THE DEVOPS TOOLKIT

Kubernetes Chaos Engineering

Viktor Farcic
Darin Pope

KUBERNETES CHAOS ENGINEERING WITH CHAOS TOOLKIT AND ISTIO

The DevOps Toolkit: Kubernetes Chaos Engineering

Kubernetes Chaos Engineering With Chaos Toolkit And Istio

Viktor Farcic and Darin Pope

This book is for sale at http://leanpub.com/the-devops-2-7-toolkit

This version was published on 2020-11-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By These Authors

Books by Viktor Farcic

The DevOps 2.3 Toolkit: Kubernetes

The DevOps 2.5 Toolkit: Monitoring, Logging, and Auto-Scaling Kubernetes

The DevOps Toolkit: Catalog, Patterns, And Blueprints

Books by Darin Pope

The DevOps Toolkit: Catalog, Patterns, And Blueprints

Contents

Introduction To Kubernetes Chaos Engineering

There are very few things as satisfying as destruction, especially when we're frustrated.

How often did it happen that you have an issue that you cannot solve and that you just want to scream or destroy things? Did you ever have a problem in production that is negatively affecting a lot of users? Were you under a lot of pressure to solve it, but you could not "crack" it as fast as you should. It must have happened, at least once, that you wanted to take a hammer and destroy servers in your datacenter. If something like that never happened to you, then you were probably never in a position under a lot of pressure. In my case, there were countless times when I wanted to destroy things. But I didn't, for quite a few reasons. Destruction rarely solves problems, and it usually leads to negative consequences. I cannot just go and destroy a server and expect that I will not be punished. I cannot hope to be rewarded for such behavior.

What would you say if I tell you that we can be rewarded for destruction and that we can do a lot of good things by destroying stuff? If you don't believe me, you will soon. That's what chaos engineering is about. It is about destroying, obstructing, and delaying things in our servers and in our clusters. And we're doing all that, and many other things, for a very positive outcome.

Chaos engineering tries to find the limits of our system. It helps us deduce what are the consequences when bad things happen. We are trying to simulate the adverse effects in a controlled way. We are trying to do that as a way to improve our systems to make them more resilient and capable of recuperating and resisting harmful and unpredictable events.

That's our mission. We will try to find ways how we can improve our systems based on the knowledge that we will obtain through the chaos.

Who Are We?

Before we dive further, let me introduce you to the team comprised of me, Viktor, and another guy, which I will introduce later.

Who Is Viktor?

Let's start with me. My name is Viktor Farcic. I currently work in CloudBees[1]. However, things are changing and, by the time you are reading this, I might be working somewhere else. Change is constant, and one can never know what the future brings. At the time of this writing, I am a

[1] https://www.cloudbees.com/

principal software delivery strategist and developer advocate. It's a very long title and, to be honest, I don't like it. I need to read it every time because I cannot memorize it myself. But that's what I am officially.

What else can I say about myself? I am a member of the Google Developer Experts (GDE)[2] group, and I'm one of the Docker Captains[3]. You can probably guess from those that I am focused on containers, Cloud, Kubernetes, and quite a few other things.

I'm a published author. I wrote quite a few books under the umbrella of The DevOps Toolkit Series[4]. I also wrote DevOps Paradox[5] and Test-Driven Java Development[6]. Besides those, there are a few Udemy courses[7].

I am very passionate about DevOps, Kubernetes, microservices, continuous integration, and continuous delivery, and quite a few other topics. I like coding in Go.

I speak in a lot of conferences and community gatherings, and I do a lot of workshops.

I have a blog TechnologyConversations.com[8] where I keep my random thoughts, and I co-host a podcast DevOps Paradox[9].

What really matters is that I'm curious about technology, and I often change what I do. A significant portion of my time is spent helping others (individuals, communities, or companies).

Now, let me introduce the second person that was involved in the creation of this book. His name is Darin, and I will let him introduce himself.

Who Is Darin?

My name is Darin Pope. I'm currently working at CloudBees[10] as a professional services consultant. Along with Viktor, I'm the co-host of DevOps Paradox[11].

Whether its figuring out the latest changes with Kubernetes or creating more content to share with our listeners and viewers, I'm always learning. Always have, always will.

Principles Of Chaos Engineering

What is chaos engineering? What are the principles behind it?

We can describe chaos engineering as a discipline of experimenting on a system to build confidence in the system's capability to withstand turbulent conditions in production. Now, if

[2] https://developers.google.com/community/experts
[3] https://www.docker.com/community/captains
[4] https://www.devopstoolkitseries.com/
[5] https://amzn.to/2myrYYA
[6] http://www.amazon.com/Test-Driven-Java-Development-Viktor-Farcic-ebook/dp/B00YSIM3SC
[7] https://www.udemy.com/user/viktor-farcic/
[8] https://technologyconversations.com/
[9] https://www.devopsparadox.com/
[10] https://www.cloudbees.com/
[11] https://www.devopsparadox.com/

that was too confusing and you prefer a more straightforward definition of what chaos engineering is, I can describe it by saying that you should **be prepared because bad things will happen**. It is inevitable. Bad things will happen, often when we don't expect them. Instead of being reactive and waiting for unexpected outages and delays, we are going to employ chaos engineering and try to simulate adverse effects that might occur in our system. Through those simulations, we're going to learn what the outcomes of those experiments are and how we can improve our system by building confidence by making it resilient and by trying to design it in a way that it is capable of withstanding unfavorable conditions happening in production.

Are You Ready For Chaos?

Before we proceed, I must give you a warning. **You might not be ready for chaos engineering**. You might not benefit from this book. You might not want to do it. Hopefully, you're reading this chapter from the sample (free) version of the book, and you did not buy it yet since I am about to try to discourage you from reading.

When can you consider yourself as being ready for chaos engineering? Chaos engineering requires teams to be very mature and advanced. Also, if you're going to practice chaos engineering, be prepared to do it in production. We don't want to see how, for example, a staging cluster behaves when unexpected things happen. Even if we do, that would be only a practice. "Real" chaos experiments are executed in production because we want to see how the real system used by real users is reacting when bad things happen.

Further on, you, as a company, must be prepared to have sufficient budget to invest in real reliability work. It will not come for free. There will be a cost for doing chaos engineering. You need to invest time in learning tools. You need to learn processes and practices. And you will need to pay for the damage that you will do to your system.

Now, you might say that you can get the budget and that you can do it in production, but there's more. There is an even bigger obstacle that you might face.

You must have enough observability in your system. You need to have a relatively advanced monitoring and alerting processes and tools so that you can detect harmful effects of chaos experiments. If your monitoring setup is non-existent or not reliable, then you will be doing damage to production without being able to identify the consequences. Without knowing what went wrong, you won't be able to (easily) restore the system to the desired state.

On the other hand, you might want to jump into chaos engineering as a way to justify investment in reliability work and in observability tools. If that's the case, you might want to employ the practices from this book as a way to show your management that reliability investment is significant and that being capable of observing the system is a good thing.

So you can look at it from both directions. In any case, I'm warning you that this might not be the book for you. You might not be in the correct state. Your organization might not be mature enough to be able to practice what I am about to show you.

Examples Of Chaos Engineering

Right now, you might be saying, "okay, I understand that chaos engineering is about learning from the destruction, but what does it really mean?" Let me give you a couple of use-cases. I cannot go through all the permutations of everything we could do, but a couple of examples might be in order.

We can validate what happens if you have improper fallback settings when a service is unavailable. What happens when a service is not accessible, one way or another. Or, for example, what happens if an app is retrying indefinitely to reach a service without having properly tuned timeouts? What is the result of outages when an application or a downstream dependency receives too much traffic or when it is not available? Would we experience cascading errors when a single point of failure crashes an app? What happens when our application goes down? What happens when there is something wrong with networking? What happens when a node is not available?

Those are just a few of the examples we will explore through practical exercises.

For quite a few reasons, we will not be able to explore everything that you can or that you should do. First of all, chaos experiments that we can run depend on a system. Not all the experiments apply to everybody. Also, going through all the possible permutations would require too much time. We can even say that the number of experiments and their permutations is infinite. I could write thousands of pages, and I'd still not be able to go through all of them.

Instead, I will try to teach you how to think, how to use some tools, and how to figure out what makes sense in your organization, in your team, and in your system.

Now that we went through a few examples (more is to come), let's talk about the principles of chaos engineering.

The Principles And The Process

What we usually want to do is build a hypothesis around the steady-state behavior. What that means is that we want to define how our system, or a part of it, looks like. Then, we want to perform some potentially damaging actions on the network, on the applications, on the nodes, or on any other component of the system. Those actions are, most of the time, very destructive. We want to create violent situations that will confirm that our state, the steady-state hypothesis, still holds. In other words, we want to validate that our system is in a specific state, perform some actions, and finish with the same validation to confirm that the state of our system did not change.

We want to try to do chaos engineering based on real-world events. It would be pointless to try to do things that are not likely to happen. Instead, we want to focus on replicating the events that are likely going to happen in our system. Our applications will go down, our networking will be disrupted, and our nodes will not be fully available all the time. Those are some of the things that do happen, and we want to check how our system behaves in those situations.

We want to run chaos experiments in production. As I mentioned before, we could do it in a non-production system. But that would be mostly for practice and for gaining confidence in chaos

experiments. We want to do it in production because that's the "real" system. That's the system at its best, and our real users are interacting with it. If we'd just do it staging or in integration, we would not get a real picture of how the system in production behaves.

We want to automate our experiments to run continuously. It would be pointless to run an experiment only once. We could never be sure what the right moment is. When is the system in conditions under which it would produce some negative effect? We should run the experiments continuously. That can be every hour, every day, every week, every few hours, or every time some event is happening in our cluster. Maybe we want to run experiments every time we deploy a new release or every time we upgrade the cluster. In other words, experiments are either scheduled to run periodically, or they are executed as part of continuous delivery pipelines.

Finally, we want to reduce the blast radius. In the beginning, we want to start small and to have a relatively small blast radius of the things that might explode. And then, over time, as we are increasing confidence in our work, we might be expanding that radius. Eventually, we might reach the level when we're doing experiments across the whole system, but that comes later. In the beginning, we want to start small. We want to be tiny.

The summary of the principles we discussed is as follows.

- Build a hypothesis around steady-state
- Simulate real-world events
- Run experiments in production
- Automate experiments and run them continuously
- Minimize blast radius

Now that it is a bit clearer what chaos engineering is and what are the principles behind it, we can turn our attention towards the process. It is repetitive.

To begin with, we want to define a steady-state hypothesis. We want to know how does the system look like before and after some actions. We want to confirm the steady-state, and then we want to simulate some real-world events. And after those events, we want to confirm the steady-state again. We also want to collect metrics, to observe dashboards, and to have alerts that will notify us when our system misbehaves. In other words, we're trying very hard to disrupt the steady-state. The less damage we're able to do, the more confidence we will have in our system.

The summary of the process we discussed is as follows.

1. Define the steady-state hypothesis
2. Confirm the steady-state
3. Produce or simulate "real world" events
4. Confirm the steady-state
5. Use metrics, dashboards, and alerts to confirm that the system as a whole is behaving correctly.

Chaos Experiments Checklist

Before we dive into practical examples, we'll define a checklist of the things we might want to accomplish. What could be our goals?

To begin with, we probably want to see what happens when we terminate an instance of an application. We also might want to see what happens when we partially terminate network or delay network requests. We might want to increase the latency of the network. We might want to simulate denial of service attacks. We might want to drain or even delete a node.

We're going to focus on instances of our applications, on networking, and on nodes. For all that to be visible, we might need to create some reports as well as to send notifications to all those interested. Finally, we almost certainly want to run our experiments inside a Kubernetes cluster. Not only that we want to experiment targeting a Kubernetes cluster, but we want to run our experiments inside such a cluster.

The summary of the tasks we want to accomplish is as follows.

- Terminate instance of an app
- Partially terminate network
- Increase latency
- Simulate Denial of Service (DoS) attacks
- Drain a node
- Delete a node
- Create reports
- Send notifications
- Run the experiments inside a Kubernetes cluster

It might be just as important to define what we will NOT do. One thing that I will not go through is how to modify the internals of an application. We will not touch the architecture of our applications. We are going to assume that applications are as they are. That is not because we shouldn't be modifying the internals of our apps. We definitely should be adapting the code and the architecture of applications based on the results of our experiments.

Nevertheless, that's not the subject of this book, simply because I would need to start guessing which programming language you are using. I would need to provide examples in Go, in Java, in NodeJS, in Python, etc. All in all, we will not be modifying applications.

Also, we will never permanently change a definition of anything. Whatever we do, we will try (when possible) to undo the effects of our experiments. If we damage a network, we will have to undo the changes that caused the damage. Nevertheless, we might not always be successful in that. Sometimes, we might not be able to roll back the results of our experiments. We'll do our best, but we are yet to see whether we'll succeed.

Next, we'll explore how this book, or at least parts of it, is organized.

How Is The Book Organized?

We will have different chapters, and each section will start from scratch. You will be able to jump to any individual chapter and run all the examples independently of any other. For that to work, I will give you instructions on how to create a cluster, if it's not already running, how to deploy a demo application, and how to set up everything we need in a cluster. After a short introduction to the theme of a chapter, each will start with such instructions. Similarly, at the end of each chapter, we will destroy everything we created.

