheat a little bit - to take shortcuts. Let's take a look at some useful tools you can leverage to avoid doing so much typing in your next experiments to achieve the same effect.

7.3 Existing tools

Although it's important to understand how the JVM `java.lang.instrument` package works in order to design meaningful chaos experiments, there is no need to reinvent the wheel every time. In this section, I'd like to show you a few free, open source tools that you can use to make your life easier. Let's start with Byteman.

7.3.1 Byteman

Byteman (https://byteman.jboss.org/) is a versatile tool that allows for modifying the bytecode of JVM classes on the fly (using the same instrumentation you covered in this chapter) to trace, monitor, and overall mess around with the behavior of your Java code.

Its differentiating factor is the fact that it comes with a simple DSL (Domain Specific Language) that's very expressive and allows you to describe how you'd modify the source code of the Java class, mostly forgetting about the actual bytecode structure (you can afford to do that, because you already know how it works under the hood). Let's have a look at how to use it, starting by installing it.

7.3.1.1 Installing Byteman

You can get the binary releases, source code and documentation for all versions of Byteman at https://byteman.jboss.org/downloads.html. At the time of writing, the latest version is 4.0.11. Inside of your VM, that version is downloaded and unzipped to `~/src/examples/jvm/byteman-download-4.0.11`. If you'd like to download it on a different host, you can do that by running the following command in a terminal:

```
wget https://downloads.jboss.org/byteman/4.0.11/byteman-download-4.0.11-bin.zip
unzip byteman-download-4.0.11-bin.zip
```

This will create a new folder called `byteman-download-4.0.11` which contains byteman and its docs. You're going to need the byteman.jar file, which can be found in the lib subfolder. To see it, run the following command in the same terminal.

```
ls -l byteman-download-4.0.11/lib/
```

You will see three jar files, and you're interested in the byteman.jar (bold font), that you can use as a `-javaagent` argument:

```
-rw-rw-r-- 1 chaos chaos  10772 Feb 24 15:32 byteman-install.jar
-rw-rw-r-- 1 chaos chaos 848044 Feb 24 15:31 byteman.jar
-rw-rw-r-- 1 chaos chaos  15540 Feb 24 15:29 byteman-submit.jar
```

That's it. You're good to go. Let's use it.

7.3.1.2 Using Byteman

To illustrate how much easier it is to use Byteman, let's reimplement the same modification you did for the chaos experiment from section 7.2.4.2. In order to do that, you need to follow three steps:

1. prepare a Byteman script that throws an exception in the targeted method (let's call it `throw.btm`)
2. run Java using the `byteman.jar` as the `-javaagent` argument
3. point `byteman.jar` to use our `throw.btm` script

Let's start with the first point. A byteman script is a flat text file, with any number of rules, each of which follows this format (the programmer's guide is available at https://downloads.jboss.org/byteman/4.0.11/byteman-programmers-guide.html#the-byteman-rule-language):

```
  # rule skeleton
  RULE <rule name>
  CLASS <class name>
  METHOD <method name>
  BIND <bindings>
```

```
  IF  <condition>
  DO  <actions>
  ENDRULE
```

I prepared a script that does exactly what the chaos experiment you implemented earlier does. You can see it by running the following command in a terminal window:

```
cd ~/src/examples/jvm/
cat throw.btm
```

You will see the following rule. It does exactly what you did before: it changes the method `output` in class `SystemOutFizzBuzzOutputStrategy`, to throw a `java.io.IOException` exception at the entry into the method:

```
RULE throw an exception at output
CLASS SystemOutFizzBuzzOutputStrategy                                    #A
METHOD output                                                            #B
AT ENTRY                                                                 #C
IF true                                                                  #D
DO
  throw new java.io.IOException("BOOM");                                 #E
ENDRULE
```

#A modify the class SystemOutFizzBuzzOutputStrategy
#B modify the method output
#C at the entry into the method
#D always execute (it's possible to add here conditions for the rule to trigger)
#E throw a new exception

With that in place, let's handle the points 2 and 3. When using the `-javaagent` parameter with Java, it is possible to pass extra arguments after the "=" sign. With byteman, the only parameter supported is `script=<location of the script to execute>`. Therefore, to run the same `FizzBuzzEnterpriseEdition` class you did before, but have byteman execute our script (bold font), all you need to do is run the following command:

```
cd ~/src/examples/jvm/
java \
  -javaagent:./byteman-download-4.0.11/lib/byteman.jar=script:throw.btm \      #A
  -classpath "./FizzBuzzEnterpriseEdition/lib/*" \
  com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.Main \
  2>/dev/null                                                                  #B
```

#A use byteman jar file as a javaagent, and specify our script after the "=" sign
#B discard the stderr to avoid looking at the logging noise

You will see no output at all, just like in the experiment you ran before. You achieved the same result without writing or compiling any Java code or dealing with any bytecode.

Compared to writing your own instrumentation, using Byteman is simple, and the DSL makes it easy to quickly write rules, without having to worry about bytecode instructions at all. It also offers other advanced features, like attaching to a running JVM, triggering rules based on complex conditions, adding code at various points in methods and much more.

It's definitely worth knowing about Byteaman, but there are some other interesting alternatives. One of them is Byte-monkey - let's take a closer look.

7.3.2 Byte-monkey

Although not as versatile as Byteman, I think that Byte-monkey (https://github.com/mrwilson/byte-monkey) deserves a mention here. It also works by leveraging the `-javaagent` option of the JVM and uses ASM library to modify the bytecode. The unique proposition of Byte-monkey is that it only offers actions useful for chaos engineering. Namely, there are four modes you can use (verbatim from the README):

```
Fault: Throw exceptions from methods that declare those exceptions
Latency: Introduce latency on method-calls
Nullify: Replace the first non-primitive argument to the method with null
Short-circuit: Throw corresponding exceptions at the very beginning of try blocks
```

I'll show you how to use Byte-monkey to achieve the same effect you did for the chaos experiment. But first, let's install it.

7.3.2.1 Installing Byte-monkey

You can get the binary releases, and the source code of Byte-monkey from https://github.com/mrwilson/byte-monkey/releases. At the time of writing, the only version available is 1.0.0. Inside of your VM, that version is downloaded to `~/src/examples/jvm/byte-monkey.jar`. If you'd like to download it on a different host, you can do that by running the following command in a terminal:

```
wget https://github.com/mrwilson/byte-monkey/releases/download/1.0.0/byte-monkey.jar
```

That single file, `byte-monkey.jar` is all you need. Let's see how to use it.

7.3.2.2 Using Byte-monkey

Now, for the fun part. Let's reimplement the experiment once again, but this time with a small twist! Byte-monkey makes it easy to only throw the exceptions at a particular rate, so to make things more interesting, let's modify the method to only throw an exception 50% of the time. This can be achieved by passing the rate argument, when specifying the -javaagent jar for the JVM.

Run the following command, to use the `byte-monkey.jar` file as our javaagent, use the fault mode, rate of 0.5, and filter to only our fully-qualified (and very long) name of the class and the method (all in bold font):

```
java \
-javaagent:byte-
       monkey.jar=mode:fault,rate:0.5,filter:com/seriouscompany/business/java/fizzbuzz/pack
       agenamingpackage/impl/strategies/SystemOutFizzBuzzOutputStrategy/output \
                                                                  #A
-classpath "./FizzBuzzEnterpriseEdition/lib/*" \
com.seriouscompany.business.java.fizzbuzz.packagenamingpackage.impl.Main \
2>/dev/null
```

#A use the fault mode (throwing exceptions), at a rate of 50%, and filter once again to only affect the very long name of the class and method you're targeting.

You will see an output similar to the following, with about 50% of the lines printed, and the other 50% skipped:

```
(...)
1314FizzBuzz1619
Buzz
22Fizz29Buzz

FizzBuzzFizz

38Buzz41Fizz43
FizzBuzz
4749
(...)
```

And voila! Another day, another tool in your awesome toolbox. Give it a star on Github (https://github.com/mrwilson/byte-monkey) they deserve it! When you're back, let's take a look at Chaos Monkey for Spring Boot.

7.3.3 Chaos Monkey for Spring Boot

The final mention in this section goes to *Chaos Monkey for Spring Boot* (https://github.com/codecentric/chaos-monkey-spring-boot). I won't get too much into the details here, but if your application uses Spring Boot, you might be interested in it. The documentation (for the latest version 2.2.0 it's at https://codecentric.github.io/chaos-monkey-spring-boot/2.2.0/) is pretty good, and gives you a decent overview of how to get started.

In my opinion, the differentiating feature here is that it understands Spring Boot, and offers failure (called assaults) on the high-level abstractions. It can also expose an API, which allows you to add, remove and reconfigure these assaults on the fly through HTTP or JMX. Currently supported are:

- Latency assault - inject latency to a request
- Exception assault - throw exceptions at runtime
- AppKiller assault - shutdown the app on a call to a particular method
- Memory assault - use up memory

If you're using Spring Boot, I recommend that you take a good look at this framework. That's the third and final tool I wanted to show you. Let's take a look at some further reading.

7.4 Further reading

If you'd like to learn more about chaos engineering and JVM, I'd like to recommend a few pieces of further reading. First, two papers from the KTH Royal Institute of Technology in Stockholm. You can find them both, along with the source code at https://github.com/KTH/royal-chaos:

- ChaosMachine (https://arxiv.org/pdf/1805.05246.pdf): analyses exception-handling hypotheses of three popular pieces of software written in Java (TTorrent, Broadleaf and XWiki) and produces actionable reports for the developers automatically. It leverages the same `-javaagent` mechanism we learnt about here.
- TripleAgent (https://arxiv.org/pdf/1812.10706.pdf): a system that automatically monitors, injects failure and improves resilience of existing software running in JVM.

The paper evaluates BitTorrent and HedWig projects to demonstrate the feasibility of automatic resilience improvements.

Second, from the University of Lille & INRIA in Lille, the paper *Exception Handling Analysis and Transformation Using Fault Injection: Study of Resilience Against Unanticipated Exceptions* (https://hal.inria.fr/hal-01062969/document) analyses nine open source projects and shows that 39% of catch blocks executed during test suite execution can be made more resilient.

Finally, I wanted to mention, that when we covered the `java.lang.instrument` package (https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html) I only spoke about instrumenting the classes when starting a JVM. It is also possible to attach to a running JVM, and instrument classes which had already been loaded. Doing so involves implementing the `agentmain` method and you can find all the details in the mentioned documentation page.

7.5 Summary

- JVM allows you to instrument and modify code on the fly through the use `java.lang.instrument` package (part of the JDK)
- In Java programs, exception handling is often a weak spot, and it's a good starting point for chaos engineering experiments, even on a source code base you're not very familiar with
- Open source tools like Byteman, Chaos Monkey for Spring Boot and Byte-monkey make it easier to inject failure for your chaos experiments and they run on top of the same `java.lang.instrument` package to achieve that

8

Application-level fault-injection

This chapter covers

- **Building chaos engineering capabilities directly into your application**
- **Making sure that the extra code doesn't affect the application's performance**
- **More advanced usage of ApacheBench**

So far, you've covered a variety of ways of applying chaos engineering to a selection of different systems. The languages, tools, and approaches varied, but they all had one thing in common - working with source code outside your control. If you're in a role like Site Reliability Engineer (SRE) or platform engineer, that's going to be your bread and butter. But sometimes, you will have the luxury of applying chaos engineering to your own code. This chapter focuses on how baking chaos engineering options directly into your application can be a quick, easy and - dare I say it - fun way of increasing your confidence in the overall stability of the system as a whole. I'll guide you through designing and running two experiments: one injecting latency into functions responsible for communicating with an external cache and another injecting intermittent failure through the simple means of raising an exception. The example code is written in Python, but don't worry if it's not your forte: I promise to keep it basic.

If you like the sound of it, let's go for it. First things first: a scenario.

8.1 Scenario

Let's say that you work for an e-commerce company and you're designing a system for recommending new products to your customers, based on their previous few queries. As a practitioner of chaos engineering, you're excited: this might be a perfect opportunity to add features allowing you to inject failure directly into the code base.

In order to generate recommendations, you need to be able to track the queries your customers make, even if they are not logged in. The e-commerce store is a website, so you

decide to simply use a cookie (https://en.wikipedia.org/wiki/HTTP_cookie) to store a session ID for each new user. This allows you to distinguish between the requests and attribute each search query to a particular session.

In your line of work, latency is very important; if the website doesn't feel quick and responsive to the customers, they will buy from your competitors. The latency therefore influences some of the implementation choices, as well as becomes one of the targets for chaos experiments. In order to minimize the latency added by your system, you decide to use an in-memory key-value store, Redis (https://redis.io/), as your session cache and store only the last three queries the user made. These previous queries are then fed to the recommendation engine every time the user searches for a product, and come back with some potentially interesting products to display in a "you might be interested in" box.

So here's how it all works together. When a customer visits your e-commerce website, the system checks whether a session ID is already stored in a cookie in the browser. If it's not, a random session ID is generated and stored. As the customer searches through the website, the last three queries are saved in the session cache, and are used to generate a list of recommended products that is then presented to the user in the search results. For example, after the first search query of "apple," the system might recommend "apple juice." After the second query for "laptop," given that the two consecutive queries were "apple" and "laptop," the system might recommend a "macbook pro." If you've worked in e-commerce before, you know this is a form of cross-selling (https://en.wikipedia.org/wiki/Cross-selling) and it's a serious and powerful technique used by most of the online stores and beyond. This process is summarized in figure 8.1.

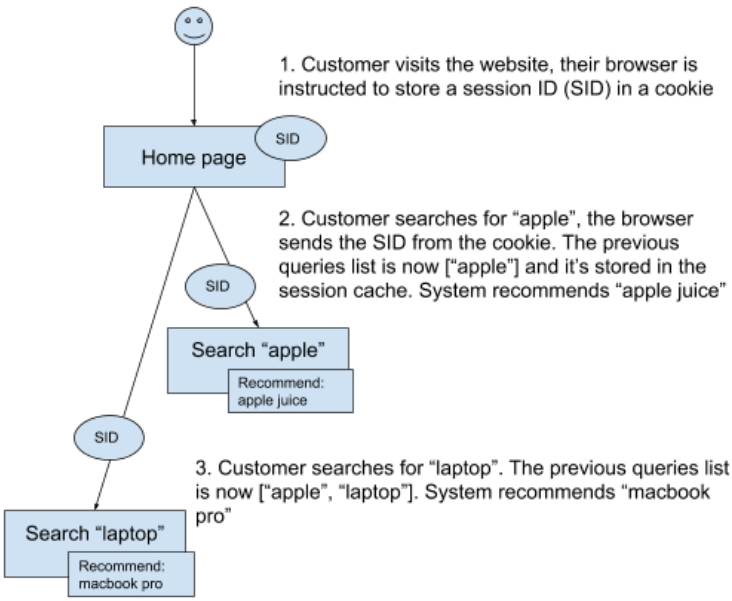


Figure 8.1. High-level overview of the session tracking system

Learning how to implement this system is not the point of this chapter. What I'm aiming at here is to show you a concrete, realistic example of how you can add minimal code directly into the application to make running chaos experiments on it easy. To do that, let me first walk you through a very simple implementation of this system, for now without any chaos engineering changes, and then, once you're comfortable with it, I'll walk you through the process of building two chaos experiments into it.

8.1.1 Implementation details - before chaos

I'm providing you with a bare-bones implementation of the relevant parts of this website, written in Python and using Flask HTTP framework (https://flask.palletsprojects.com/). If you don't know Flask, don't worry, we'll walk through the implementation to make sure everything is clear.

Inside of your VM, the source code can be found in `~/src/examples/app` (for installation instructions outside of the VM, please refer to appendix A). The code doesn't implement any chaos experiments quite yet - we'll add that together. The main file, `app.py`, provides a single HTTP server, exposing three endpoints:

- the index page (at `/`) that displays the search form and sets the session ID cookie
- the search page (at `/search`) that stores the queries in the session cache and displays the recommendations
- the reset page (at `/reset`) that replaces the session ID cookie with a new one to make testing easier for you (this endpoint is for your convenience only)

Let's start with the index page route, the first one any customer will see. It's implemented in the `index` function, and does exactly two things: returns some static HTML to render the search form, and sets a new session ID cookie, through the `set_session_id` function. The latter is made easy through Flask's built-in method of accessing cookies (`flask.request.cookies.get`) as well as setting new ones (`response.set_cookie`). After visiting this endpoint, the browser stores the random UUID value in the `sessionID` cookie, and it sends that value with every subsequent request to the same host. That's how the system is able to attribute the further actions to a session ID. If you're not familiar with Flask, the `@app.route("/")` decorator tells Flask to serve the decorated function (in this case `index`) under the `/` endpoint.

Next, the search page is where the magic happens. It's implemented in the `search` function, decorated with `@app.route("/search", methods=["POST", "GET"])`, meaning that both GET and POST requests to `/search` will be routed to it. It reads the session ID from the cookie, the query sent from the search form on the home page (if any), and stores the query for that session using the `store_interests` function. `store_interests` reads the previous queries from Redis, appends the new one, stores it back, and returns the new list of interests. Using that new list of interests, it calls the `recommend_other_products` function, that - for simplicity - returns a hardcoded list of products. This process is summarized in figure 8.2.

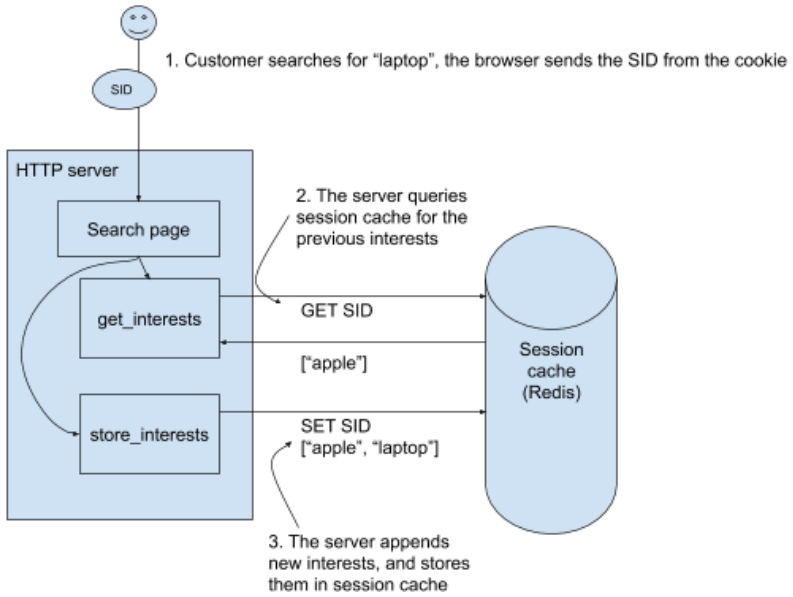


Figure 8.2. Search page and session cache interactions

When that's done, the `search` function renders an HTML page presenting the search results as well as the recommended items. Finally, the third endpoint, implemented in the `reset` function replaces the session ID cookie with a new, random one and redirects the user to the home page.

Listing 8.1 provides the full source code for this application. For now, ignore the commented out section on chaos experiments.