All in all, each chapter will start from scratch. We'll set up everything we need, we will do some exercises, and then we will destroy everything. Each chapter is independent of the others.

Now, as for Kubernetes clusters, I created examples and tested them in **Minikube**, in **Docker Desktop**, in **Google Kubernetes Engine (GKE)**, in **AWS Elastic Kubernetes Service (EKS)**, and in **Azure Kubernetes Service (AKS)**. That does not mean that you have to use one of those Kubernetes distributions. You can run the experiments anywhere you like, in any type of distribution running in any provider. However, I tested all the examples only in those previously mentioned, and sometimes you might need to modify some definitions or commands if you're using a different Kubernetes flavor.

If you run into any problems, contact me, and I will do my best to help you out. I might even extend this book to provide instructions for other Kubernetes distributions. You just need to let me know what your favorite is.

While we are at the subject of Kubernetes distributions, Minikube and Docker Desktop will work most of the time. Not always. Specifically, **we will not be able to experiment on nodes in Minikube and Docker Desktop** simply because they are single-node clusters with control plane and worker node mixed together. Damaging a single-node cluster would not produce the desired results. It will result in permanent destruction.

All in all, everything works and is tested in GKE, EKS, and AKS, while Minikube and Docker Desktop work most of the time. In those few cases, when something doesn't work in one of the local single-node clusters, you should be able to observe the outcomes from the outputs I'll provide. You should be able to see what is happening and learn, even if an example doesn't work in Minikube or Docker Desktop.

I will give you assignments. You will have homework that you might choose to do. Unlike some other books, those assignments will not be easy. They will be hard. They will be harder than the examples I will show you. If you choose to do the homework, expect it to be challenging. It won't be some easy stuff, because I want you to spend time with it and get yourself immersed in the world of chaos engineering by doing hard things, not easy ones.

I will not go through everything that the tooling we'll use will provide. Instead, I will assume that you can read the documentation and figure out all the additional modules and plugins and capabilities. We will only go through the most commonly used scenarios.

Finally, there is one more critical assumption that I'm making for this book. I will assume that you

are already proficient with Kubernetes. You will not learn through this book how to do some basic Kubernetes operations. I will also assume that you have at least a basic understanding of Istio. You don't need to be *Istio ninja*, but some basic knowledge is welcome. Also, you will see that I will be talking about monitoring, alerting, and dashboards. I will not go through those in detail.

All in all, the subject of this book is not to teach you Kubernetes, nor to teach you service mesh, nor to teach you monitoring and alerting. It is focused on chaos engineering in Kubernetes. I will, however, provide the references for material that you might want to go through if you're going to dive deeper into some of the topics that are not directly related to the subject of this book.

Off We Go

The book is about to start unfolding. We are about to dive into chaos engineering.

For now, please assume that anything can fail. That's the most important thing you should memorize. It should be engraved in your brain. Whatever can fail is likely to fail.

We are going to start small and expand over time. The first exercises might seem too basic. That's intentional. I want you to start small and to grow over time.

I want you to communicate with your colleagues. That is probably the most important thing in chaos engineering. It is essential that the lessons learned and the lessons that you will apply in your own system are communicated with the rest of your company, with your teammates, with your colleagues. It is vital that you all collaborate in solving the issues you will find.

In parallel with writing this book, I'm working on a video course based on the same material. Both were published at the same time. If you want to watch videos, that's great. If you're going to read this book, that's great as well. In some cases, you might want both. If that's the case, please send me an email (viktor@farcic.com), or a message on Twitter (@vfarcic), or join the DevOps20 Slack workspace[12] (my user is vfarcic). Any of those ways to communicate are okay. Send me a message, I will provide you with a coupon for the video course if you already purchased the book, and vice versa.

No matter the coupons, I highly encourage you to join the DevOps20 Slack workspace[13]. I am there, and I'll do my best to answer any questions and help with any issues you might have.

Finally, there is a mailing list[14] that you might want to subscribe to if you wish to receive notifications when I do some other courses, books, and other publicly accessible material.

[12] http://slack.devops20toolkit.com
[13] http://slack.devops20toolkit.com
[14] http://eepurl.com/bXonVj

The Requirements Guiding The Choice Of A Tool

Before we start defining and running chaos experiments, we need to pick a tool. Which one are we going to use?

Chaos Engineering is not yet the segment of the market that is well established and developed. Nevertheless, there are several tools we can pick from. We cannot use them all, at least not in this course. That would take too much time, and it would never end. So we'll have to pick one.

Before we select a tool, let's go through a couple of requirements that I believe are important.

The tool **should be open-source**. I am a huge believer in open-source. That does not mean that everything which we use should be open-source. There is a time and place where we should use enterprise editions of the software, and there is a time and place when we should use a service. But, in this case, I wanted to make sure that it is free that it is open. So, open-source is a requirement, at least for this book.

As long as it is possible and practical, the tool **should work both inside and outside Kubernetes**. Even though this book is focused on Kubernetes, it would be great if we could pick a tool that could do chaos engineering not only inside Kubernetes, but also outside. You may want to destroy nodes directly, or you may tweak some other aspect of your cluster and your infrastructure in general.

That begs the question. Why are we focused on Kubernetes, even though we want the tool that works both inside and outside Kubernetes? If I would show you how to do chaos engineering on AWS, then you might say that you need examples in Azure. Or you might prefer Google, or you might have an on-prem cluster based on VMware. It would be close to impossible for me to provide examples in all those (and other) platforms. That would require a few books. So, I did not want to focus on a particular hosting platform (e.g., only AWS, or only Azure) since that would not be inclusive. I wanted to make that the scope is as wide as possible, so the course is focused on Kubernetes, which happens to work (almost) anywhere. Everything you learn should work, more or less, in any Kubernetes distribution with any hosting provider.

Which Tool Should We Choose?

We established that the requirements for a tool we'll pick are to be open-source and to work both inside Kubernetes and outside Kubernetes. What are the options? What are the tools we could choose from?

To begin with, we could **do things manually**. We could modify and tweak Kubernetes resources ourselves, we could do some changes to service mesh, and so on and so forth. But manual execution

of experiments is not what we want. As I already mentioned before, I believe that the execution of chaos experiments should be automated. It should be executable periodically, or through continuous delivery pipelines. So, manual chaos is not an option.

Of course, we could automate things by writing our own scripts. But why would we do that? There are tools that can help us, and that can get us from nothing to something very fast. That does not exclude writing your own custom scripts. You're almost certainly going to end up creating your scripts sooner or later. But picking a tool that already does at least some of the things we need will get us to a certain level much faster.

Now that we know that we're not going to do manual chaos engineering and that we are not going to write all the scripts from scratch ourselves, let's see which tools do we have at our disposal.

There is **Chaos Monkey**[15], Simian Army[16], and other Netflix tools aimed at chaos engineering. What Netflix did with Chaos Monkey and the other tools is excellent. They were pioneers, at least among those that made their tools public. However, Chaos Monkey does not work well in Kubernetes. On top of that, it requires Spinnaker and MySQL. I don't believe that everybody should adopt Spinnaker only for chaos engineering. Not working well in Kubernetes is a huge negative point, and Spinnaker is another one. I'm not saying it's a bad tool. Quite the contrary. Spinnaker is very useful for certain things. However, the question is whether you should adopt it only because you want to do some chaos engineering? Most likely not. If you are already using it, Chaos Monkey might be the right choice, but I could not make that assumption.

Next, we have **Gremlin**[17]. It might be one of the best tools we have on the market. While I encourage you to try it out and see whether it fits your needs, it's a service (you cannot run it yourself), and it's not open-source. Since open-source is one of the requirements, Gremlin is out as well.

Further on, we have **PowerfulSeal**[18], which is immature and poorly documented. Besides that, it works only in Kubernetes, and that fails one of our requirements. We also have **kube-monkey**[19], which is inspired by Chaos Monkey but designed for Kubernetes. Just like PowerfulSeal, it is immature and poorly documented, and I would not recommend it. Then we have **Litmus**[20], which suffers from similar problems. It is better documented than other tools I mentioned, but it is still not there. It is green, and it is Kubernetes only. We also have **Gloo Shot**[21]. I love the tools coming from solo.io[22]. I like Gloo, and I like their service mesh. But, at least at the time of this writing (March 2020), Gloo Shot is relatively new, and it works only on the service mesh level. So, it's also not a good choice.

Finally, we have **Chaos Toolkit**[23]. It is very well documented, and you should have not problems finding all the information you need. It has quite a few modules that can significantly extend its capabilities. We can use it with or without Kubernetes. We can run experiments against GCP, AWS,

[15] https://github.com/netflix/chaosmonkey
[16] https://github.com/Netflix/SimianArmy
[17] https://www.gremlin.com/
[18] https://github.com/bloomberg/powerfulseal/blob/master/README.md
[19] https://github.com/asobti/kube-monkey
[20] https://litmuschaos.io/
[21] https://glooshot.solo.io
[22] https://www.solo.io/
[23] https://chaostoolkit.org/

Azure, service mesh (Istio in particular), etc. It has a very active community. If you go to their Slack workspace[24], you will see that there are quite a few folks who will be happy to help you out. Even though the project is not fully there, I think it's the best choice we have today (March 2020), at least among those I mentioned.

We are going to choose Chaos Toolkit because it's the only one that is open-source and that it works both inside and outside Kubernetes. It has decent documentation, and its community is always willing to help. That does not mean that other tools are bad. They're not. But we have to make a choice, and that choice is Chaos Toolkit.

Remember, the goal of this book, and of almost everything I do, is to teach you how to think and the principles behind something, rather than how to use a specific tool. Tools are a means to an end. The goal should rarely be to master a tool but to understand the processes and the principles behind it.

By the end of this book, you will, hopefully, be chaos engineering ninja, and you will know how to use Chaos Toolkit. If you do choose to use some other tool, you will be able to translate the knowledge because the principles behind chaos engineering are the same no matter which tool you choose. You should be able to adapt to any tool easily. Nevertheless, I have a suspicion that you will like Chaos Toolkit and that you'll find it very useful.

[24] https://join.chaostoolkit.org/

Defining Course Requirements

Before we jump into practical exercises and start creating chaos, there are a few requirements that you should be aware of.

You will need Git[25]. I'm sure that you have it. Everyone does. If you don't, then change the profession to something else.

If you're a **Windows user**, I strongly recommend that you run all the commands from a Git Bash terminal. It is available through the Git setup. If you don't have it already, I recommend that you re-run Git installation. You'll see a checkbox that, when selected, will install Git Bash. If you have something against it, any other Bash should do. You could, theoretically, run your commands in PowerShell, or Windows terminal, but then there would be some differences when compared with the commands in this book. So, if you're a Windows user, and to have parity with Mac and Linux users, I strongly recommend that you run the commands from Git Bash or any other Bash terminal.

Since we're going to work with Kubernetes, you will need **kubectl**[26] as well as a Kubernetes cluster. It can be **Docker Desktop**, **Minikube**, **GKE**, **EKS**, **AKS**. I will provide instructions on how to create those. It could also be any other distribution. However, bear in mind that I tested only those five, even though everything should work in any other Kubernetes flavor, as long as you're aware that you might need to modify some commands every once in a while.

While we're at the subject of Kubernetes distributions, if you do prefer **Minikube** or **Docker Desktop**, please be aware that few of the examples that are dealing with nodes will not work. In those cases, you'll need to observe the outputs from the book to understand what's happening. The reason for that lies in the fact that Minikube and Docker Desktop are single-node clusters, with both the worker node and the control plane being mixed together. As you can probably guess, destroying the only node is not something we can recuperate from. All in all, most of the things will work in local Kubernetes distributions (Minikube or Docker Desktop), except the parts dealing with nodes.

If you're using **EKS**, you will need **Helm v3.x**. At one moment, we will set up Cluster Autoscaler, which is probably easiest to deploy with Helm.

Soon, we will install **Chaos Toolkit**, which requires **Python v3.x** and **pip**. The latter is typically installed during Python setup.

Other than those requirements, I might ask you to install other things throughout the book. But for now, those requirements are all you need. If you're already working with Kubernetes (as I hope you do), you should already have all those executables in your laptop. The exception might be Python.

Every chapter, and this one is no exception, has an associated Gist with all the commands. That way, you can be lazy and copy and paste them instead of staring at the book and typing them in your terminal.

[25] https://git-scm.com/
[26] https://kubernetes.io/docs/tasks/tools/install-kubectl/

All the commands from this chapter are available in the 02-setup.sh[27] Gist.

The full list of the requirements is as follows.

- Git[28]
- GitBash (if Windows)
- kubectl[29]
- A Docker Desktop, Minikube, GKE, EKS, AKS, or any other Kubernetes cluster (not tested)
- Helm v3.x[30] (if EKS)
- Python v3.x[31]
- pip[32]

With that out of the way, we can proceed and install Chaos Toolkit.

Installing Chaos Toolkit

Installing Chaos Toolkit is easy, fast, and straightforward, assuming that you have Python v3.x and `pip` installed.