Listing 8.1 app.py

```
import uuid, json, redis, flask

COOKIE_NAME = "sessionID"

def get_session_id():
    """ Read session id from cookies, if present """
    return flask.request.cookies.get(COOKIE_NAME)

def set_session_id(response, override=False):
    """ Store session id in a cookie """
    session_id = get_session_id()
    if not session_id or override:
        session_id = uuid.uuid4()
    response.set_cookie(COOKIE_NAME, str(session_id))


CACHE_CLIENT = redis.Redis(host="localhost", port=6379, db=0)
```

```
# Chaos experiment 1 - uncomment this to add latency to Redis access
#import chaos
#CACHE_CLIENT = chaos.attach_chaos_if_enabled(CACHE_CLIENT)

# Chaos experiment 2 - uncomment this to raise an exception every other call
#import chaos2
#@chaos2.raise_rediserror_every_other_time_if_enabled
def get_interests(session):
    """ Retrieve interests stored in the cache for the session id """
    return json.loads(CACHE_CLIENT.get(session) or "[]")

def store_interests(session, query):
    """ Store last three queries in the cache backend """
    stored = get_interests(session)
    if query and query not in stored:
        stored.append(query)
    stored = stored[-3:]
    CACHE_CLIENT.set(session, json.dumps(stored))
    return stored


def recommend_other_products(query, interests):
    """ Return a list of recommended products for a user, based on interests """
    if interests:
        return {"this amazing product": "https://youtube.com/watch?v=dQw4w9WgXcQ"}
    return {}


app = flask.Flask(__name__)

@app.route("/")
def index():
    """ Handle the home page, search form """
    resp = flask.make_response("""
    <html><body>
        <form action="/search" method="POST">
            <p><h3>What would you like to buy today?</h3></p>
            <p><input type='text' name='query'/>
            <input type='submit' value='Search'/></p>
        </form>
        <p><a href="/search">Recommendations</a>. <a href="/reset">Reset</a>. </p>
    </body></html>
    """)
    set_session_id(resp)
    return resp

@app.route("/search", methods=["POST", "GET"])
def search():
    """ Handle search, suggest other products """
    session_id = get_session_id()
    query = flask.request.form.get("query")
    try:
        new_interests = store_interests(session_id, query)
    except redis.exceptions.RedisError as exc:
        print("LOG: redis error %s", str(exc))
        new_interests = None
    recommendations = recommend_other_products(query, new_interests)
    return flask.make_response(flask.render_template_string("""
```

```
    <html><body>
        {% if query %}<h3>I didn't find anything for "{{ query }}"</h3>{% endif %}
        <p>Since you're interested in {{ new_interests }}, why don't you try...
        {% for k, v in recommendations.items() %} <a href="{{ v }}">{{ k }}</a>{% endfor
       %}!</p>
        <p>Session ID: {{ session_id }}. <a href="/">Go back.</a></p>
    </body></html>
    """,
        session_id=session_id,
        query=query,
        new_interests=new_interests,
        recommendations=recommendations,
    ))

@app.route("/reset")
def reset():
    """ Reset the session ID cookie """
    resp = flask.make_response(flask.redirect("/"))
    set_session_id(resp, override=True)
    return resp
```

Let's now see how to start the application. It has two external dependencies:

- Flask (https://flask.palletsprojects.com/)
- redis-py (https://github.com/andymccurdy/redis-py)

You can install both in the versions that were tested with this book by running the following command in your terminal window:

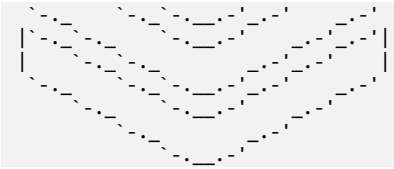
```
pip install redis==3.5.3 Flask==1.1.2
```

You're also going to need an actual instance of Redis running on the same host, listening for new connections on the default port 6379. If you're using the VM, it's preinstalled (consult appendix A for installation instructions if you're not using the VM). Open another terminal window, and start a Redis server by running the following command:

```
redis-server
```

You will see the characteristic output of Redis, similar to the following:

```
54608:C 28 Jun 2020 18:32:12.616 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
54608:C 28 Jun 2020 18:32:12.616 # Redis version=6.0.5, bits=64, commit=00000000,
       modified=0, pid=54608, just started
54608:C 28 Jun 2020 18:32:12.616 # Warning: no config file specified, using the default
       config. In order to specify a config file use ./redis-server /path/to/redis.conf
54608:M 28 Jun 2020 18:32:12.618 * Increased maximum number of open files to 10032 (it was
       originally set to 8192).
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 6.0.5 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 54608
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           http://redis.io
```



With that, you are ready to start the application! While Redis is running in the second terminal window, go back to the first one and run the following command, still from `~/src/examples/app`. Note, that it will start the application in development mode, with detailed error stacktraces and automatic reload on changes to the source code:

```
cd ~/src/examples/app                                    #A
FLASK_ENV=development \                                  #B
FLASK_APP=app.py \                                       #C
       python3 -m flask run                              #D
```

#A go to the location with the source code of the application
#B specify development environment for easier debugging and auto-reload
#C specify FLASK_APP environment variable, which points Flask to run the application
#D run the flask module, specifying run command to start a web server

The application will start, and you'll see an output just like the following, specifying the app it's running, the host and port where the application is accessible, and the environment (all in bold font):

```
 * Serving Flask app "app.py" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 289-495-131
```

You can now browse to http://127.0.0.1:5000/ to confirm it's working. You will see a simple search form, asking you to type the name of the product you're interested in. Try searching for "apple." You are taken to a second page, where you will be able to see your previous queries as well as the recommendations. Be absolutely sure to click the recommendations, they are great! If you repeat it a few times, you will notice that the page retains the last three search queries. Finally, note that the page also prints the session ID, and if you're curious, you can see it in the cookies section in your browser.

OK, so now you have a simple, yet functional application that we'll pretend you wrote. Time to have some fun with it! Let's do some chaos engineering.

8.2 Experiment 1 - Redis latency

In the e-commerce store scenario I described at the beginning of the chapter, the overall latency of the website is paramount: you know that if you slow the system down too much, customers will start leaving the website and buying from your competitors. It's therefore important that you understand how the latency communicating with the session cache

(Redis) affects the overall speed of the website. And that's where chaos engineering shines - we can go, simulate some latency, and measure how much it affects the system as a whole.

You have injected latency before in different ways. In chapter 4, you used Traffic Control (tc) to add latency to a database and in chapter 5 you leveraged Docker and Pumba to do the same. So how is this different this time? In the previous scenarios, we made a lot of effort to modify the behavior of the system without modifying the source code. This time I want to add to that by showing you how easy it is to add chaos engineering when you *are* in control of the application's design. Everyone can do that - you just need to have a little bit of imagination! Let's design a simple experiment around the latency.

8.2.1 Experiment 1 plan

In the example application, it's easy to establish that for each request the session cache is accessed twice: first to read the previous queries, and second to store the new set. We can therefore hypothesize that we will see a double of any latency added to the Redis calls in the overall latency figure for the website. Let's find out whether that's true. By now, you're well versed in using Apache Bench (ab) for generating traffic and observing latencies, so let's leverage that once again. Here's one possible version of a chaos experiment that will help us test that theory:

1. Observability: generate traffic and observe the latency using ab
2. Steady state: observe latency without any chaos changes
3. Hypothesis: if we add a 100ms latency to each interaction with the session cache (reads and writes), the overall latency of the `/search` page should increase by 200ms
4. Run the experiment!

That's it! Now, all we need to do is follow this plan, starting with the steady state.

8.2.2 Experiment 1 steady state

So far we've used ab to generate GET requests. This time, we have a good opportunity to learn how to use it to send POST requests, like the ones sent from the search form on the index page that the browser sends to the `/search` page. In order to do that, we need to do the following things:

1. Use the POST method, instead of GET
2. Use the `Content-type` header to specify the value used by the browser when sending an HTML form (`application/x-www-form-urlencoded`)
3. Pass the actual form data as the body of the request to simulate the value from a form
4. Pass the session ID (we can make it up) in a cookie in another header - just like the browser does with every request

Fortunately, this all can be done with `ab`, by using the following arguments:

- `-H "Header: value"` to set custom headers, one for the cookie with session ID and one for the content type. This flag can be used multiple times to set multiple headers.
- `-p post-file` to send the contents of the specified file as the body of the request. It also automatically assumes the POST method. That file needs to follow the HTML form

format, but don't worry if you don't know it. In this simple use case, I'll show you a body we can use: `query=TEST` to query for "TEST". The actual query in this case doesn't matter

Putting this all together, and using our typical concurrency of 1 (`-c 1`), run time of 10 seconds (`-t 10`), we end up with the following command. Assuming that the server is still running, open another terminal window and run the following:

```
echo "query=Apples" > query.txt                                    #A
ab -c 1 -t 10 \
       -H "Cookie: sessionID=something" \                         #B
       -H "Content-type: application/x-www-form-urlencoded" \     #C
       -p query.txt \                                             #D
       http://127.0.0.1:5000/search
```

#A create a simple file with the query content
#B send a header with the cookie specifying the sessionID
#C send a header specifying the content type to a simple HTML form
#D use the previously created file with the simple query in it

You will see the familiar output of ab, similar to the following (abbreviated). My VM managed to do 1673 requests, or about 167 requests per second (5.98 ms per request) with no errors (all four in bold font):

```
Server Software:        Werkzeug/1.0.1
Server Hostname:        127.0.0.1
Server Port:            5000
(...)
Complete requests:      1673
Failed requests:        0
(...)
Requests per second:    167.27 [#/sec] (mean)
Time per request:       5.978 [ms] (mean)
```

So far so good. These numbers represent your steady state, the baseline. Let's implement some actual chaos and see how these change.

8.2.3 Experiment 1 implementation

Time to implement the core of your experiment. This is the cool part - because you own the code, there are a million and one ways of implementing the chaos experiment, and you're free to pick whichever works best for you! I'm going to guide you through just one example of what that could look like, focusing on three things:

1. Keep it simple
2. Make the chaos experiment parts optional for your application and disabled by default
3. Be mindful of the performance impact the extra code has on the whole application

These are good guidelines for any chaos experiments, but like I said before, you will pick the right implementation based on the actual application you're working on. For the example application, it relies on a Redis client accessible through the `CACHE_CLIENT` variable, and then the two functions using it, `get_interests` and `store_interest` use the `get` and `set` methods on that cache client, respectively (all in bold font):

```
CACHE_CLIENT = redis.Redis(host="localhost", port=6379, db=0)                    #A

def get_interests(session):
    """ Retrieve interests stored in the cache for the session id """
    return json.loads(CACHE_CLIENT.get(session) or "[]")                         #B

def store_interests(session, query):
    """ Store last three queries in the cache backend """
    stored = get_interests(session)
    if query and query not in stored:
        stored.append(query)
    stored = stored[-3:]
    CACHE_CLIENT.set(session, json.dumps(stored))                                #C
    return stored
```

#A an instance of Redis client is created and accessible through CACHE_CLIENT variable
#B get_interests is using the get method of CACHE_CLIENT
#C stores_interests is using the set method of CACHE_CLIENT (and get by transition, through the call to get_interests)

That means that all you need to do to implement the experiment is to modify the `CACHE_CLIENT` to inject the latency into both of the get and set methods. There are plenty of ways of doing that, but the one I'm going to suggest is to write a simple wrapper class. The wrapper class would have the two required methods (get and set) and rely on the wrapped class for the actual logic. Before actually calling the wrapped class, it would sleep for the desired time. And then, based on an environment variable, you'd need to optionally replace the `CACHE_CLIENT` with an instance of the wrapper class. Still with me? I prepared a simple wrapper class for you (`ChaosClient`), along with a function to attach it (`attach_chaos_if_enabled`) in another file called `chaos.py`, in the same folder (`~/src/examples/app`). `attach_chaos_if_enabled` is written in a way so as to inject the experiment only if an environment variable called `CHAOS` is set. That's to satisfy the "disabled by default" expectation. The amount of time to inject is controlled by another environment variable called `CHAOS_DELAY_SECONDS` and defaults to 750ms. Listing 8.2 is an example implementation.

Listing 8.2 chaos.py

```
import time
import os


class ChaosClient:
    def __init__(self, client, delay):
        self.client = client                                                     #A
        self.delay = delay
    def get(self, *args, **kwargs):                                              #B
        time.sleep(self.delay)                                                   #C
        return self.client.get(*args, **kwargs)
    def set(self, *args, **kwargs):                                              #D
        time.sleep(self.delay)
        return self.client.set(*args, **kwargs)

def attach_chaos_if_enabled(cache_client):
    """ creates a wrapper class that delays calls to get and set methods """
    if os.environ.get("CHAOS"):                                                  #E
```

```
        return ChaosClient(cache_client, float(os.environ.get("CHAOS_DELAY_SECONDS",
       0.75)))
    return cache_client
```

#A the wrapper class stores a reference to the original cache client
#B the wrapper class provides the get method, that's expected on the cache client, that wraps the client's method of the same name
#C before the method relays to the original get method, it waits for a certain amount of time
#D the wrapper class provides also the set method, exactly like the get method
#E return the wrapper class only if CHAOS environment variable is set

Now, equipped with this, you can modify the application (`app.py`) to make use of this new functionality. You can import it and use it to conditionally replace the `CACHE_CLIENT`, provided that the right environment is set. To do that, all you need to do is find the line where you instantiate the cache client inside of the `app.py` file:

```
CACHE_CLIENT = redis.Redis(host="localhost", port=6379, db=0)
```

Add two lines after it, importing and calling the `attach_chaos_if_enabled` function, passing the CACHE_CLIENT variable as an argument. Together, they will look like the following:

```
CACHE_CLIENT = redis.Redis(host="localhost", port=6379, db=0)
import chaos
CACHE_CLIENT = chaos.attach_chaos_if_enabled(CACHE_CLIENT)
```

With that, the scene is set and ready for the grand finale. Let's run the experiment!

8.2.4 Experiment 1 execution

In order to activate the chaos experiment, we need to restart the application with the new environment variables. You can do that by stopping the previously run instance (press Ctrl-C) and running the following command:

```
CHAOS=true \                                                    #A
CHAOS_DELAY_SECONDS=0.1 \                                       #B
FLASK_ENV=development \                                         #C
FLASK_APP=app.py \                                              #D
python3 -m flask run                                            #E
```

#A activate the conditional chaos experiment code by setting the CHAOS environment variable
#B specify chaos delay injected as 0.1 second, or 100ms
#C specify the flask development env for better error messages
#D specify the same app.py application
#E run flask

Once it's up and running, you're good to go to re-run the same ab command you used to establish the steady state once again. To do that, run the following command in another terminal window:

```
echo "query=Apples" > query.txt && \                            #A
ab -c 1 -t 10 \
       -H "Cookie: sessionID=something" \                       #B
       -H "Content-type: application/x-www-form-urlencoded" \   #C
       -p query.txt \                                           #D
       http://127.0.0.1:5000/search
```

#A create a simple file with the query content
#B send a header with the cookie specifying the sessionID
#C send a header specifying the content type to a simple HTML form
#D use the previously created file with the simple query in it

After the 10-second wait, when the dust settles, you will see the ab output, much like the following. This time, my setup managed to complete only 48 requests (208 ms per request), still without errors (all three in bold font):

```
(...)
Complete requests:      48
Failed requests:        0
(...)
Requests per second:    4.80 [#/sec] (mean)
Time per request:       208.395 [ms] (mean)
(...)
```

That's consistent with the expectations. The initial hypothesis was that adding 100ms to every interaction with the session cache should result in extra 200ms additional latency overall. And as it turns out, for once, our hypothesis was correct! It took a few chapters for that, but that's a bucket list item check off! Now, before we get too narcissistic, let's discuss a few pros and cons of running chaos experiments this way.

8.2.5 Experiment 1 discussion

Adding chaos engineering code directly to the source code of the application is a double-edged sword: it's often easier to do, but it also increases the scope of things that can go wrong. For example, if your code introduces a bug that breaks your program, instead of increasing the confidence in the system you decreased it. Or, if you added latency to the wrong part of the codebase, your experiments might yield results that don't match the reality, giving you false confidence (which is arguably even worse).

You might also think, "Duh, I added code to sleep for X seconds, of course it's slowed down by that amount." And yes, you're right. But now imagine that this application is larger than the few dozen lines we looked at. It might be much harder to be sure about how latencies in different components affect the system as a whole. But if the argument of the human fallibility doesn't convince you, here's a more pragmatic one: it's often quicker to do an experiment and confirm even the simple assumptions, than it is to analyze them and reach meaningful conclusions.

I'm also sure that you noticed that reading and writing to Redis in two separate actions is not going to work with any kind of concurrent access and can lose writes. Instead, it could be implemented using a Redis set and atomic add operation, fixing this problem as well as the double penalty for any network latency. My focus here was to keep it as simple as possible, but thanks for pointing that out!

Finally, there is always the question of performance: if you add extra code to the application, you might make it slower. Fortunately, because you are free to write the code whatever way you please, there are ways around it. In the example you just covered, the extra code is applied only if the corresponding environment variables are set during startup. That means that apart from the extra if statement, there is no overhead when running the application without the chaos experiment. And when it's on, the penalty is the cost of an

extra function call to our wrapper class. Given that we're waiting for times at a scale of milliseconds, that overhead is negligible.

That's what my lawyers advised me to tell you, anyway. With all these caveats out of the way, let's do another experiment, this time injecting failure, rather than slowness.

8.3 Experiment 2 - failing requests

Let's focus on what happens when things fail rather than slow down. Let's take a look at the function `get_interests` again. As a reminder, it looks like the following. (Note, that there is no exception handling whatsoever.) If the `CACHE_CLIENT` throws any exceptions (bold font), they will just bubble up further up the stack:

```
def get_interests(session):
    """ Retrieve interests stored in the cache for the session id """
    return json.loads(CACHE_CLIENT.get(session) or "[]")
```

To test the exception handling of this function, you'd typically write unit tests and aim to cover all legal exceptions that can be thrown. That will cover this bit, but will tell you a little about how the entire application behaves when these exceptions arise. To test the whole application, you'd need to set up some kind of *integration* or *end-to-end* (e2e) tests, where an instance of the application is stood up along with its dependencies, and some client traffic is created. By working on that level, you can verify things from the user's perspective (what error will the user see, as opposed to what kind of exception some underlying function returns), test for regressions and more. It's another step toward reliable software.

And this is where applying chaos engineering can create even more value. You can think of it as the next step in that evolution - a kind of end-to-end testing, while injecting failure into the system to verify the whole reacts the way that you expect. Let me show you what I mean: let's design another experiment to test whether an exception in the `get_interests` function is handled in a reasonable manner.

8.3.1 Experiment 2 plan

What should happen if `get_interests` receives an exception when trying to read from the session store? Depending on the type of page you're serving, it will differ. For example, if you're using that session date to list recommendations in a sidebar to the results of a search query, it might make more economic sense to skip the side bar and allow the user to at least click on other products. If, on the other hand, we are talking about the checkout page, then not being able to access the session data might make it impossible to finish the transaction, so it makes sense to return an error and ask the user to try again.

In our case, we don't even have a buy page, so let's focus on the first type of scenario: if the get_interests function throws an exception, it will bubble up in the store_interests function, which is called from our search website with the following code. Note the except block, which catches `RedisError`, the type of error that might be thrown by our session cache client (in bold font):

```
    try:
        new_interests = store_interests(session_id, query)
```

```
    except redis.exceptions.RedisError:                                    #A
        print("LOG: redis error %s", str(exc))
        new_interests = None
```

#A the type of exception thrown by the Redis client we use is caught and logged here

That error handling should result in the exception in `get_interests` being transparent to the user - they just won't see any recommendations. We can create a simple experiment to test that out:

1. Observability: browse to the application and see the recommended products
2. Steady state: the recommended products are displayed in search results
3. Hypothesis: if we add a `redis.exceptions.RedisError` exception every other time `get_interests` is called, we should see the recommended products every other time we refresh the page
4. Run the experiment!

We already have seen that the recommended products are there, so we can jump directly to the implementation!

8.3.2 Experiment 2 implementation

Similar to the first experiment, there are plenty of different ways of implementing this. And just like in the first experiment, let me suggest a very simple example of how you can do that. Since we're using Python, let's write a simple decorator, that we can apply to the `get_interests` function. Like before, we want to activate this behavior only when the CHAOS environment variable is set.

I prepared another file in the same folder, called `chaos2.py`, that implements a single function `raise_rediserror_every_other_time_if_enabled`, that's designed to be used as a Python decorator (https://wiki.python.org/moin/PythonDecorators). This rather verbosely named function takes another function as a parameter, and implements the desired logic: return the function if the chaos experiment is not active, and return a wrapper function if it is active. The wrapper function tracks the number of times it's called and raises an exception on every other call. On the other calls, it relays to the original function with no modifications. The source code of one possible implementation is in listing 8.3.