So let's install it.

```
1 pip install -U chaostoolkit
```

Now, to be on the safe side, let's double-check that Chaos Toolkit was installed correctly by outputting help.

```
1 chaos --help
```

The output should contain the `usage`, the `options`, and the `commands`. If it does, we are good to go. We have Chaos Toolkit installed, and we can start destroying things.

[27] https://gist.github.com/37b4c10aca0d1214965506a146fc3488
[28] https://git-scm.com/
[29] https://kubernetes.io/docs/tasks/tools/install-kubectl/
[30] https://helm.sh/docs/intro/install/
[31] https://www.python.org/downloads
[32] https://pip.pypa.io/en/stable/installing

Destroying Application Instances

I believe that was just enough of theory. From now on, everything we'll do will be hands-on.

So, what are we going to start with? What will be our first set of chaos experiments?

I don't want to stress you too much from the start, so we'll start easy and increase the complexity as we're progressing.

The easiest thing we'll start exploring is how to terminate instances of our applications. We'll learn how to delete, remove, and kill instances of our applications. That will provide us with quite a few things. To begin with, it will give us a solid base of how chaos experiments work. It will give us also some side effects. It will hopefully show us the importance of scaling, fault tolerance, high availability, and a few other things.

So, what are we going to do in this section? To begin with, we are going to create a Kubernetes cluster. After all, this is about chaos engineering in Kubernetes. So, we'll need a Kubernetes cluster. As I mentioned before, you can use Minikube, Docker Desktop, GKE, EKS, or AKS. With some small tweaks, any other cluster should do.

Then, we are going to deploy an application. We will need an app that we will use for demo purposes. It'll be a simple one. The goal is not to develop a complex application, but to deploy a simple one, and to try experimenting on top of it.

Before we start running chaos experiments, we'll explore the Kubernetes plugin for Chaos Toolkit. With it, we'll be able to start terminating instances or, to be more precise, replicas of a demo application.

We are going to learn about the steady-state hypothesis, and we might need to add some tweaks to our scripts. We might need to explore how to pause things. In some cases, it might not be the best idea to go as fast as we can. We will learn how to put some conditions and how to define phases to our hypotheses. We will try to observe what we learn from those experiments, and those learnings might lead us towards making our application fault-tolerant. Once we're finished with all that, and a few other things, we are going to destroy what we created.

Every section is going to end with the destruction of everything so that you can start the next chapter from a clean slate. That will allow you to take a break. It might be an hour break, or maybe daybreak, or even a full week. In any case, you don't want to keep your cluster running while on the break. That would only incur unnecessary costs.

How does that sound?

Let's get going. Let's create our Kubernetes cluster.

Gist With Commands

From now on, everything we do will be hands-on, and I will run and execute a lot of commands. We might be changing the contents of some files, and we might do some light coding. We'll see.

I expect you to follow all the exercises, and there are two general ways how you can do that. You can read and type all the commands, or you might copy and paste. And if you prefer the latter, I will provide Gists for this chapter, and for all those that follow. You can use it to copy and paste. Trust me, that's easier than reading the commands from a book, and typing them into your terminal.

All the commands from this chapter are available in the 03-terminating-pods.sh[33] Gist.

Creating A Cluster

Here comes the first task for you. Get a Kubernetes cluster. If you already have one, whichever it is, you can use it for the exercises. If you do not have a Kubernetes cluster, and you know how to create one, just go and create it. But, if you do not have a Kubernetes cluster and you want me to help you create one, then the Gists with the commands for creating a cluster are available below.

You can choose between **Minikube**, **Docker Desktop**, **AKS**, **EKS**, **GKE**. In other words, the most commonly used Kubernetes flavors are available as Gists. If you don't like any of them, create your own cluster. As long as you have one, you should be fine.

If you follow my commands to create a cluster, you will see that they're all a single zone clusters, at least those in Google, AWS, and Azure. They're running in a single zone, and they have a single node outside the control plane. It's a very simple cluster, and very small. It's just big enough for what we need. We'll see later whether those are well-defined clusters. Chaos experiments will help us with that.

All in all, use my instructions to create a cluster, or create it any other way you like, or use an existing cluster. It's up to you.

Gists with the commands to create and destroy a Kubernetes cluster are as follows.

- Docker Desktop: docker.sh[34]
- Minikube: minikube.sh[35]
- GKE: gke.sh[36]
- EKS: eks.sh[37]

[33] https://gist.github.com/419032bc714cc31cd2f72d45ebef07c7
[34] https://gist.github.com/f753c0093a0893a1459da663949df618
[35] https://gist.github.com/ddc923c137cd48e18a04d98b5913f64b
[36] https://gist.github.com/2351032b5031ba3420d2fb9a1c2abd7e
[37] https://gist.github.com/be32717b225891b69da2605a3123bb33

- AKS: aks.sh[38]

There's one more task left for us to do before we start destroying things. We'll need to deploy a demo application.

Deploying The Application

It would be hard for us to start destroying instances of an application if we do not have any. So the first thing we're going to do, now that we have a cluster, is to deploy a demo application. I've already prepared one that you can use.

We're going to clone my repository `vfarcic/go-demo-8` that contains a very simple demo application. It contains just enough complexity or simplicity, depending on how you look at it. We'll use it for all our chaos experiments.

So let's clone the application repository.

```
1 git clone \
2     https://github.com/vfarcic/go-demo-8.git
```

Next, we'll enter into the directory where the repository was cloned. And, in case you already have that repository from some other exercises you did with me, we're going to pull the latest version.

```
1 cd go-demo-8
2
3 git pull
```

It shouldn't matter whether you cloned the repo for the first time, or you entered into a copy of the existing repo and pulled the code. In any case, you must always have the latest version because I might have changed something.

Everything in this book will be created in separate namespaces so that the rest of your cluster is not affected (much). With that in mind, we are going to create a namespace called `go-demo-8`.

```
1 kubectl create namespace go-demo-8
```

Next, we're going to take a quick look at the definition of the application we're going to deploy. It is located in the `terminate-pods` directory, in a file called `pod.yaml`.

[38] https://gist.github.com/c7c9a8603c560eaf88d28db16b14768c

```
cat k8s/terminate-pods/pod.yaml
```

The output is as follows.

```
---

apiVersion: v1
kind: Pod
metadata:
  name: go-demo-8
  labels:
    app: go-demo-8
spec:
  containers:
  - name: go-demo-8
    image: vfarcic/go-demo-8:0.0.1
    env:
    - name: DB
      value: go-demo-8-db
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /
        port: 8080
    readinessProbe:
      httpGet:
        path: /
        port: 8080
    resources:
        limits:
          cpu: 100m
          memory: 50Mi
        requests:
          cpu: 50m
          memory: 20Mi
```

As you can see, this is a very simple definition of an application. It is an app defined as a single Pod that has only one container called `go-demo-8`. It is based on my image `vfarcic/go-demo-8`. It has an environment variable, but that is not really important. Finally, there are `livenessProbe`, `readinessProbe`, and some resources. If you have any familiarity with Kubernetes, this is as simple as it can get. It's a definition of a Pod.

If you're not familiar with Kubernetes, then this might be too advanced, and you might want to check some other resources and get familiar with Kubernetes first. I can recommend my book, The DevOps Toolkit 2.3: Kubernetes[39], but any other resource should do. What matters is that I will assume that you have at least basic knowledge of Kubernetes. If not, stop reading this, and learn at least fundamentals.

Now that we saw the definition of the application in the form of a single Pod, we are going to apply that definition to our cluster inside the `go-demo-8` Namespace.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/terminate-pods/pod.yaml
```

And that's about it. We have our application up and running as a Pod.

Now, let's see how we can destroy it. What damage can we do to that application? Let's be destructive and figure out how we can improve things through chaos.

Discovering Chaos Toolkit Kubernetes Plugin

There is one crucial thing you should know about Chaos Toolkit. It does not support Kubernetes out-of-the-box, and that's not the only thing it does not provide initially. There are many others that we will need. But, luckily for us, Chaos Toolkit itself is very basic because it is based on the plugins (or modules) ecosystem.

For anything beyond basic out-of-the-box features, we need to install a plugin. And since our primary interest right now is in Kubernetes, we are going to install a Kubernetes plugin, and we will do that using `pip`.

```
pip install -U chaostoolkit-kubernetes
```

In case you already have the plugin, `-U` argument made sure that it was updated. And if you don't have it, we just installed it.

Now we should be able to create our first experiment. But how are we going to use the plugin if we don't know what it provides?

Discovering what a Chaos Toolkit plugin can do is relatively easy through the `discover` command that allows us to see all the features, all the options, and all the arguments of a plugin.

```
chaos discover chaostoolkit-kubernetes
```

The result is stored in a `discovery.json` file, which we can output. Let's see what's inside.

[39] https://amzn.to/2GvzDjy

```
cat discovery.json
```

The output is too big to be presented in a book. You should see it on your screen anyway, assuming that you are indeed executing the commands I'm providing.

What you see is JSON representation of all the actions and probes and what so not. It contains everything that can be done through this plugin, and I encourage you to go through all of it right now. Take a break from this book and explore everything that the plugin offers. Think of this as being full documentation of everything you can do through that specific plugin.

Don't worry if you get confused. Don't worry if you don't understand everything that's in there. We're going to go through all those things one by one soon. However, don't take that as an excuse to avoid going through the `discover` output and to avoid trying to figure out at least the topics and the types of the things that we can do by using this plugin. As I said before, don't get scared. I will explain everything soon.

Terminating Application Instances

We're finally there. We can finally create some chaos. We are about to destroy stuff.

Let's take a look at the first definition that we are going to use. It is located in the `chaos` directory, in the file `terminate-pod.yaml`.

```
cat chaos/terminate-pod.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we terminate a Pod?
description: If a Pod is terminated, a new one should be created in its places.
tags:
- k8s
- pod
method:
- type: action
  name: terminate-pod
  provider:
    type: python
    module: chaosk8s.pod.actions
    func: terminate_pods
    arguments:
      label_selector: app=go-demo-8
      rand: true
      ns: go-demo-8
```

What do we have there? We have `version`, `title`, `description`, and `tags`. They're all informative. They do not influence how an experiment is executed. They just provide additional information to whoever would like to know what that experiment is about. The `title`, for example, says `what happens if we terminate a Pod?`. That's a valid question. And the `description` is a statement that `if a pod is terminated, a new one should be created in its place`. We also have tags that are, again, optional. They're letting us know that the experiment is about `k8s` (Kubernetes), and it is about a `pod`.

The real action is happening in the `method`. That section specifies the activities that we want to perform. They can be actions or probes. For now, we're going to focus only on actions.

We have only one `action` that we're going to perform. We're calling it `terminate-pod`. Within that `action`, there is a `provider` that is of type `python`. There could be theoretically other types, but since Chaos Toolkit uses Python, in most cases, the `type` will be `python`.

The `module` is matching the plugin that we installed before (`chaosk8s`). We're also specifying that we want to do something with a `pod` and that something is `actions`. Hence, the full name of the module is `chaosk8s.pod.actions`.

The main thing is the function (`func`). Whenever you're in doubt, consult the output of the `discover` command, and you'll see types of functions you can use within that module. In our case, we're going to use the function called `terminate_pods`.

In this, and in most other cases, functions require some arguments. We're using `label_selector` that specified that we're going to terminate a Pod that matches the `app=go-demo-8` label. It could be any other label as long as there is at least one pod matches it.

We also have the argument `rand`, short for random, set to `true`. If more Pods are matching that `label_selector`, a random one will be terminated.

And finally, there is the `ns` argument that specifies the Namespace.

All in all, we want to terminate a Pod in the `go-demo-8` Namespace that has the matching label `app=go-demo-8`. And if there is more than one Pod with those criteria, it should select a random one to destroy.

Now that we saw the definition, we can run it and see what we'll get.

```
1 chaos run chaos/terminate-pod.yaml
```

The output is as follows (timestamps are removed for brevity).

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate a Pod?
[... INFO] No steady state hypothesis defined. That's ok, just exploring.
[... INFO] Action: terminate-pod
[... INFO] No steady state hypothesis defined. That's ok, just exploring.
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: completed
```

We can see from the output that there are certain events. They started by validating that the syntax is correct.

After the initial validation, it started running the experiment called `What happens if we terminate a Pod?`. It found that there is `no steady state hypothesis defined`. We're going to see what steady state hypothesis is. For now, just observe that there isn't any.

Judging by the output, there is one action `terminate-pod`. We'll see the effect of that action later, even though you can probably guess what it is.

Further on, it went back to the `steady state hypothesis` and figured out there is none. Then it tried to `rollback`, and it found out that it couldn't. We did not specify any of those things. We did not set the `steady state hypothesis`, and we did not specify how to `rollback`. All we did so far was to execute an action to terminate a Pod. We can see the result in the last line that says that the `experiment ended with status: complete`. It's finished successfully.

That was the simplest experiment I could come up with, even though it might not be useful in its current form.

Now, let's output the exit code of the previous command.

```
echo $?
```

We can see that the output is `0`. If you're not familiar with Linux, exit code `0` means success. And any other number would indicate a failure.

Why are we outputting the exit code? The reason is simple. Even though we're running the experiments manually and we can see from the output whether they are successful or not, sooner or later, we will want to automate all that. And it is essential that whatever we execute returns proper exit codes. That way, no matter the mechanism we use to run experiments, those exit codes will tell the system whether it's a failure or a success.

Now, let's take a look at the Pods in our Namespace.

```
kubectl --namespace go-demo-8 \
    get pods
```

The output states that `no resources` were `found in go-demo-8 namespace.`

What happened is that we deployed the single Pod, and we run an experiment that destroyed it. That experiment by itself is not that useful because we did not validate whether the state before we run it was correct, nor we checked whether the state after the action was as expected. We did not do any validations. Instead, we only executed a single action meant to terminate a Pod, and we can see from the output that Pod is no more. It was destroyed.

Defining The Steady-State Hypothesis

What we did so far cannot be qualified as an experiment. We just executed an action that resulted in the destruction of a Pod. And the best we can get from that is a satisfying feeling like *"oh, look at me, I destroyed stuff."* But the goal of chaos engineering is not to destroy for the sake of feeling better or for the purpose of destruction itself. The objective is to find weak points in our cluster, in our applications, in our data center, and in other parts of our system. To do that, typically, we start by defining a steady-state that is validated before and after actions.

We define something like *this is how that should look like*. If the state of that something is indeed as we defined it, we start destroying stuff. We introduce some chaos into our cluster. After that, we take another look at the state and check whether it is still the same.

So if the state is the same both before and after actions, we can conclude that our cluster is fault-tolerant, that it is resilient, and that everything is just peachy. In the case of Chaos Toolkit, we accomplish that by defining `steady state hypothesis`.

We're going to take a look at yet another definition that specifies the state that will be validated before and after some actions. Let's take a look.

```
cat chaos/terminate-pod-ssh.yaml
```

Since it is difficult to see the differences from the file itself, we're going to output the `diff` and see what was added compared to what we had before.

```
diff chaos/terminate-pod.yaml \
    chaos/terminate-pod-ssh.yaml
```

The output is as follows.

```
> steady-state-hypothesis:
>   title: Pod exists
>   probes:
>   - name: pod-exists
>     type: probe
>     tolerance: 1
>     provider:
>       type: python
>       func: count_pods
>       module: chaosk8s.pod.probes
>       arguments:
>         label_selector: app=go-demo-8
>         ns: go-demo-8
```

The new section, as you can see, is `steady-state-hypothesis`.

It has a `title`, which is just informative, and it has some `probes`. In this case, there's only one, but there can be more.

The name of the `probe` is `pod-exists`. Just like the `title`, it is informative and serves no practical purpose.

The `type` is `probe`, meaning that it will probe the system. It will validate whether the result is acceptable or not, and, as you will see soon, it will do that before and after actions.

The `tolerance` is set `1`. We'll come back to it later.

Further on, we can see that we have the `provider` set to `python`. Get used to it. Almost all providers are based on Python.

The function is `count_pods`, so, as the name implies, we're going to count how many Pods we have that match that criteria.

The module is `chaosk8s.pods.probes`. It comes from the same plugin we installed before.

Finally, we have two `arguments`. The first one will select only Pods that have the matching label `app=go-demo-8`. The second is `ns`, short for Namespace. When combined, it means that the `probe` will count only the Pods with the matching label `app=go-demo-8` and in the Namespace `go-demo-8`.

Now, if we go back to the `tolerance` argument, we can see that it is set to `1`. So, our experiment will expect to find precisely one Pod with the matching label and in the specified Namespace. Not more, not less.

Let's run this chaos experiment and see what we're getting.

```
chaos run chaos/terminate-pod-ssh.yaml
```

The output is as follows (timestamps are removed for brevity).

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate a Pod?
[... INFO] Steady state hypothesis: Pod exists
[... INFO] Probe: pod-exists
[... CRITICAL] Steady state probe 'pod-exists' is not in the given tolerance so fail\
ing this experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: failed
```

We can see that there is a critical issue. `Steady state probe 'pod-exists' is not in the given tolerance`. That probe failed before we even started executing actions like those that we had before. That's normal because we destroyed the Pod in the previous section. Now there are no Pods in that Namespace, at least not those with that matching label.

So our experiment failed at the very beginning. It confirmed that the initial state of what we are measuring is not matching what we want it to be. The experiment failed, and we can see that by outputting the exit code of the previous command.

```
echo $?
```

This time the output is `1`, meaning that the experiment was indeed unsuccessful. It failed at the very beginning before it even tried to execute actions. It attempted to validate the initial state, which expects to have a single Pod with the matching label. The experiment failed in that first validation. The Pod is not there. The previous experiment terminated it, and we did not recreate it.

Now, let's apply the `terminate-pods/pod.yaml` definition so that we recreate the Pod we destroyed with the first experiment. After that, we'll be able to see what happens if we re-run the experiment with the `steady-state-hypothesis`.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/terminate-pods/pod.yaml
```

We re-created our pod, and now we're going to re-run the same experiment.

```
chaos run chaos/terminate-pod-ssh.yaml
```

The output is as follows (timestamps are removed for brevity).

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate a Pod?
[... INFO] Steady state hypothesis: Pod exists
[... INFO] Probe: pod-exists
[... INFO] Steady state hypothesis is met!
[... INFO] Action: terminate-pod
[... INFO] Steady state hypothesis: Pod exists
[... INFO] Probe: pod-exists
[... INFO] Steady state hypothesis is met!
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: completed
```

This time everything is green. We can see that the probe `pod-exists` confirmed that the state is correct, and we can see that the action `terminate-pod` was executed. Further on, we can observe that the steady state was re-evaluated. It confirmed that the Pod still exists. The Pod existed before the action, and Pod existed after the action, and, therefore, everything is green.

Now, that is kind of strange, isn't it? We confirmed that the Pod exists, and then we destroyed it. Nevertheless, the experiment shows that the Pod still exists or, to be more precise, that it existed after that action that removed it. How can the Pod exist if we destroyed it?

Before we discuss that, let's confirm that the experiment was really successful by outputting `$?`.

```
echo $?
```

We can see that the exit code is indeed `0`. That's indeed strange. The probe should have failed after the action that destroyed the Pod. The reason why it didn't fail will be explained soon. For now, let's recreate that Pod again.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/terminate-pods/pod.yaml
```

Next, we're going to try to figure out why our experiment didn't fail. It's wasn't supposed to be successful.

Pausing After Actions

In our previous experiment, we validated the state before and after actions. We checked whether the Pod exists, then we terminate the Pod, and then we verified whether the Pod still exists. And the experiment should have failed, but it didn't. The reason why it didn't fail is that all those probes and actions were executed immediately one after another.

When Chaos Toolkit sent an instruction to Kube API to destroy the Pod, it received an acknowledgment of that action, and immediately after that, it validated whether the Pod was still there. And it was. Kubernetes did not have enough time to remove it entirely. Maybe the Pod was still running at that time, and perhaps we were too fast. Or, maybe the Pod was terminating. That would probably explain that strange outcome. A Pod was not gone right away. It was still there, with the state terminating. So the thing that we are probably missing to make that experiment more useful is a pause.

Let's see how we can pause after or before actions and give enough time for the system to perform whichever tasks it needs to perform before we validate the state again.

We're going to take a look at yet another yaml.

```
cat chaos/terminate-pod-pause.yaml
```

We will almost always output the diffs between the new and the old definition as a way to easily spot the differences.

```
diff chaos/terminate-pod-ssh.yaml \
    chaos/terminate-pod-pause.yaml
```

The output is as follows.

```
>   pauses:
>     after: 10
```

This time, we can see that the only addition is to add the `pauses` section. It is placed after the `action` that terminates the Pod, and it has the `after` section set to `10`. Pauses can be added `before` or `after` some actions. In this case, when we execute the action to terminate the Pod, the system will wait for 10 seconds, and then it will validate the state again.

So, let's see what we'll get if we execute this experiment.

```
chaos run chaos/terminate-pod-pause.yaml
```

The output is as follows (timestamps are removed for brevity).

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate a Pod?
[... INFO] Steady state hypothesis: Pod exists
[... INFO] Probe: pod-exists
[... INFO] Steady state hypothesis is met!
[... INFO] Action: terminate-pod
[... INFO] Pausing after activity for 10s...
[... INFO] Steady state hypothesis: Pod exists
[... INFO] Probe: pod-exists
[... CRITICAL] Steady state probe 'pod-exists' is not in the given tolerance so fail\
ing this experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: deviated
[... INFO] The steady-state has deviated, a weakness may have been discovered
```

We can see that it paused for 10 seconds, and then the probe failed. It said that `steady state probe 'pod-exists' is not in the given tolerance so failing this experiment`.

With enough pause, we managed to make a more realistic experiment. We gave Kubernetes enough time to remove that Pod entirely, and then we validated whether the Pod is still there. The system came back to us, saying that it isn't. And we can confirm that by outputting the exit code of the last command.

```
echo $?
```

We can see that the output is `1`, meaning that the experiment indeed failed.

Now before we move forward and explore a few other essential things, we're going to recreate that failed Pod before we add a few more missing pieces that would make this experiment really valid.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/terminate-pods/pod.yaml
```

Probing Phases And Conditions

We are most likely not really whether a Pods exist. That does not really serve much of a purpose. Instead, we should validate whether the Pods are healthy. Knowing that a Pod exists would not really do us much good if it is unhealthy or if the application inside it is hanging. What we really want to validate is whether the Pods exist, and whether that they are in a certain phase.

But before we go through that, let's describe the Pod that we created and see which are the types and phases and conditions it's in.

```
kubectl --namespace go-demo-8 \
    describe pod go-demo-8
```

The output, limited to the relevant parts, is as follows.

```
...
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
...
```

If we take a closer look at the `conditions` section, we can see that there are two columns; the `type` and the `status`. We can see that this Pod is `initialized`, but that it is not yet `Ready`. We can see that there are different conditions of that Pod that we might want to take into account when constructing our experiments. And what is even more important is that, so far, we did not really check the status of our Pod, neither through experiments nor with `kubectl`. Maybe that Pod was never really running. Perhaps it was always failing, no matter the experiments that we were running. And that's entirely possible because our experiments were not validating the readiness and the state of our Pod. They were just checking whether it exists. Maybe it did exist, but it was never functional.

Let's validate that. Let's introduce some additional arguments to our experiment.

We'll take a look at a new YAML definition.

```
cat chaos/terminate-pod-phase.yaml
```

What matters more is the `diff`.

```
diff chaos/terminate-pod-pause.yaml \
    chaos/terminate-pod-phase.yaml
```

The output is as follows.

```
> - name: pod-in-phase
>   type: probe
>   tolerance: true
>   provider:
>     type: python
>     func: pods_in_phase
>     module: chaosk8s.pod.probes
>     arguments:
>       label_selector: app=go-demo-8
>       ns: go-demo-8
>       phase: Running
> - name: pod-in-conditions
>   type: probe
>   tolerance: true
>   provider:
>     type: python
>     func: pods_in_conditions
>     module: chaosk8s.pod.probes
>     arguments:
>       label_selector: app=go-demo-8
>       ns: go-demo-8
>       conditions:
>       - type: Ready
>         status: "True"
```

We added two new probes. The tolerance for the first one (`pod-in-phase`) is set to `true`. So we expect the output of whatever we're going to probe to return that value. This time, the function is `pods_in_phase`. It's yet another function available in the module `chaosk8s.pod.probes`.

Next, we have a couple of arguments. The `label_selector` is set to `app=go-demo-8`, and `ns` is set to `go-demo-8`. Those are the same arguments we used before, but this time we have a new one called `phase`, which is set to `Running`. We are validating whether there is at least one Pod with the matching label, inside a specific Namespace, and in the phase `Running`.

The second probe (`pod-in-conditions`) is similar to the first one, except that it uses the function `pods_in_conditions`. That's another function available out of the box in the Kubernetes plugin. The arguments the same; `label_selector` and `ns`. The new thing here is the `conditions` section. In it, we are specifying that the `type` of condition should be `Ready` and the status should be `"True"`.

All in all, we are adding two new probes. The first one is going to confirm that our Pod is indeed running, and the second will check whether it is ready to serve requests.

Let's see what we get when we execute this experiment.

```
chaos run chaos/terminate-pod-phase.yaml
```

The output is as follows (timestamps are removed for brevity).

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate a Pod?
[... INFO] Steady state hypothesis: Pod exists
[... INFO] Probe: pod-exists
[... INFO] Probe: pod-in-phase
[... INFO] Probe: pod-in-conditions
[... ERROR]   => failed: chaoslib.exceptions.ActivityFailed: pod go-demo-8 does not \
match the following given condition: {'type': 'Ready', 'status': 'True'}
[... WARNING] Probe terminated unexpectedly, so its tolerance could not be validated
[... CRITICAL] Steady state probe 'pod-in-conditions' is not in the given tolerance \
so failing this experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: failed
```

The experiment failed. It failed to execute one of the probes before it even started executing actions. Something is wrong. The state of our Pod is not correct. It probably never was.

We have the `ActivityFailed` exception saying that it `does not match the following given condition: {'type': 'Ready', 'status': 'True'}`. Our Pod is running. We know that because the previous probe (`pod-in-phase`), is checking whether it is running. So, Pod is indeed running, but the condition of the Pod is not `ready`.

Now we know that there was something wrong with our Pod from the very beginning. We had no idea until now because we did not really examine the Pod itself. We just trusted that its existence is enough. "Oh, I created a Pod. Therefore, the Pod must be running."

We can see from the experiment that the Pod is not running. Or, to be more precise, it is running, but it is not ready. Everything we did so far was useless because the initial state was never really well defined.

Just as before, we are going to validate that the status of the experiment is indeed indicating an error. We'll do that by retrieving the exit code of the last command.