Listing 8.3 chaos2.py

```
import os
import redis


def raise_rediserror_every_other_time_if_enabled(func):
    """ Decorator, raises an exception every other call to the wrapped function """
    if not os.environ.get("CHAOS"):
        return func                                                        #A
    counter = 0
    def wrapped(*args, **kwargs):
        nonlocal counter
        counter += 1
        if counter % 2 == 0:
```

```
            raise redis.exceptions.RedisError("CHAOS")                    #B
        return func(*args, **kwargs)                                      #C
    return wrapped
```

#A if the special environment variable CHAOS is not set, return the original function
#B raise an exception on every other call to this method
#C relay the call to the original function

Now we just need to actually use it. Similar to the first experiment, we'll modify the app.py file to add the call to this new function. Find the definition of the get_insterests function, and prepend it with a call to the decorator we just saw. It should look like the following (the decorator is in bold font):

```
import chaos2
@chaos2.raise_rediserror_every_other_time_if_enabled
def get_interests(session):
    """ Retrieve interests stored in the cache for the session id """
    return json.loads(CACHE_CLIENT.get(session) or "[]")
```

Also, make sure that you undid the previous changes, or otherwise you'd be running two experiments at the same time! If you did, then that's all we need to implement for experiment 2. You're ready to roll. Let's run the experiment!

8.3.3 Experiment 2 execution

Let's make sure the application is running. If you still have it running from the previous sections, you can keep it, otherwise, start it by running the following command:

```
CHAOS=true \                                                              #A
FLASK_ENV=development \                                                   #B
FLASK_APP=app.py \                                                        #C
python3 -m flask run                                                      #D
```

#A activate the conditional chaos experiment code by setting the CHAOS environment variable
#B specify the flask development env for better error messages
#C specify the same app.py application
#D run flask

This time the actual experiment execution step is really simple: browse to the application (http://127.0.0.1:5000/) and refresh a few times. You will see the recommendations every other time, and no recommendations the other times, just like we predicted, proving our hypothesis! Also, in the terminal window running the application, you will see logs similar to the following, showing an error on every other call. That's another confirmation that what we did worked:

```
127.0.0.1 - - [07/Jul/2020 22:06:16] "POST /search HTTP/1.0" 200 -
127.0.0.1 - - [07/Jul/2020 22:06:16] "POST /search HTTP/1.0" 200 -
LOG: redis error CHAOS
```

And that's a wrap. Two more experiments under your belly. Pat yourself on the back, and let's take a look at some pros and cons of the approach presented in this chapter.

8.4 Application versus infrastructure

When should you bake the chaos engineering directly into your application, as opposed to doing that on the underlying layers? Like most things like life, that choice is a trade-off.

Doing it directly in your application can be much easier, and has the advantage of using the same tools that you're already familiar with. You can also get creative about how you structure the code for the experiments, and implementing sophisticated scenarios tends to not be a problem.

The flip side to that, is that since you're writing code, all the problems you have writing any code at all apply: you can introduce bugs, you can test something else than you intend, or you can break the application altogether. In some cases, for example if you wanted to restrict all outbound traffic from your application, there might be a lot of places in your code that need changes, so a platform-level approach might be more suitable.

The goal of this chapter is to show you that both approaches can be useful, and demonstrate, that it's not only for SREs - everyone can do chaos engineering, even if it's only on a single application.

8.5 Summary

- Building fault injection directly into an application can be an easy way of practicing chaos engineering
- If you work on an application, rather than the infrastructure level, it can be a good first step into chaos engineering, because it often requires no extra tooling
- Although it might require less work to set up, it also carries higher risks; the added code might contain bugs or introduce unexpected changes in behavior
- *"With great powers come great responsibility"* - The Peter Parker principle (https://en.wikipedia.org/wiki/With_great_power_comes_great_responsibility)

9

There's a monkey in my browser!

This chapter covers

- Applying chaos engineering to front-end code
- Overriding browser JavaScript requests to inject failure - with no source code changes

The time has come for us to visit the weird and wonderful world of JavaScript. Regardless of what stage of the love-hate relationship you two are at right now, there is no escaping JavaScript (JS) in one form or another. If you're part of the 4.5 billion people using the internet, you're almost certainly running JS, and the applications keep getting more and more sophisticated. If the recent explosion in popularity of frameworks for building rich front-ends, like React.js (https://github.com/facebook/react) and Vue.js (https://github.com/vuejs/vue) is anything to go by, it doesn't look like that situation is about to change.

The ubiquitous nature of JavaScript makes for an interesting angle for chaos engineering experiments. On top of the layers you've covered in the previous chapters, from the infrastructure level to the application level, there is another layer where the failure can occur (and therefore can be injected): the front-end JavaScript. The proverbial cherry on the equally proverbial cake.

In this chapter, you'll take a real, open-source application and see how you can inject slowness and failure into it with just a few lines of extra code that can be added to a running application on the fly. If you love JS, come and learn new ways it can be awesome. If you hate it, come and see how it can be used as a force for good. And to make it more real, let's start with a scenario.

9.1 Scenario

One of the neighboring teams is looking for a better way of managing their PostgreSQL (https://www.postgresql.org/) databases. They evaluated a bunch of free, open-source options, and they suggested a PostgreSQL database UI called pgweb (https://github.com/sosedoff/pgweb) as the way forward. The only problem is that the manager of that team is pretty old school. He reads HackerNews (https://news.ycombinator.com/news) through a plugin in his emacs, programs his microwave directly in Assembly, has JavaScript disabled on all his kids' browsers, and uses a Nokia 3310 (2000 was the last year they made a proper phone) to avoid being hacked.

To resolve the conflict between the team and their manager, both parties turn to you, asking to take a look at pgweb from the chaos engineering perspective and see how reliable it is--and in particular, at the JavaScript that the manager is so distrustful of. Not too sure what you're getting yourself into, you accept, of course.

In order to help them, you're going to need to understand what pgweb is doing, and then design and run some meaningful experiments. Let's start by looking into how pgweb actually works.

9.1.1 Pgweb

Pgweb is written in Go, and lets you connect to any PostgreSQL 9.1+ database and manage all the usual aspects of it, such as browse and export the data, execute queries, and insert new data.

It's distributed as a simple binary, and it's preinstalled, ready to use inside of the VM shipped with this book. The same goes for an example PostgreSQL installation, without which you wouldn't have anything to browse (as always, refer to appendix A for installation instructions if you don't want to use the VM). Let's bring it all up.

First, start the database, by running the following command. The database is prepopulated with some example data:

```
sudo service postgresql start
```

The credentials and data needed for this installation are the following:

- user: chaos
- password: chaos
- some example data in a database called booktown

To start pgweb using these credentials, all you need to do is run the following command:

```
pgweb --user=chaos --pass=chaos --db=booktown
```

And voila! You will see the output similar to the following, inviting you to open a browser (bold font):

```
Pgweb v0.11.6 (git: 3e4e9c30c947ce1384c49e4257c9a3cc9dc97876) (go: go1.13.7)
```

```
Connecting to server…
Connected to PostgreSQL 10.12
Checking database objects…
Starting server…
To view database open http://localhost:8081/ in browser
```

Go ahead and browse to http://localhost:8081. You will see the neat pgweb UI. On the left, it will show you the available tables that you can click to start browsing the data. It will look very similar to figure 9.1.

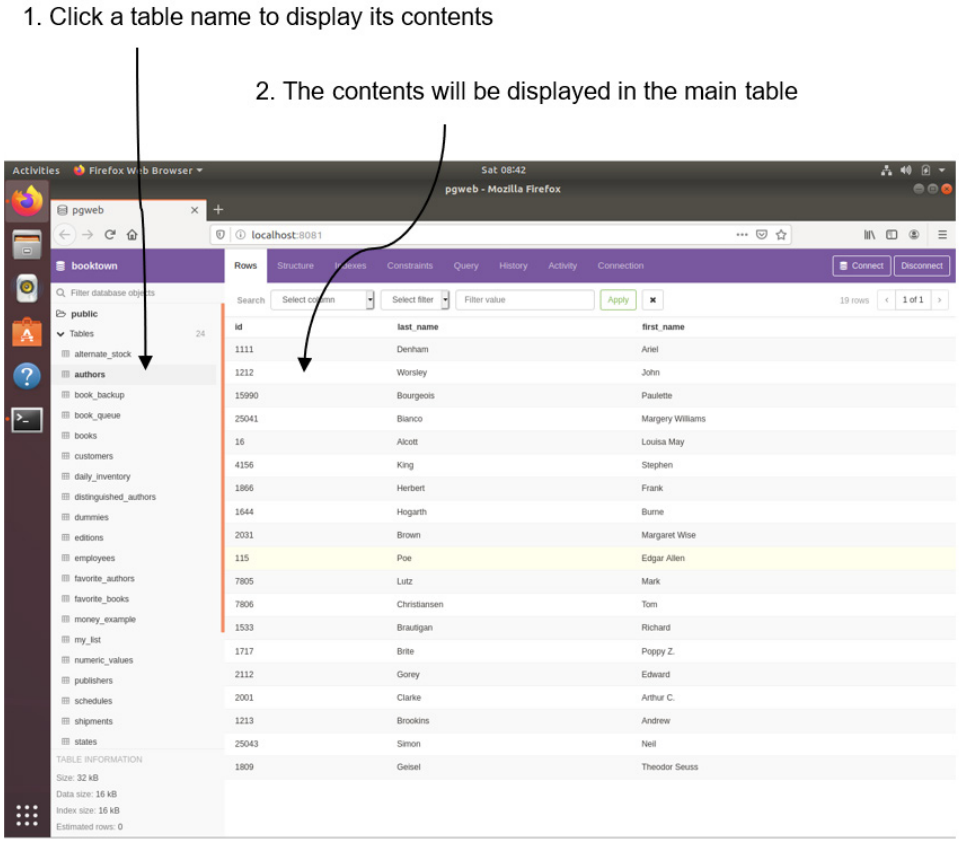


Figure 9.1. The UI of pgweb in action, displaying some example data

As you click around the website, you will see new data being loaded. From the chaos engineering perspective, every time there is data being loaded, it means an opportunity for failure. Let's see what is happening behind the scenes to populate the screen with that new data.

9.1.2 Pgweb - implementation details

In order to design a chaos experiment, we need to first understand how the data is loaded. Let's see howis is populated. Modern browsers make it easy to look into what's going on under the hood. I'm going to use Firefox that's open source and accessible in your VM, but the same thing can be done in all major browsers.