```
echo $?
```

The output is `1`, so that's fine, or, to be more precise, it is as we expected it to be.

Now let's take a look at the logs of the Pod. Why is it failing, and why didn't we notice that there is something wrong with it from the start?

```
kubectl --namespace go-demo-8 \
    logs go-demo-8
```

The output is as follows.

```
2020/01/27 20:50:00 Starting the application
2020/01/27 20:50:00 Configuring DB go-demo-8-db
panic: no reachable servers

goroutine 1 [running]:
main.setupDb()
        /Users/vfarcic/code/go-demo-8/main.go:74 +0x32e
main.main()
        /Users/vfarcic/code/go-demo-8/main.go:52 +0x78
```

If you read one of my other books, you probably recognize that output. If you haven't, I won't bore you with the architecture of the application. Instead, I'll just say that the app uses MongoDB, which we did not deploy. It tries to connect to it, and it fails. As a result, the container fails, and it is recreated by Kubernetes. Continuously. For eternity. Or, at least, until we fix it.

From the very beginning, the Pod was failing because it was trying to connect to the database, which does not exist. We just discovered that by running an experiment probe that failed in its initial set of validations.

Now that we know that the Pod was never really running because a database is missing, we're going to fix that by deploying MongoDB. I will not go through the definition of the DB because it's not relevant. The only thing that matters is that we're going to deploy it to fulfill the requirement of the application. Hopefully, that will fix the issue, and the Pod will be running this time.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/db
```

Next, we'll wait until the database is rolled out.

```
kubectl --namespace go-demo-8 \
    rollout status \
    deployment go-demo-8-db
```

Now that the database is rolled out, we're going to take a look at the Pods.

```
kubectl --namespace go-demo-8 \
    get pods
```

The output is as follows.

```
NAME             READY STATUS  RESTARTS AGE
go-demo-8        1/1   Running 21       66m
go-demo-8-db-... 1/1   Running 0        7m17s
```

If, in your case, the `go-demo-8` Pod is not yet `Running`, please wait for a while. The next time it restarts, it should be working because it will be able to connect to the database.

The `go-demo-8` Pod is now ready and running after restarting for quite a few times (in my case, `21` times).

Please note that this is the first time we are retrieving Pods. We can see that `go-demo-8` is now running. We would have saved ourselves from a lot of pain if we retrieved the Pods earlier. But, while that would help us in this "demo" environment, we cannot expect to validate things manually in a potentially large production environment. That's why we have tests, but that's not the subject of this book. And even if we don't have tests, now we have chaos experiments to validate the desired state both before and after destructive actions.

Our application is now fully operational.

Do you think that our chaos experiment will pass now? What do you think will be the result when we re-run the same experiment?

Let's see.

```
chaos run chaos/terminate-pod-phase.yaml
```

The output is as follows (timestamps are removed for brevity).

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate a Pod?
[... INFO] Steady state hypothesis: Pod exists
[... INFO] Probe: pod-exists
[... INFO] Probe: pod-in-phase
[... INFO] Probe: pod-in-conditions
[... INFO] Steady state hypothesis is met!
[... INFO] Action: terminate-pod
[... INFO] Pausing after activity for 10s...
```

```
[... INFO] Steady state hypothesis: Pod exists
[... INFO] Probe: pod-exists
[... INFO] Probe: pod-in-phase
[... INFO] Probe: pod-in-conditions
[... ERROR]   => failed: chaoslib.exceptions.ActivityFailed: pod go-demo-8 does not \
match the following given condition: {'type': 'Ready', 'status': 'True'}
[... WARNING] Probe terminated unexpectedly, so its tolerance could not be validated
[... CRITICAL] Steady state probe 'pod-in-conditions' is not in the given tolerance \
so failing this experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: deviated
[... INFO] The steady-state has deviated, a weakness may have been discovered
```

To begin with, all three probes executed before the actions are passing, so the Pod did exist when the experiment started. It was running and was is ready. And then, the action to terminate the Pod was executed. After that, and the pause of `10` seconds, the probe failed.

So we managed to confirm that, this time, the Pod is really in the correct state. But after we terminate the Pod, the first probe failed. If that one didn't fail, the second one would fail, and the third one would fail as well. Basically, all three probes are guaranteed to fail after we destroy the only Pod we have with the matching labels and in that Namespace. However, Chaos Toolkit did not execute all the probes. It failed fast. The execution of the experiment stopped after the first failure.

Just as before, we'll retrieve the last exit code, and confirm that Chaos Toolkit indeed exits correctly.

```
echo $?
```

We can see the output is `1`, meaning that the experiment failed.

We are moving forward. We're improving our experiments. However, while our experiment is now relatively good, our application is not. It is not fault-tolerant, and we're going to correct that next.

Making The Application Fault-Tolerant

Now we have a decent experiment that is validating whether our Pod is in the correct state, whether it is under the right conditions, and whether it exists. And then it destroys the Pod, only to repeat the same validations of its state. The experiment confirms something that you should already know. Deploying a Pod by itself does not make it fault-tolerant. Pods alone are not even highly available. They're not many things. But what interests us, for now, is that they are not fault-tolerant. And we confirmed that through the experiment.

We did something that we shouldn't do. We deployed a Pod directly, instead of using some higher-level constructs in Kubernetes. That was intentional. I wanted to start with a few simple experiments

and to progress over time towards more complex ones. I promise that I will not make the same ridiculous definitions of the application in the future.

Let's correct the situation.

We know through the experiment that our application is not fault-tolerant. When an instance of our app is destroyed, Kubernetes does not recreate a new one. So how can we fix that? If you have any experience with Kubernetes, you should already know that we are going to make our application fault-tolerant by creating a Deployment instead of a Pod.

Let's take a look at a new definition.

```
cat k8s/terminate-pods/deployment.yaml
```

The output is as follows.

```
---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: go-demo-8
  labels:
    app: go-demo-8
spec:
  selector:
    matchLabels:
      app: go-demo-8
  template:
    metadata:
      labels:
        app: go-demo-8
    spec:
      containers:
      - name: go-demo-8
        image: vfarcic/go-demo-8:0.0.1
        env:
        - name: DB
          value: go-demo-8-db
        - name: VERSION
          value: "0.0.1"
        ports:
        - containerPort: 8080
        livenessProbe:
```

```
29            httpGet:
30              path: /
31              port: 8080
32          readinessProbe:
33            httpGet:
34              path: /
35              port: 8080
36          resources:
37              limits:
38                cpu: 100m
39                memory: 50Mi
40              requests:
41                cpu: 50m
42                memory: 20Mi
```

The new definition is a Deployment with the name `go-demo-8`. It contains a single container defined in almost the same way as the Pod we defined before. The major difference is that we know that destroying our instances created as Pods will do us no good. You surely knew that already, but soon we'll have a "proof" that's really true.

So, we are changing our strategy, and we're going to create a Deployment. Since you are familiar with Kubernetes, you know that a Deployment creates a ReplicaSet and that ReplicaSet creates one or more Pods. More importantly, you know that ReplicaSet's job is to ensure that a specific number of Pods is always running.

Let's apply that definition.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/terminate-pods/deployment.yaml
```

Next, we're going to wait until the Pods of that Deployment rollout.

```
kubectl --namespace go-demo-8 \
    rollout status \
    deployment go-demo-8
```

Finally, we're going to execute the same chaos experiment we executed before.

```
chaos run chaos/terminate-pod-phase.yaml
```

We can see that, this time, everything is green. At the very beginning, all three probes passed. Then, the action to terminate the pod was executed, and we waited for `10` seconds. After that, the experiment executed probes again and validated that the state met the requirements, just as it did before executing the actions. Everything worked.

With that experiment, we can confirm that if we destroy an instance of our application, then nothing terrible will happen. Or, at least, we know that Pods will continue running because they are now controlled by a ReplicaSet created by a Deployment.

Now that we improved our application, no matter how silly it was in the first place, we confirmed through a chaos experiment that it is indeed fault-tolerant. Bear in mind that fault-tolerant is not the same as highly available. We are yet to check that. What matters is that now we know that instances of our application will be recreated no matter how we destroy them, or what happens to them. Kubernetes, through Deployment definitions, is making sure that the specific number of instances is always running.

Destroying What We Created

That's it. We're at the end of the chapter, and we'll destroy everything we did. Bear in mind that, in this context, when I say "destroy", I don't mean run another chaos experiment, but delete the resources we created.

Every section will delete everything we created so that you can start a new chapter from a clean slate. You will be able to take a break unless you choose to continue straight away. In any case, we are going to remove the application, and we are going to do that by deleting the whole `go-demo-8` namespace.

```
cd ..

kubectl delete namespace go-demo-8
```

We went one directory back (out of the `go-demo-8` repository), and we deleted the Namespace.

Now you need to make a decision. Do you want to take a break or not? If you do want to take a break, I suggest you destroy your cluster. If you created it using one of my Gists, at the bottom are instructions on how to destroy it. That way, if you're using AWS, or Google, or Azure, you will not pay more than what you use. Even if you're using Minikube or Docker Desktop, you will not waste CPU and memory for no good reason.

So, destroy the cluster if you're going to take a break. If you want to move forward to the next section immediately, then keep the cluster and just go for it.

Experimenting With Application Availability

We saw how we can define chaos experiments that terminate our instances for the sake of validating whether our application is fault-tolerant.

Fault-tolerance is necessary, but it is insufficient for most of us. Typically, we don't want our applications to spin up after something happens to them if that means that we will have seconds or maybe even minutes of downtime. Failed Pods will indeed be re-created when they are managed by higher-level constructs like Deployments and StatefulSets. However, we might have downtime between destruction and being available again, and that's not really a good thing.

In this chapter, we'll try to define chaos experiments that will validate whether the demo application, the same one we've been using before, is highly available. As a result of those experiments, we might need to change its definition. We might need to improve it.

Let's get going and see how that works for us.

Gist With Commands

As you already know, every section that has hands-on exercises is accompanied by a Gist. That allows us to copy and paste stuff instead of typing every single command. I know that you might be lazy and that you might not want to stare at the book and type the commands in a terminal. So use the Gist that follows if you want to copy and paste commands that you'll see in this chapter.

All the commands from this chapter are available in the 04-health.sh[40] Gist.

Creating A Cluster

We need a cluster to run our experiments and to deploy our application. Whether you have one or not depends on whether you destroyed the cluster at the end of the previous chapter. If you didn't destroy it, just skip this part. If you did destroy the cluster, use the Gists that follow, or just roll out your own cluster. The choice is yours. It does not matter much how you create a Kubernetes cluster. As long as you have one, you should be fine.

Gists with the commands to create and destroy a Kubernetes cluster are as follows.

[40] https://gist.github.com/6be19a176b5cbe0261c81aefc86d516b

- Docker Desktop: docker.sh[41]
- Minikube: minikube.sh[42]
- GKE: gke.sh[43]
- EKS: eks.sh[44]
- AKS: aks.sh[45]

We're almost ready to continue destroying things. The only missing requirement is to deploy the demo application.

Deploying The Application

At the end of the previous chapter, we deleted the whole `go-demo-8` Namespace. Now we need to deploy the application again. It will be exactly the same as the last version that we deployed before. With that in mind, we should be able to go through the deployment very fast.

Let's deploy the application and go through this sub-chapter fast. Let's not waste time on that.

We are going to go to the `go-demo-8` directory, the place where we have our local repository, and we're going to pull the latest version of it just in case I made some changes.

```
1 cd go-demo-8
2
3 git pull
```

Next, we're going to create the Namespace `go-demo-8`, and then we're going to output the contents of the files in the `terminate-pods/app` directory.

```
1 kubectl create namespace go-demo-8
2
3 cat k8s/terminate-pods/app/*
```

The output of the latter command should display the same definitions as those we used in the previous chapter. So, there's probably no need to go through it. It is the same app with the database and the go-demo-8 API defined as a Deployment. Nothing more, nothing less.

So let's apply this definition and wait until it rolls out.

[41] https://gist.github.com/f753c0093a0893a1459da663949df618
[42] https://gist.github.com/ddc923c137cd48e18a04d98b5913f64b
[43] https://gist.github.com/2351032b5031ba3420d2fb9a1c2abd7e
[44] https://gist.github.com/be32717b225891b69da2605a3123bb33
[45] https://gist.github.com/c7c9a8603c560eaf88d28db16b14768c

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/terminate-pods/app

kubectl --namespace go-demo-8 \
    rollout status deployment go-demo-8
```

It might take a while until the Deployment rolls out. Be patient and wait until the execution of the latter command is finished.

That's it. The application is rolled out, and we can continue experimenting.

Later on, we'll create some chaos on the cluster level. But, for now, we will be focused on applications.

Let's see what else we can do. Can we prove that our application is highly available? Can we prove that it is not? That's a mystery that is yet to be solved.

What do you think? Do you believe that the application we just deployed is highly available? And if you think it's not, why do you think that?

Validating The Application

Before we go into more experiments, we're going to verify whether our application is accessible from outside the cluster and whether we can send requests to it. And we can do that easily by first checking whether we have an Ingress controller?

```
kubectl --namespace go-demo-8 \
    get ingress
```

We don't have it. As you almost certainly know, without Ingress, or a variation of it, our application is not accessible from outside of the cluster. So, we need to create Ingress resources. For that, we need to install Ingress in our cluster.

We're going to use nginx Ingress simply because it is the most commonly used one.

Let's apply the mandatory resources for nginx Ingress first.

```
kubectl apply \
    --filename https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27\
.0/deploy/static/mandatory.yaml
```

Next, we're going to apply the additional definitions that vary from one platform to another.

You might be using Minikube or Docker Desktop or EKS or AKS or GKE or something completely different. Instructions will differ from one Kubernetes platform to another. No matter the flavor, all those commands will do the same thing. They will install nginx Ingress specific to the selected

Kubernetes platform, and they will retrieve the IP through which we'll be able to access the applications.

If you are not using one of the Kubernetes distributions I tested, you might need to pick one of those is closest to yours, and you might need to modify one of the commands.

Please execute the commands that follow if you are using **Minikube**.

```
minikube addons enable ingress

export INGRESS_HOST=$(minikube ip)
```

Please execute the commands that follow if you are using **Docker Desktop**.

```
kubectl apply \
    --filename https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27\
.0/deploy/static/provider/cloud-generic.yaml

export INGRESS_HOST=$(kubectl \
    --namespace ingress-nginx \
    get service ingress-nginx \
    --output jsonpath="{.status.loadBalancer.ingress[0].hostname}")
```

Please execute the commands that follow if you are using **GKE** or **AKS**.

```
kubectl apply \
    --filename https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27\
.0/deploy/static/provider/cloud-generic.yaml

export INGRESS_HOST=$(kubectl \
    --namespace ingress-nginx \
    get service ingress-nginx \
    --output jsonpath="{.status.loadBalancer.ingress[0].ip}")
```

Please execute the commands that follow if you are using **EKS**.

```
kubectl apply \
    --filename https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27\
.0/deploy/static/provider/aws/service-l4.yaml

kubectl apply \
    --filename https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27\
.0/deploy/static/provider/aws/patch-configmap-l4.yaml

export INGRESS_HOST=$(kubectl \
    --namespace ingress-nginx \
    get service ingress-nginx \
    --output jsonpath="{.status.loadBalancer.ingress[0].hostname}")
```

Ingress should be up and running, and we retrieved its address. In most cases, that address is provided by an external load balancer.

To be on the safe side, we'll output the retrieved Ingress host and double-check that it looks OK.

```
echo $INGRESS_HOST
```

If your output is showing an IP or an address, you're done, and we can move on. On the other hand, if the output is empty, you were probably too hasty, and the external load balancer was not yet created by the time you exported `INGRESS_HOST`. To mitigate that, re-run the `export` command, and the `echo` should not be empty anymore.

Now that Ingress controller is up and running and that we know its IP or its address, we are going to deploy Ingress resource that is tied to our application. Let's take a look at it first.

```
cat k8s/health/ingress.yaml
```

The output is as follows.

```
---

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: go-demo-8
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
  - host: go-demo-8.acme.com
    http:
      paths:
      - backend:
          serviceName: go-demo-8
          servicePort: 80

---

apiVersion: v1
kind: Service
metadata:
  name: go-demo-8
  labels:
    app: go-demo-8
spec:
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 8080
    protocol: TCP
    name: http
  selector:
    app: go-demo-8
```

We can see that we have two resources defined. We have Ingress that will forward all the requests entering our cluster, as long as the address of the host of those requests is `go-demo-8.acme.com`. All the requests coming to that host will be forwarded to the service `go-demo-8` on port `80`.

Further on, we can see below that we have the Service defined as well. It is also called `go-demo-8`. The `type` is `ClusterIP`, meaning that only internal communication is allowed to that service. Remember, Ingress picks up external communication and forwards it to that service. Service will be accessible on port `80`, and the `targetPort`, the port of our application, is `8080`. Finally, we have the selector `app: go-demo-8`. So this service will forward the request to all the Pods with the matching labels.

All that was very basic, and I didn't go into detail. I'm assuming that you already have at least elemental knowledge of Kubernetes. You'll know that you do if the things we did so far in this chapter were boring.

Let's apply that definition.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/health/ingress.yaml
```

Finally, we're going to send a request to our application to validate whether it is indeed accessible through the Ingress controller. To do that, we need to simulate the host because you almost certainly don't have `go-demo-8.acme.com` configured as a domain. To circumvent that, we're going to fake the host by sending a `curl` request with injected `Host: go-demo-8.acme.com` header.

```
curl -H "Host: go-demo-8.acme.com" \
    "http://$INGRESS_HOST"
```

We sent a request to the external load balancer, or whatever is our Ingress host, and we injected the hostname `go-demo-8.acme.com` header so that Ingress thinks that it is coming from that domain.

It worked! The application responded, so we know that everything was set up correctly. Now we can go back to chaos experiments.

Validating Application Health

Let's take a look at yet another chaos experiment definition.

```
cat chaos/health.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we terminate an instance of the application?
description: If an instance of the application is terminated, a new instance should \
be created
tags:
- k8s
- pod
- deployment
steady-state-hypothesis:
  title: The app is healthy
  probes:
  - name: all-apps-are-healthy
    type: probe
    tolerance: true
    provider:
      type: python
      func: all_microservices_healthy
      module: chaosk8s.probes
      arguments:
        ns: go-demo-8
method:
- type: action
  name: terminate-app-pod
  provider:
    type: python
    module: chaosk8s.pod.actions
    func: terminate_pods
    arguments:
      label_selector: app=go-demo-8
      rand: true
      ns: go-demo-8
```

What do we have there?

The `title` asks `what happens if we terminate an instance of an application?`. The `description` says that `if an instance of the application is terminated, a new instance should be created`. Both should be self-explanatory and do not serve any practical purpose. They are informing us about the objectives of the experiment.

"So far, that definition looks almost the same as what we were doing in the previous section." If that's what you're thinking, you're right. Or, to be more precise, they are very similar.

The reason why we're doing this again is that there is an easier and faster way to do what we did before.

Instead of verifying conditions and states of specific Pods of our application, we are going to validate whether all the instances of all the apps in that namespace are healthy. Instead of going from one application to another and checking states and conditions, we're just going to tell the experiment, "look, for you to be successful, everything in that namespace needs to be healthy." We're going to do that by using the function `all_microservices_healthy`. The module is `chaosk8s.probes`, and the argument is simply a `namespace`.

So, our conditions before and after actions are that everything in that namespace needs to be healthy.

Further on, we have an action in the `method` that will terminate a Pod based on that `label_selector` and in that specific Namespace (`ns`). It will be a random Pod with that label selector. Such randomness doesn't make much sense right now since we have only one pod. Nevertheless, it is going to be random, even if there's only one Pod.

Before we run that experiment, please take another look at the definition. I want you to try to figure out whether the experiment will be successful or failed. And, if it will fail, I want you to guess what is missing? Why will it fail? After all, our applications should now be fault-tolerant. It should be healthy, right? Think about the experiment and try to predict the outcome.

Now that you did some thinking and that you guessed the outcome, let's run the experiment.

```
chaos run chaos/health.yaml
```

The output is as follows (timestamps are removed for brevity).

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate an instance of the appli\
cation?
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: all-apps-are-healthy
[... INFO] Steady state hypothesis is met!
[... INFO] Action: terminate-app-pod
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: all-apps-are-healthy
[... ERROR]   => failed: chaoslib.exceptions.ActivityFailed: the system is unhealthy
[... WARNING] Probe terminated unexpectedly, so its tolerance could not be validated
[... CRITICAL] Steady state probe 'all-apps-are-healthy' is not in the given toleran\
ce so failing this experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: deviated
[... INFO] The steady-state has deviated, a weakness may have been discovered
```

We can see that the initial probe passed and that the action was executed. And then, the probe was executed again, and it failed. Why did it fail? What are we missing? We know that Kubernetes will

recreate terminated Pods that are controlled by a Deployment. We validated that behavior in the previous chapter. So, why is it failing now? It should be relatively straightforward to figure out what's missing. But before we go there, let's take a look at the Pods. Are they really running?

```
kubectl --namespace go-demo-8 \
    get pods
```

The output is as follows.

```
NAME             READY STATUS  RESTARTS AGE
go-demo-8-...    1/1   Running 0        9s
go-demo-8-db-... 1/1   Running 0        11m
```

We can see that the database (`go-demo-8-db`) and the API (`go-demo-8`) are both running. In my case, the API, the one without the `-db` suffix, is running for `9` seconds. So, it is really fault-tolerant (failed Pods are recreated), and yet the experiment failed. Why did it fail?

Let's take a look at yet another experiment definition.

```
cat chaos/health-pause.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we terminate an instance of the application?
description: If an instance of the application is terminated, a new instance should \
be created
tags:
- k8s
- pod
- deployment
steady-state-hypothesis:
  title: The app is healthy
  probes:
  - name: all-apps-are-healthy
    type: probe
    tolerance: true
    provider:
      type: python
      func: all_microservices_healthy
      module: chaosk8s.probes
      arguments:
```

```
        ns: go-demo-8
method:
- type: action
  name: terminate-app-pod
  provider:
    type: python
    module: chaosk8s.pod.actions
    func: terminate_pods
    arguments:
      label_selector: app=go-demo-8
      rand: true
      ns: go-demo-8
  pauses:
    after: 10
```

We were missing a pause. That was done intentionally since I want you to understand the importance of not only executing stuff one after another but giving the system appropriate time to recuperate when needed. If you expect your system to have healthy Pods immediately after destruction, then don't put the pause. But, in our case, we're setting the expectation that within `10` seconds of the destruction of an instance, a new one should be fully operational. That's the expectation we're having right now. Yours could be different. We could expect the system to recuperate immediately, or it could be after one second, or after three days. With pauses, we are setting the expectation of how long the system would need to recuperate from a potentially destructive action. In this case, we are setting it to `10` seconds.

So, let's confirm that's really the only change by outputting the differences between the old and the new definition.

```
diff chaos/health.yaml \
    chaos/health-pause.yaml
```

The output is as follows.

```
>   pauses:
>     after: 10
```

We can see that, indeed, the only change is the additional pause of `10` seconds after the action.

Let's run this experiment and see what we're getting.

```
chaos run chaos/health-pause.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate an instance of the appli\
cation?
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: all-apps-are-healthy
[... INFO] Steady state hypothesis is met!
[... INFO] Action: terminate-app-pod
[... INFO] Pausing after activity for 10s...
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: all-apps-are-healthy
[... INFO] Steady state hypothesis is met!
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: completed
```

The probe confirmed that, initially, everything is healthy. All the instances were operational. Then we performed an action to terminate a Pod, and we waited for ten seconds. We can observe that pause by checking the timestamps of the `Pausing after activity for 10s...` and the `Steady state hypothesis: The app is healthy` events. After that, the probe confirmed that all the instances of the applications are healthy.

Let's take another look at the Pods and confirm that's really true.

```
kubectl --namespace go-demo-8 \
    get pods
```

The output is as follows.

```
NAME              READY STATUS  RESTARTS AGE
go-demo-8-...     1/1   Running 0        2m19s
go-demo-8-db-...  1/1   Running 0        22m
```

AS you can see, the new Pod was created, and, in my case, it already exists for over two minutes.

Validating Application Availability

Validating whether all the Pods are healthy and running is useful. But that does not necessarily mean that our application is accessible. Maybe the Pods are running, and everything is fantastic and peachy, but our customers cannot access our application.

Let's see how we can validate whether we can send HTTP requests to our application and whether we can continue doing that after an instance of our app is destroyed. How would that definition look like?

Before we dive into application availability, we have a tiny problem that needs to be addressed. I couldn't define the address of our application in YAML because your IP is almost certainly different than mine. And neither of us are using real domains because that would be too complicated to set up. That problem allows me to introduce you to yet another feature of Chaos Toolkit.

We are going to define a variable that can be injected into our definition.

Let's take a quick look at yet another YAML.

```
cat chaos/health-http.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we terminate an instance of the application?
description: If an instance of the application is terminated, the applications as a \
whole should still be operational.
tags:
- k8s
- pod
- http
configuration:
  ingress_host:
      type: env
      key: INGRESS_HOST
steady-state-hypothesis:
  title: The app is healthy
  probes:
  - name: app-responds-to-requests
    type: probe
    tolerance: 200
    provider:
      type: http
      timeout: 3
      verify_tls: false
      url: http://${ingress_host}/demo/person
      headers:
        Host: go-demo-8.acme.com
method:
- type: action
```

```
  name: terminate-app-pod
  provider:
    type: python
    module: chaosk8s.pod.actions
    func: terminate_pods
    arguments:
      label_selector: app=go-demo-8
      rand: true
      ns: go-demo-8
  pauses:
    after: 2
```

So the new sections of this definition, when compared to the previous one, is that we added a configuration section and we changed our `steady-state-hypothesis`. There are a few other changes. Some of those are cosmetic, while others are indeed important.

We'll skip commenting on the contents of this file because it is hard to see what's different when compared to what we had before. We'll comment on the differences by executing a `diff` between the new and the old version of the definition.