While browsing the pgweb UI, open the Web Developer tools on Network tab by pressing Ctrl-Shift-E (or going to Tools -> Web Developer -> Network in the firefox menu). You will see a new pane open at the bottom of the screen. It will initially be empty.

Now, click to select another table on the pgweb menu on the left. You will see the Network pane populate with three different requests. For each request, you will see the status (HTTP response code), method (GET), domain (localhost:8081), the file (endpoint) requested, a link to the code that made the request as well as some other details. Figure 9.2 shows what it looks like in my VM.

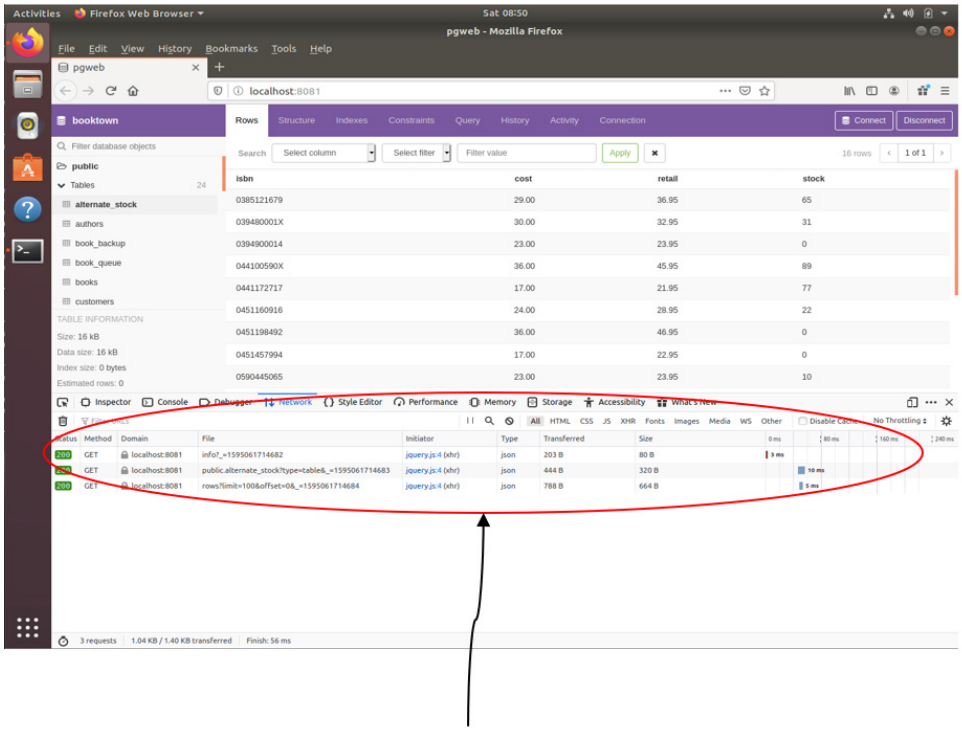


Figure 9.2. Network view in Firefox, showing requests made by pgweb from JavaScript

The cool stuff doesn't end here either: you can now click any of these three requests, and an extra pane, this time on the right, shows more details about it. Click the request to the "info" endpoint. A new pane opens, with extra details, just like in figure 9.3. You can see the headers sent and received, cookies, the parameters sent, response received, and more.

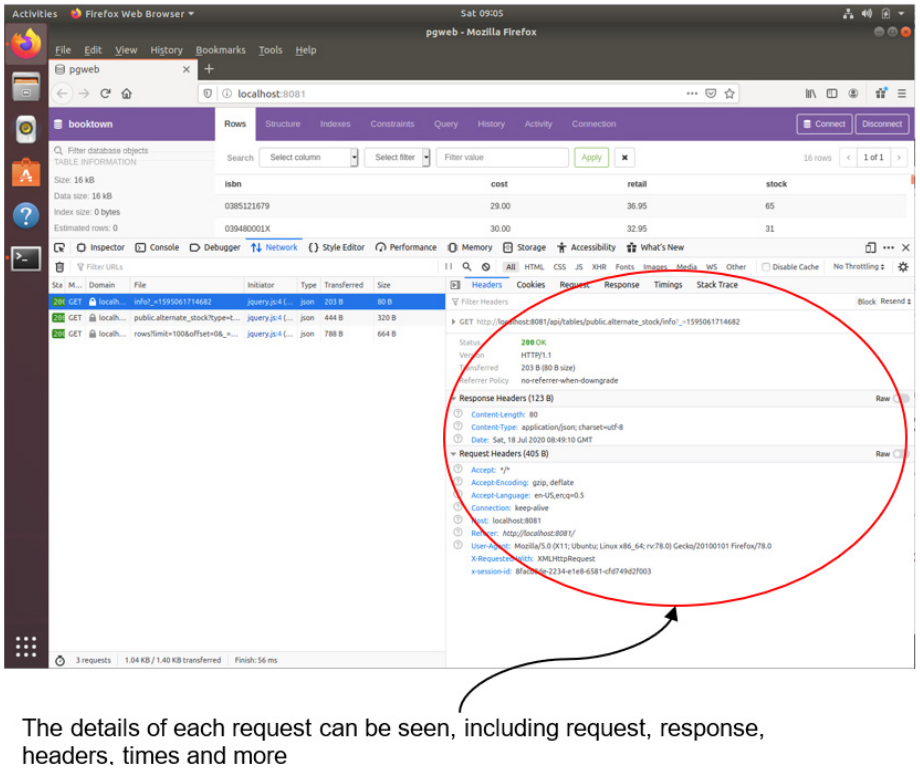


Figure 9.3. Request details view in Network tab of Web Developer tools in Firefox, displaying a request made by pgweb UI

Looking at these three requests gives you a lot of information about how the UI is actually implemented. For every action the user makes, you can see in the "initiator" column that the UI leverages *jquery* (https://jquery.com/), a very popular JavaScript library, to make requests to the backend. And we can see all of that before we even looked at any source code. The browsers we have today sure came a long way from the days of IE6!

So let's put all of this together.

1. When you browse to see the pgweb UI, your browser connects to the HTTP server built-into the pgweb application. It sends back the basic webpage, and the JavaScript code that together make the UI.
2. When you click something in the UI, the JavaScript code makes a request to the pgweb HTTP server to load the new data, like the contents of a table, and displays the data it receives in the browser, by rendering it as part of the webpage.
3. To return that data to the UI, the pgweb HTTP server reads the data from the PostgreSQL database.
4. Finally, the browser receives and displays the new data.

This process is summarized in figure 9.4.

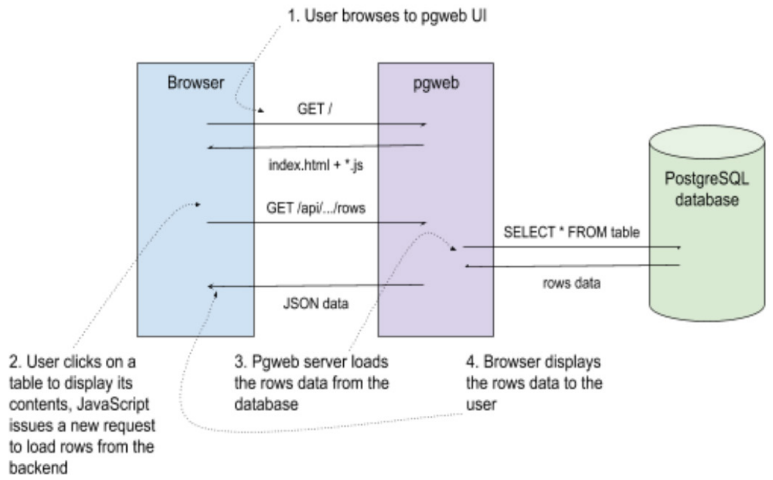


Figure 9.4. Events that happen when users browse the pgweb UI to display table contents

This is a pretty common sight along recent web applications, and it's often referred to as a *single-page application* or SPA (https://en.wikipedia.org/wiki/Single-page_application), because only the initial "traditional" webpage is served, and all the content then is displayed through JavaScript code manipulating it.

Feel free to poke around some more. When you're done, let's design a chaos experiment.

9.2 Experiment 1: adding latency

You're running pgweb and PostgreSQL locally, so you're not exposed to any networking latencies while using it. The first idea you might have is to check how the application copes with such latencies. Let's explore that idea.

In the previous chapters you've covered how to introduce latencies on various levels, and you could use that knowledge to add latency between the pgweb server and the database. But we're here to learn, so this time, let's focus on how to do that in the JavaScript application itself. This way you add yet another tool to your chaos engineering toolbox.

You saw that there were three different requests made when you clicked a table to display. They were all made in quick succession, so it's not clear whether they're prone to cascading delays (where requests are made in a sequence, so all the delays add up), and that's something that's probably worth investigating. And as usual, the chaos engineering way to do that is to add the latency and see what happens. Let's turn this idea into a chaos experiment.

9.2.1 Experiment 1 - plan

Let's say that we would like to add a one-second delay to all the requests that are made by the JavaScript code of the application, when the user selects a new table to display. An educated guess is that all the three requests we saw earlier are done in parallel, rather than sequentially, because there doesn't seem to be any dependencies between them. Therefore, we expect the overall action to take about one second longer than before. In terms of observability, we should be able to leverage the built-in timers that the browser offers to see how long it takes. So the plan for the experiment is the following:

1. **Observability**: use the timer built into the browser to read the time taken to execute all three requests made by the JavaScript code
2. **Steady state**: read the measurements from the browser before we implement the experiment
3. **Hypothesis**: if we add a one-second delay to all requests made from the JavaScript code of the application, the overall time it takes to display the new table will increase by one second
4. **Run the experiment**!

As always, let's start with the steady state.

9.2.2 Experiment 1 - steady state

Let me show you how to use the timeline built into Firefox to establish how long the requests made by clicking a table name really take. In the browser with the pgweb UI, with the Network tab still open (press Ctrl-Shift-E to reopen it, if you closed it before), let's clean the inputs. You can do that by clicking the trashcan icon in the left top corner of the Network pane. It should wipe the list.