```
diff chaos/health-pause.yaml \
    chaos/health-http.yaml
```

The output is as follows.

```
3c3
< description: If an instance of the application is terminated, a new instance shoul\
d be created
---
> description: If an instance of the application is terminated, the applications as \
a whole should still be operational.
7c7,11
< - deployment
---
> - http
> configuration:
>   ingress_host:
>       type: env
>       key: INGRESS_HOST
11c15
<   - name: all-apps-are-healthy
---
>   - name: app-responds-to-requests
```

```
13c17
<     tolerance: true
---
>     tolerance: 200
15,19c19,24
<       type: python
<       func: all_microservices_healthy
<       module: chaosk8s.probes
<       arguments:
<         ns: go-demo-8
---
>       type: http
>       timeout: 3
>       verify_tls: false
>       url: http://${ingress_host}/demo/person
>       headers:
>         Host: go-demo-8.acme.com
32c37
<     after: 10
---
>     after: 2
```

We can see that, this time, quite a lot of things changed. The `description` and the `tags` are different. Those are only informative, so there's no reason to go through them.

What matters is that we added a `configuration` section and that it has a variable called `ingress_host`.

Variables can be of different types, and, in this case, the one we defined is an environment variable (`env`). The `key` is `INGRESS_HOST`. That means that if we set the environment variable `INGRESS_HOST`, the value of that variable will be assigned to Chaos Toolkit variable `ingress_host`.

The name of the `steady-state-hypothesis` changed, and the `tolerance` is now `200`. Before, we were validating whether our application is healthy by checking whether the output is `true` or `false`. This time, however, we will verify whether our app responds with `200` response code, or with something else.

Further on, we can see that we changed the probe `type` from `all_microservices_healthy` to `http`. The `timeout` is set to `3` seconds.

All in all, we expect our application to respond within three seconds. That's unrealistically high, I would say. It should much lower like, for example, a hundred milliseconds. However, I couldn't be hundred percent sure that your networking is fast, so I defined a relatively high value to be on the safe side.

You should also note that we are not verifying TLS (`verify_tls`) because it would be hard to define

certificates without having a proper domain. In "real life" situations, you would always validate TLS.

Now comes the vital part.

The `url` to which we're going to send the request is using the `ingress_host` variable. We'll be sending requests to our Ingress domain/IP and to the path `/demo/person`. Since our Ingress is configured to accept only requests from `go-demo-8.acme.com`, we are adding that `Host` as one of the `headers` to that request.

Finally, we're changing the `pause`. Instead of `10` seconds we had before, we're going to give it `2` seconds. After destroying a Pod, we're going to wait for two seconds, and then we're going to validate whether we can send a request to it and whether it responds with `200`. I wanted to ensure that the Pod is indeed destroyed. Otherwise, we wouldn't need such a pause.

Let's run this experiment and see what we'll get.

```
chaos run chaos/health-http.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate an instance of the appli\
cation?
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Action: terminate-app-pod
[... INFO] Pausing after activity for 2s...
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... CRITICAL] Steady state probe 'app-responds-to-requests' is not in the given tol\
erance so failing this experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: deviated
[... INFO] The steady-state has deviated, a weakness may have been discovered
```

The initial probe passed. It validated that our application responds to requests before it started executing probes. The action that terminated a Pod was executed and, after that, we waited for two seconds. Further on, we can see that the post-action probe failed. Two seconds after destroying an instance of the application, we were unable to send a request to it. That's very unfortunate, isn't it? Why does that happen?

I want you to stop here and think. Why is our application not highly available? It should be responsive and available at all times. If you destroy an instance of our application, it should continue serving our users. So, think about why it is not highly available, and what is missing?

Did you figure it out? Kudos to you if you did. If you didn't, I'll provide an answer.

Our application is not highly available. It does not continue serving requests after a Pod, or an instance is destroyed because there is only one instance. Every application, when architecture allows, should run multiple instances as a way to prevent this type of situation. If, for example, we would have three instances of our application and we'd destroy one of them, the other two should be able to continue serving requests while Kubernetes is recreating the failed Pod. In other words, we need to increase the number of replicas of our application.

We could scale up in quite a few ways. We could just go to the definition of the Deployment and say that there should be two or three or four replicas of that application. But that's a bad idea. That's static. That would mean that if we say three replicas, then our app would always have three replicas. What we want is for our application to go up and down. It should increase and decrease the number of instances depending on their memory or CPU utilization. We could even define more complicated criteria based on Prometheus.

We're not going to go into details of how to scale applications. That's not the subject of this book. Instead, I'll just say that we're going to define HorizontalPodAutoscaler (HPA). So, let's take a look at yet another YAML.

```
cat k8s/health/hpa.yaml
```

The output is as follows.

```
---

apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: go-demo-8
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: go-demo-8
  minReplicas: 2
  maxReplicas: 6
  metrics:
  - type: Resource
    resource:
      name: cpu
```

```
      targetAverageUtilization: 80
  - type: Resource
    resource:
      name: memory
      targetAverageUtilization: 80
```

That definition specifies a `HorizontalPodAutoscaler` called `go-demo-8`. It is targeting the `Deployment` with the same name `go-demo-8`. The minimum number of replicas will be `2`, and the maximum number will be `6`. Our application will have anything between two and six instances. Which exact number will it be, depends on the metrics.

In this case, we have two basic metrics. The average utilization of CPU should be around 80%, and the average usage of memory should be 80% as well. In most "real-world" cases, those two `metrics` would be insufficient. Nevertheless, they should be suitable for our example.

All in all, that HPA will make our application run at least two replicas. That should hopefully make it highly available. If you're unsure about the validity of that statement, try to guess what happens if we destroy one or multiple replicas of an application. The others should continue serving requests.

Let's deploy the HorizontalPodAutoscaler.

```
kubectl apply --namespace go-demo-8 \
    --filename k8s/health/hpa.yaml
```

To be on the safe side, we'll retrieve HPAs and confirm that everything looks OK.

```
kubectl --namespace go-demo-8 \
    get hpa
```

The output is as follows.

```
NAME      REFERENCE             TARGETS          MINPODS MAXPODS REPLICAS AGE
go-demo-8 Deployment/go-demo-8 16%/80%, 0%/80% 2       6       2        51s
```

It might take a few moments until the HPA figures out what it needs to do. Keep repeating the `get hpa` command until the number of replicas increases to `2` (or more).

Now we're ready to proceed. The HorizontalPodAutoscaler increased the number of replicas of our application. However, we are yet to see whether that was enough. Is our app now highly available?

Just like everything we do in this book, we are always going to validate our theories and ideas by running chaos experiments. So let's re-run the same experiment as before and see what happens. Remember, the experiment we are about to run (the same one as before) validates whether our application can serve requests after an instance of that application is destroyed. In other words, it checks whether the app highly available.

```
chaos run chaos/health-http.yaml
```

The output, without the timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate an instance of the appli\
cation?
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Action: terminate-app-pod
[... INFO] Pausing after activity for 2s...
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: completed
```

We can see that the initial probe passed and that the action was executed to terminate a Pod. After that, it waited for two seconds just to make hundred percent sure that the Pod is destroyed. Then we re-run the probe. It passed! Everything was successful. Our experiment was a success. Our application is indeed highly available.

Terminating Application Dependencies

There's one more thing within the area of what we're exploring that we might want to try. We might want to check what happens if we destroy an instance of a dependency of our application. As you already know, our demo application depends on MongoDB. We saw what happens when we destroy an instance of our application. Next, we'll explore how does the application behaves if we terminate a replica of the database (the dependency).

We are going to take a look at yet another chaos experiment definition.

```
cat chaos/health-db.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we terminate an instance of the DB?
description: If an instance of the DB is terminated, dependant applications should s\
till be operational.
tags:
- k8s
- pod
- http
configuration:
  ingress_host:
      type: env
      key: INGRESS_HOST
steady-state-hypothesis:
  title: The app is healthy
  probes:
  - name: app-responds-to-requests
    type: probe
    tolerance: 200
    provider:
      type: http
      timeout: 3
      verify_tls: false
      url: http://${ingress_host}/demo/person
      headers:
        Host: go-demo-8.acme.com
method:
- type: action
  name: terminate-db-pod
  provider:
    type: python
    module: chaosk8s.pod.actions
    func: terminate_pods
    arguments:
      label_selector: app=go-demo-8-db
      rand: true
      ns: go-demo-8
  pauses:
    after: 2
```

As you're already accustomed, we'll output differences between that definition and the one we used before. That will help us spot what really changed, given that the differences are tiny and subtle.

```
diff chaos/health-http.yaml \
    chaos/health-db.yaml
```

The output is as follows.

```
2,3c2,3
< title: What happens if we terminate an instance of the application?
< description: If an instance of the application is terminated, the applications as \
a whole should still be operational.
---
> title: What happens if we terminate an instance of the DB?
> description: If an instance of the DB is terminated, dependant applications should\
 still be operational.
27c27
<   name: terminate-app-pod
---
>   name: terminate-db-pod
33c33
<       label_selector: app=go-demo-8
---
>       label_selector: app=go-demo-8-db
```

We can see that, if we ignore the `title`, the `description`, and the `name`, what really changed is the `label_selector`. We'll terminate one of the Pods with the label `app` set to `go-demo-8-db`. In other words, we are doing the same thing as before. Instead of terminating instances of our application, we are terminating instances of the database that is the dependency of our app.

Let's see what happens. Remember, the previous experiment that was terminating a Pod of the application passed. We could continue sending requests. It is highly available.

Let's see what happens when we do the same to the database.

```
chaos run chaos/health-db.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we terminate an instance of the DB?
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Action: terminate-db-pod
[... INFO] Pausing after activity for 2s...
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... ERROR]   => failed: activity took too long to complete
[... WARNING] Probe terminated unexpectedly, so its tolerance could not be validated
[... CRITICAL] Steady state probe 'app-responds-to-requests' is not in the given tol\
erance so failing this experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: deviated
[... INFO] The steady-state has deviated, a weakness may have been discovered
```

The initial probe passed, and the action was successful. It terminated an instance of the database, it waited for two seconds, and then it failed. Our application does not respond to requests when the associated database is not there.

We know that Kubernetes will recreate the terminated Pod of the database. However, there is downtime between destroying an existing Pod and the new one being up and running. Not only that the database is not available, but our application is not available either.

Unlike other cases where I showed you how to fix something, from now on, you will have homework. The goal is for you to figure out how to solve this problem. How can you make this experiment a success? You should have a clue how to do that from the application. It's not really that different. So think about it and take a break from reading this book. Execute the experiment again when you figure it out.

If the homework turns up to be too difficult, I'll give you a tip. Do the same thing to the database as we did to the application. You need to run multiple instances of it to make it highly available. Our demo application depends on it, and it will never be highly available if the database is not available. So, multiple instances of the database are required as well. Since the database is stateful, you probably want to use StatefulSet instead of a Deployment, and I strongly recommend that you don't try to define all that by yourself. There is an excellent Helm chart in the stable channel. Search "MongoDB stable channel Helm chart", and you will find a definition of MongoDB that can be replicated and made highly available.

Destroying What We Created

That's it. One more chapter is finished, and it's time for us to terminate the application by executing the same commands as before.

```
cd ..

kubectl delete namespace go-demo-8
```

We went one directory back, and we deleted the whole `go-demo-8` namespace.

What will happen next is your choice. If you want to take a break, destroy the cluster. You will find the instructions in the same Gist that you used to create it. Or, you can choose to keep the cluster and move forward right away. It's up to you.

Obstructing And Destroying Network

So far, we saw how to destroy physical things, or, to be more precise, how to terminate instances of our applications. But we can accomplish so much more by introducing networking into the picture. By destroying, obstructing, or faking certain things in networking, we can put experiments and chaos engineering to an entirely different level.

There are many different ways how we can do networking in Kubernetes. But for today, we are going to explore the one that is most commonly used, and that happens to be Istio. I am choosing it mostly because today (March 2020), it is the most widely used service mesh networking solution. The same lessons that you will learn from using Istio and combining it with chaos experiments, you could apply to other networking solutions. An additional advantage of using Istio is that it is already available in Chaos Toolkit through a plugin. For other networking types, you might need to work a bit more to create your own commands. That should be relatively easy because you can execute any command from Chaos Toolkit. Nevertheless, there is a plugin for Istio. We are going to use it, given that Istio is the most commonly used service mesh.

By now, you might be wondering why do we need service mesh in the first place. There could be many answers to that question, so I'll limit them to the context of chaos engineering. Service meshes have options to abort requests, to circumvent certain things, to delay requests, and so on and so forth. They are mighty, and they fit very well into the experiments that we are going to run.