With this clean slate, select a table in the left menu of the UI by clicking its name. You will see another three requests made, just like you did before. But this time, I'd like to focus your attention on two things. First, the rightmost columns in the list display a timeline, where each request is represented by a bar, starting at the time the request was issued, and ending when it was resolved. The longer the request took, the longer the bar. It will look like figure 9.5.

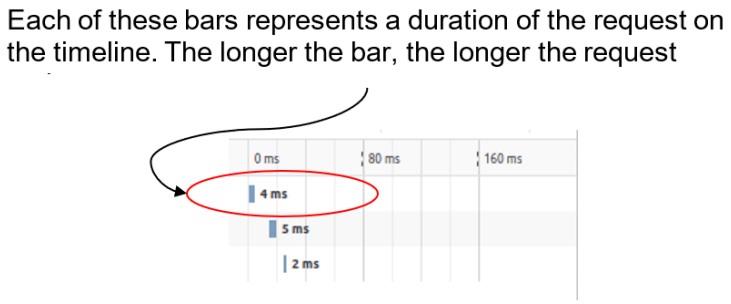


Figure 9.5. Firefox's timeline showing three requests issued and the times they took to complete

Second, at the bottom of the page there is also a line saying "Finish," that displays the total time between the first request started and the last event is finished, within the ones you captured. In my test runs, the number seemed to hover around the 25ms mark.

So there's our steady state. We don't have an exact number from between the user click action, but we have the time from the beginning of the first request to the end of the last one, and that number is around 25ms. That should be good enough for our use. Let's see how to add the actual implementation!

9.2.3 Experiment 1 - implementation

One of the reasons why people dislike JavaScript is that it's really easy to shoot yourself in the foot, for example by accidentally overriding a method or using an undefined variable - very few things are prohibited. And while that is a valid criticism, it also makes it fun to implement chaos experiments.

You want to add latency to requests, so you need to find the place in the code that makes the requests. As it turns out, there are two main ways browser JavaScript can make requests:

1. `XMLHttpRequest` built-in class (https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest)
2. `Fetch API` (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

jQuery (and therefore by extension pgweb, which uses jQuery) uses XMLHttpRequest, so we'll focus on it here (don't worry - we'll look into the Fetch API later in this chapter).

In order to avoid disturbing the learning flow from the chaos engineering perspective, I'm going to make an exception here, skip directly to the code snippet, and add the explanation in the sidebar. If you're interested in JavaScript, read the sidebar now, but if you're here for chaos engineering, let's go straight to the point.

Overriding XMLHttpRequest.send()

To make a request, you first create an instance of XMLHttpRequest class, set all the parameters you care about, and then you call the parameterless `send` method that does the actual sending of the request. The documentation referenced earlier gives the following description of `send`:

```
XMLHttpRequest.send()
Sends the request. If the request is asynchronous (which is the default), this method returns as soon as the request is sent.
```

That means, that if you can find a way to somehow modify that method, you can add an artificial one-second delay. If only JavaScript was permissive enough to do that, and preferably do that on the fly, after all the other code is already set up, so that you can conveniently only affect the part of the execution flow you care about. But surely, something this fundamental to the correct functioning of the application mustn't be easily changeable, right? Any serious language would try to protect it from accidental overwriting, and so would JavaScript.

Just kidding! JavaScript won't bat an eye at you doing that. Let me show you how.

Back in the pgweb UI, open a console (in Firefox press Ctrl-Shift-K or go to Tools->Web Developer->Web Console in the menu). If you're not familiar with it, the console lets you execute arbitrary JavaScript. You can execute any valid code you want at any time in the console, and if you break something, you can just refresh the page and all changes will be gone. That's going to be the injection mechanism: just copy and paste the code you want to inject in the console.

What would the code look like? If you're not familiar with JavaScript, you're going to have to trust me that this is nothing straying too far out of the ordinary. Strap in.

First, you need to access the `XMLHttpRequest` object. In the browser, the global scope is called `window`, so to access `XMLHttpRequest`, you'll write `window.XMLHttpRequest`. OK, makes sense.

Next, JavaScript is a prototype-based language (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes), which means that for an object A to inherit a method from another object B, object A can set object B as its prototype. The `send` method is not actually defined on the `XMLHttpRequest` object itself, but on its prototype. So in order to access the method, you need to use the following mouthful: `window.XMLHttpRequest.prototype.send`. With this, you can store a reference to the original method as well as replace the original method with a brand new function. This way, the next time the pgweb UI code creates an instance of `XMLHttpRequest` and calls its `send` method, it's the overwritten function that will get called. A bit weirder, but JavaScript is still only warming up.

Now, what would that new function look like? To make sure that things continue working, it'll need to call the original `send` method after the one-second delay. The mechanics of calling a method with the right context are a little bit colorful (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply), but for the purposes of this experiment, just know that any function can be invoked with `.apply(this, arguments)` method, which takes a reference to the object to call the function as a method of, and a list of arguments to pass to it. And to make it easy to observe that the overwritten function was actually called, let's use a `console.log` statement to print a message to the console.

Finally, to introduce an artificial delay, you can use a built-in `setTimeout` function that takes two arguments: a function to call and a timeout to wait before doing that (in milliseconds). Note, that `setTimeout` isn't accessed through the `window` variable. Well, JavaScript is like that.

Putting this all together, you can construct the seven lines of weird that make up listing 9.1, which is ready to be copy pasted into the console window.

Take a look at listing 9.1 that contains a snippet of code that we can copy and paste directly into the console (to open it in Firefox press Ctrl-Shift-K or go to Tools->Web Developer->Web Console in the menu) to add a one-second delay to the send method of XMLHttpRequest.

Listing 9.1 XMLHttpRequest-3.js

```
const originalSend = window.XMLHttpRequest.prototype.send;              #A
window.XMLHttpRequest.prototype.send = function(){                      #B
          console.log("Chaos calling", new Date());                     #C
    let that = this;                                                    #D
    setTimeout(function() {                                             #E
        return originalSend.apply(that);                                #F
    }, 1000);                                                           #G
}
```

#A store a reference to the original send method for later use
#B override the send method in XMLHttpRequest's prototype with a new function
#C print a message to show that the function was called
#D store the context of the original call to later use when calling the original send
#E setTimeout to execute the function after a delay
#F return the result of the call to the original send method, with the stored context
#G use the delay of 1000ms

If this is your first encounter with JavaScript, I apologize. You might want to take a walk, but make it quick, because we're ready to run the experiment!

9.2.4 Experiment 1 - run!

Show time! Go back to the pgweb UI, refresh it if you've made any changes in the console and wait for it to load. Select a table from the menu on the left. Make sure the network tab is open (Ctrl-Shift-E on Firefox) and empty (use the trash bin icon to clean it up). You're ready to go.

1. Copy the code from listing 9.1.
2. Go back to the browser, open the console (Ctrl-Shift-K), paste the snippet and press enter.
3. Now go back to the network tab and select another table. It will take a bit longer this time, and you will see the familiar three requests made.
4. Focus on the timeline, on the rightmost column of the network tab. You will notice that the spacing (time) between the three requests is very similar to what we observed in our steady state. It will look something like the figure 9.6.

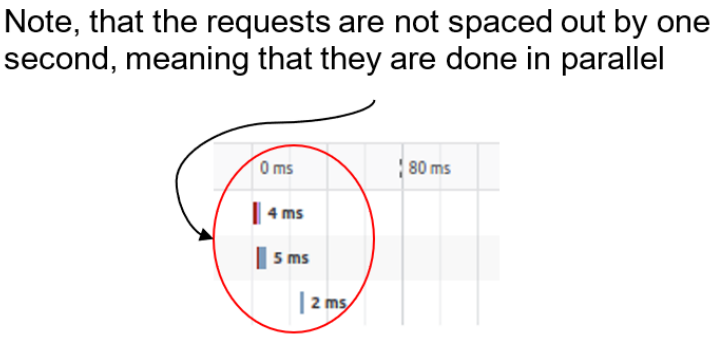


Figure 9.6. Firefox's timeline showing three requests made from JavaScript

What does it mean? You added the same one-second delay to each call of the `send` method. Because the requests on the timeline are not spaced by one second, you can conclude that they're not made in a sequence, but rather all in parallel. This is good news, because it means that with a slower connection, the overall application should slow down in a linear fashion. In other words, there doesn't seem to be a bottleneck in this part of the application.

But the hypothesis was about the entire time it takes to execute the three requests, so let's confirm whether that's the case. We can't read it directly from the timeline, because we added the artificial delay before the request is issued, and the timeline only begins at the time the first request actually starts. If we wanted to go deep down the rabbit hole, we could override more functions to print different times and calculate the overall time it took.

But because our main goal here is just to confirm that the requests aren't waiting for one another without actually reading the source code, we can do something much simpler. Go back to the console. You will see three lines starting with *Chaos calling*, printed by the snippet of code you used to inject the delay. They also print the time of the call. Now, back in the network tab, select the last request, and look at the response headers. One of them will have the date of the request. Compare the two and note that they are one second apart. In fact, you can compare the other requests, and they'll all be one second apart from the time our overwritten function was called. The hypothesis was correct, case closed!

This was fun. Ready for another experiment?

9.3 Experiment 2: adding failure

Since we're at it, let's do another experiment, this time focusing on the error handling that pgweb implements. Running pgweb locally, you're not going to experience any connectivity issues, but in the real world you definitely will. How do you expect the application to behave in face of such networking issues? Ideally it would have some retry mechanism where applicable,

and if that fails, it would present the user with a clear error message and avoid showing stale or inconsistent data. A simple experiment basically designs itself:

1. **Observability**: observe whether the UI shows any errors or stale data
2. **Steady state**: no errors or stale data
3. **Hypothesis**: if we add an error on every other request that the UI JavaScript is making, we should see an error and no inconsistent data every time we select a new table
4. **Run the experiment**!

You have already clicked around and confirmed the steady state (no errors), so let's jump directly to the implementation.

9.3.1 Experiment 2 - implementation

To implement this experiment you can use the same injection mechanism from experiment 1 (paste a code snippet in the browser console) and even override the same method (send). The only new piece of information we need is this: how does `XMLHttpRequest` fail in normal conditions?

To find out, we'll need to look it up in the documentation at https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest. As it turns out, it uses *events*. If you're not familiar with events in JavaScript, it's a simple but flexible mechanism for communicating between objects. An object can emit (dispatch) events (simple objects with a name and optionally a payload with extra data). When that happens, the dispatching object checks if there are functions registered to receive that name, and if there are, they're all called with the event. Any function can be registered to receive (listen to) any events on an object emitting objects. You can see a visual summary of this in figure 9.7. This is used extensively in web applications to handle asynchronous events; for example, those generated by user interaction (click, keydown, etc).

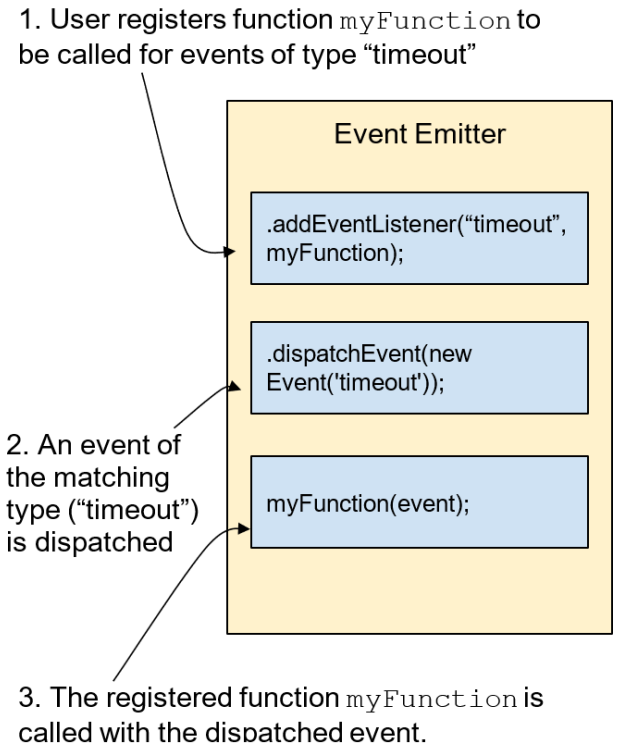


Figure 9.7. High-level overview of events in JavaScript

The Events section of the `XMLHttpRequest` documentation lists all the events that an instance of `XMLHttpRequest` can dispatch. One event looks particularly promising - the `error` event, that’s described like this:

```
error
Fired when the request encountered an error.
      Also available via the onerror property.
```

It’s a legal event that can be emitted by an instance of `XMLHttpRequest`, and it’s one that should be handled gracefully by the pgweb application, which makes it a good candidate for our experiment!

Now you’ve now got all the elements, let’s assemble them into a code snippet. Just like before, you’ll need to override the `window.XMLHttpRequest.prototype.send` but keep a reference to the original method. You’ll need a counter to keep track of which call is “every other one.” And you can use the `dispatchEvent` method directly on the `XMLHttpRequest` instance to dispatch a new event that you can create with a simple `new Event('timeout')`. Finally, you want to either

dispatch the event or do nothing (just call the original method), based on the value of the counter. You can see a snippet doing just that in listing 9.2.

Listing 9.2 XMLHttpRequest-4.js

```
const originalSend = window.XMLHttpRequest.prototype.send;                    #A
var counter = 0;                                                              #B
window.XMLHttpRequest.prototype.send = function(){                            #C
    counter++;
    if (counter % 2 == 1){
        return originalSend.apply(this, [...arguments]);                      #D
    }
    console.log("Unlucky " + counter + "!", new Date());
    this.dispatchEvent(new Event('error'));                                   #E
}
```

#A store a reference to the original send method for later use
#B keep a counter to only act on every other call
#C override the send method in XMLHttpRequest's prototype with a new function
#D on even calls, relay directly to the original method - noop
#E on odd calls, instead of calling the original method, dispatch an "error" event

With that, you're all set to run the experiment. The suspense is unbearable, so let's not waste any more time and do it!

9.3.2 Experiment 2 - run

Go back to the pgweb UI and refresh (F5, Ctrl-r, Cmd-r) to erase any artifacts of the previous experiments. Select a table from the menu on the left. Make sure the network tab is open (Ctrl-Shift-E on Firefox) and empty (use the trash bin icon to clean it up). Copy the code from listing 9.2, go back to the browser, open the console (Ctrl-Shift-K), paste the snippet and hit enter.

Now, try selecting three different tables in a row, by clicking their name in the pgweb menu on the left. What do you notice? You will see that rows of data, as well as the table information, are not refreshed every time you click it, but only every other time. What's worse, there is no visual error message to tell you there was an error. So you can select a table, see incorrect data, and not know that anything went wrong.

Fortunately, if you look into the console, you're going to see an error message like the following, for every other request:

```
Uncaught SyntaxError: JSON.parse: unexpected character at line 1 column 1 of the JSON
      data
```

Although you didn't get a visual presentation of the error in the UI, you can still use the information from the console to dig down and uncover the underlying issue. If you're curious, the reason for this is because the error handler used in the pgweb UI for all the requests accesses a property that is not available when there was an error before the response was received. It

tries to parse it as JSON, which results in an exception being thrown and the user getting stale data and no visible mention of the error, like in the following line:

```
parseJSON(xhr.responseText)
```

> **NOTE** Open source for the win
>
> Thanks to the open-source nature of the project, you can see the line in the project's repo on Github: https://github.com/sosedoff/pgweb/blob/3e4e9c30c947ce1384c49e4257c9a3cc9dc97876/static/js/app.js#L77. Technically, with a GUI implemented in JavaScript, we could always take a peek into what's running in the browser, but having it out in the open for everyone to see is pretty neat.

So there you have it. With a grand total of 10 lines of (verbose) code and about 1 minute of testing, we were able to find issues with the error handling of a popular, good-quality open source project. It goes without saying that it doesn't take away from the awesomeness of the project itself. Rather, it's an illustration of how little effort it sometimes takes to get benefits from doing chaos engineering.

JavaScript overdose can have a lot of serious side-effects, so I'm going to keep it short. Last pit stop to show you two more neat tricks, and we're done.

9.4 Other good-to-know topics

Before we wrap up the chapter, I want to give you a little bit more information on two more things that might be useful for implementing your JavaScript-based chaos experiments. Let's start with the Fetch API.

9.4.1 Fetch API

Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) is a more modern replacement for `XMLHttpRequest`. Like `XMLHttpRequest`, it allows you to send requests and fetch resources. The main interaction point is through the function `fetch` accessible in the global scope. Unlike `XMLHttpRequest` it returns a Promise object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise). In its basic form, you can just call fetch with a URL, and then attach the .then and .catch handlers, like you would with any other promise. To try it, go back to the pgweb UI, open a console, and run the following snippet (fetch, then and catch methods in bold) to try to fetch a non-existent endpoint `/api/does-not-exist`:

```
fetch("/api/does-not-exist").then(function(resp) {
        // deal with the fetched data
        console.log(resp);
      }).catch(function(error) {
        // do something on failure
        console.error(error);
      });
```

It will print the response as expected, complaining about the status code 404 (Not Found). Now, you must be wondering, "Surely, this time, with a modern code base, they designed it to be harder to override." Nope. You can use the exact same technique from the previous experiments to override it. Listing 9.3 puts it all together.

Listing 9.3 fetch.js

```
const original = window.fetch;                          #A
window.fetch = function(){                              #B
    console.log("Hello chaos");
    return original.apply(this, [...arguments]);        #C
}
```

#A store a reference to the original fetch function
#B override the fetch function in the global scope
#C call the original fetch function after printing something

To test it, copy the code from listing 9.3, paste it in the console, and press enter. Then paste the previous snippet once again. It will run the same way it did before, but this time it will print the "Hello chaos" message.

That's it. Worth knowing, in case the application you work with is using this API, rather than `XMLHttpRequest`, which is increasingly more likely every day. OK, one last stop and we're done. Let's take a look at the built-in throttling.

9.4.2 Throttling

One last tidbit I want to leave you with is the built-in throttling capacity that browsers like Firefox and Chrome offer these days. If you've worked with front-end code before, you're definitely familiar with it, but if you're coming from a more low-level background, it might be a neat surprise to you!

Go back to the pgweb UI in the browser. When you open the Web Developer tools on the Network tab by pressing Ctrl-Shift-E (or going to Tools -> Web Developer -> Network), on the right side, just above the list of calls there is a little drop-down menu that defaults to "No Throttling." You can change that to the various presets listed, like GPRS, Good 2G or DSL, that emulate the networking speed that these connections offer (figure 9.8).

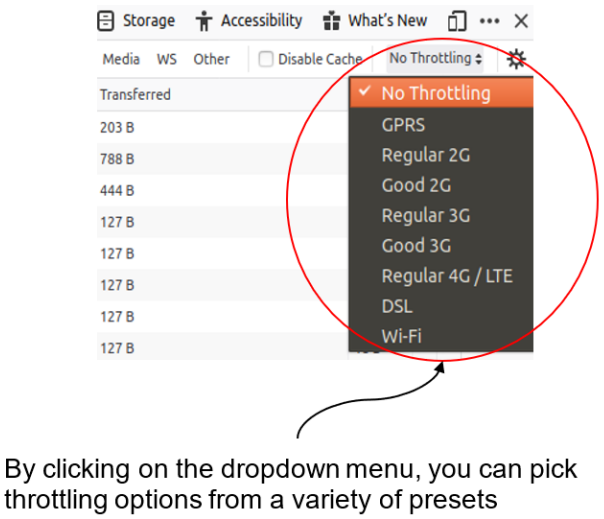


Figure 9.8. Networking throttling options built-into Firefox

If you want to inspect how the application performs on a slower connection, try setting this to GPRS! It's a neat trick to know and might come in handy during your chaos engineering adventures. And that's a JavaScript wrap!

9.5 Summary

- JavaScript's malleable nature makes it easy to inject code into applications running in the browser
- There are currently two main ways of making the requests (`XMLHttpRequest` and `Fetch API`) and they both lend themselves well to code injection in order to introduce failure
- Modern browsers offer a lot of useful tool through their Developer Tools, including insight into the requests made to the backend, as well as the console, which allows for executing arbitrary code