All in all, we'll experiment on top of Istio.

Gist With The Commands

As you already know, every section that has hands-on exercises is accompanied by a Gist that allows you to copy and paste stuff instead of typing. You might be lazy, and you might not want to stare at the book and try to type all the commands and definitions we'll need. If that's the case, a Gist is just the thing you need.

All the commands from this chapter are available in the 05-network.sh[46] Gist.

[46] https://gist.github.com/455b0321879da7abf4d358a1334fd705

Creating A Cluster

We need a cluster where we'll deploy the demo application that we'll target in our experiments. Whether you have it or not depends on whether you destroyed your cluster at the end of the previous chapter. If you didn't destroy it, just skip to the next section right away. If you did destroy the cluster, use one of the Gists that follow, or just roll out your own cluster. It's up to you. What matters is that you need to have a Kubernetes cluster.

Gists with the commands to create and destroy a Kubernetes cluster are as follows.

- Docker Desktop: docker.sh[47]
- Minikube: minikube.sh[48]
- GKE: gke.sh[49]
- EKS: eks.sh[50]
- AKS: aks.sh[51]

Installing Istio Service Mesh

I already revealed that we are going to experiment with networking and that we will use Istio as a service mesh of choice. Now we need to install it. But, before we do that, there is a warning for those using Docker Desktop.

If you are using **Docker Desktop**, you need to remove ingress from your Kubernetes cluster. The reason is simple. In Docker Desktop, both nginx Ingress and Istio Gateway are using port 80. If we use both, there would be conflicts between the two. So, if you're a Docker Desktop user and you kept the cluster from the previous chapter, please execute the commands that follow to remove Ingress.

Please execute the commands that follow only if you are re-using the **Docker Desktop** cluster from the **previous chapter**.

[47] https://gist.github.com/f753c0093a0893a1459da663949df618
[48] https://gist.github.com/ddc923c137cd48e18a04d98b5913f64b
[49] https://gist.github.com/2351032b5031ba3420d2fb9a1c2abd7e
[50] https://gist.github.com/be32717b225891b69da2605a3123bb33
[51] https://gist.github.com/c7c9a8603c560eaf88d28db16b14768c

```
kubectl delete \
    --filename https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27\
.0/deploy/static/provider/cloud-generic.yaml

kubectl delete \
    --filename https://raw.githubusercontent.com/kubernetes/ingress-nginx/nginx-0.27\
.0/deploy/static/mandatory.yaml
```

To work with Istio, you will need `istioctl`. If you don't have it already, please go to Istio Releases[52] and install it.

Now we can install Istio inside our Kubernetes cluster.

If you're already familiar with Istio, you know what's coming. You can just skip the instructions and install it yourself. If you're new to it, you will see that installing Istio is relatively easy and straightforward. All we have to do is execute `istioctl manifest install`, and that will install the default version. We will also add `--skip-confirmation` flag so that `istioctl` does not ask us to confirm whether we want to install it or not.

Please execute the command that follows.

```
istioctl manifest install \
    --skip-confirmation
```

It shouldn't take long until everything is installed. Once you see that all the resources are created, we should confirm that the external IP was assigned to the Istio Gateway service.

```
kubectl --namespace istio-system \
    get service istio-ingressgateway
```

In most cases, at least when running Kubernetes in the cloud, external IP means that it is creating an external load balancer (LB). If you see the value of the `EXTERNAL-IP` column set to `<pending>`, the external LB is still being created. If that's the case, keep re-running the previous command until you see the value of that column set to an IP.

Docker Desktop and **Minikube** cannot create an external load balancer, so the `EXTERNAL-IP` column will stay `<pending>` forever. That's OK. If one of those is your Kubernetes distribution of choice, you can ignore me saying that you should wait until the value of that column becomes an IP.

Now, to simplify everything, we are going to retrieve the Istio Ingress host and store it in a variable `INGRESS_HOST`. To do that, commands will differ depending on the type of the Kubernetes cluster you have. Follow the instructions for your distribution.

[52] https://github.com/istio/istio/releases

Please execute the commands that follow to retrieve the Istio Ingress IP if you're using **Minikube**.

```
export INGRESS_PORT=$(kubectl \
    --namespace istio-system \
    get service istio-ingressgateway \
    --output jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')

export INGRESS_HOST=$(minikube ip):$INGRESS_PORT
```

Please execute the commands that follow to retrieve the Istio Ingress IP if you're using **Docker Desktop**.

```
export INGRESS_HOST=127.0.0.1
```

Please execute the commands that follow to retrieve the Istio Ingress IP if you're using **GKE** or **AKS**.

```
export INGRESS_HOST=$(kubectl \
    --namespace istio-system \
    get service istio-ingressgateway \
    --output jsonpath="{.status.loadBalancer.ingress[0].ip}")
```

Please execute the commands that follow to retrieve the Istio Ingress IP if you're using **EKS**.

```
export INGRESS_HOST=$(kubectl \
    --namespace istio-system \
    get service istio-ingressgateway \
    --output jsonpath="{.status.loadBalancer.ingress[0].hostname}")
```

To be on the safe side, we'll output the `INGRESS_HOST` variable and confirm that it looks okay.

```
echo $INGRESS_HOST
```

If the output is an IP with an optional port (in case of Minikube), or a domain (in case of EKS), we're done setting up Istio, and we can do some chaos. We can create and run experiments and see how they fit into networking, Ingress, and a few other concepts. But, before we do that, we still need to deploy our demo application.

Deploying The Application

What we need to do next is deploy our demo application. But, this time, since we already saw that we are going to extend our chaos experiments into networking and that we are going to use Istio for that, our application will be slightly more complex. We will need to add a couple of Istio resources to it.

You might already be proficient with Istio. If you're not, then you might not find sufficient information about what we are going to create here. I will not go into details about Virtual Services, Gateways, and other Istio resources. If that is not your strong point, I strongly recommend that you check out the Canary Deployments in Kubernetes with Istio And Friends[53] course in Udemy[54]. Even though it is not focused on everything there is to know about Istio, it does, in an indirect way, provide insights into the most important concepts and constructs.

Besides Istio-specific resources, we will also need to deploy one additional application, but I will explain that later. For now, let's start by deploying the app in the same way as we did before.

First, we're going to go to the `go-demo-8` directory, and we are going to pull the latest version of the repository. That'll make sure that you have any changes that I might have made since the last time you cloned it.

```
cd go-demo-8

git pull
```

Then we're going to create the `go-demo-8` Namespace where our application will be running.

```
kubectl create namespace go-demo-8
```

Now comes the new part. We're going to label that Namespace with `istio-injection=enabled`. That will give a signal to Istio that it should add a proxy container to every single Pod running in that Namespace.

[53] https://www.udemy.com/course/canary-deployments-to-kubernetes-using-istio-and-friends/?referralCode=75549ECDBC41B27D94C4
[54] https://www.udemy.com/

```
kubectl label namespace go-demo-8 \
    istio-injection=enabled
```

Then, we are going to deploy our application, just as we did before, and without any changes. The only difference is that this time, there will be no nginx Ingress resource. Soon, we're going to start using Istio gateway for that. So let's take a quick look at what we have in the `k8s/health/app` directory.

```
cat k8s/health/app/*
```

The output is too big to be presented in a book. If you take a closer look, you'll notice that those are the same definitions we used in the previous chapter, so commenting on them would only introduce unnecessary delay. Instead, we'll just `apply` those definitions and wait until the `go-demo-8` Deployment rolls out.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/health/app/

kubectl --namespace go-demo-8 \
    rollout status deployment go-demo-8
```

Now we have our application running in the same way as before. The only difference is that it is without the Ingress resource.

Let's take a quick look at the Pods we just created.

```
kubectl --namespace go-demo-8 \
    get pods
```

The output is as follows.

```
NAME              READY STATUS  RESTARTS AGE
go-demo-8-...     2/2   Running 2        113s
go-demo-8-...     2/2   Running 1        98s
go-demo-8-db-...  2/2   Running 0        113s
```

We can see that this time, all the Pods have two containers. One of those containers is ours, and the additional one was injected by Istio. That is the proxy sidecar running in every single Pod. However, sidecar containers are not enough by themselves, and we still need to create a few Istio resources.

Next, we'll take a look at a few additional resources we'll need for the applications to be managed by Istio, or, to be more precise, for networking of those applications to be managed by Istio.

```
cat k8s/network/istio.yaml
```

The output is as follows.

```
---

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: go-demo-8
spec:
  hosts:
  - go-demo-8
  http:
  - route:
    - destination:
        host: go-demo-8
        subset: primary
        port:
          number: 80

---

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: go-demo-8
spec:
  host: go-demo-8
  subsets:
  - name: primary
    labels:
      release: primary
```

We have a Virtual Service called `go-demo-8` that will allow internal traffic to our application. We can see that we have the `host` set to `go-demo-8` (the name of the Service associated with the app). The `destination` is also set to the same host, and the `subset` is set to `primary`. Typically, we would have primary (and secondary) subsets if we'd use Canary Deployments. But, in this case, we are not going to do that. If you're curious about canaries, please check out the Canary Deployments in Kubernetes with Istio And Friends[55] course in Udemy[56]. In any case, we are going to define only the `primary` subset as being the only one since we won't have canary deployments this time. Finally,

[55] https://www.udemy.com/course/canary-deployments-to-kubernetes-using-istio-and-friends/?referralCode=75549ECDBC41B27D94C4
[56] https://www.udemy.com/

the port is set to `80`. That is the port through which network requests will be coming to this Virtual Service.

Then we have the `DestinationRule` resource, which is pretty straightforward. It is called `go-demo-8`, the host is also `go-demo-8`. It points to the `primary` subset, which will forward all the requests to `go-demo-8` Pods that have the label `release` set to `primary`.

Let's apply that definition before we see what else we might need.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/network/istio.yaml
```

Next, we're going to take a look at a new application. We're going to deploy something I call `repeater`. Why we're introducing a new (third) application, besides the API and the DB. For us to do chaos to networking, we will need an additional app so that we can, for example, do some damage to the networking of the API and see how that new application connected to it works.

The `repeater` is a very simple application. All it does is forward requests coming into it to the specified address. So, for example, if we send a request to the `repeater`, and we specify that we would like it to forward that request to `go-demo-8`, that's where it will go. It is intentionally very simple because the objective is to see how multiple applications collaborate together through networking and what happens when we do some damage to the network.

So let's take a look at the definition.

```
cat k8s/network/repeater/*
```

I will not present the output nor comment (much) on it. You can see it on your screen, and you should be able to deduce what each of the resources does. The Deployment contains the definition of the `repeater` application. HorizontalPodAutoscaler will make sure that the app scales up and down depending on memory and CPU usage, as well as that there are never less than two replicas. The VirtualService, the DestinationRule, and the Gateway are Istio resources that will handle incoming and outgoing traffic. Finally, we have a "standard" Kubernetes Service, and you hopefully already know what it does.

Let's apply the definitions of this new application and wait until it rolls out.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/network/repeater

kubectl --namespace go-demo-8 \
    rollout status deployment repeater
```

Now we should be ready.

We have our `go-demo-8` application, and we have the associated database. Those two are the same as before with the addition of Istio resources, and we have a new app called `repeater`.

The only thing left is to double-check whether all that works. We're going to do that by sending a request to the `repeater`.

```
curl -H "Host: repeater.acme.com" \
    "http://$INGRESS_HOST?addr=http://go-demo-8"
```

We sent a request with the "fake" domain, just as we did a few times before. The major difference is that, this time, the request was sent with the `Host` header set to `repeater.acme.com`, and that there is the `addr=http://go-demo-8` parameter in the address.

When we sent a request to the `repeater`, it forwarded it to the value of the query parameter `addr`, which, in turn, is set to `go-demo-8`. So, it just forwards requests wherever we specify. We can see that's true since we sent a request to the `repeater` but got the familiar response from `go-demo-8` saying `Version: 0.0.1`.

Discovering Chaos Toolkit Istio Plugin

You probably remember that we had to install the `chaostoolkit-kubernetes` module to define Kubernetes-related chaos experiments. The same is true for Istio. We could, theoretically, define Istio-related actions, probes, and rollbacks as shell commands that would be applying YAML definitions and running `istioctl`. Nevertheless, that would be a waste of time since the `chaostoolkit-istio` module already provides some (if not all) Istio-related features. It will allow us to add some additional capabilities that we wouldn't have otherwise or, to be more precise, that would be much harder without it. So, let's install the module.

```
pip install -U chaostoolkit-istio
```

Next, we are going to discover what is inside that module.

```
chaos discover chaostoolkit-istio

cat discovery.json
```

The outcome of `chaos discover` was stored in `discovery.json`, which, later on, we output on the screen. I will not go, at this moment, into all the functions that we got with the `chaostoolkit-istio` module. Instead, I will let you explore yourself by observing what inside the discovery file. Don't worry if it's overwhelming. We're going to use most, if not all, of them very soon.

Aborting Network Requests

Networking issues are very common. They happen more often than many people think. We are about to explore what happens when we simulate or when we create those same issues ourselves.

So, what can we do?

We can do many different things. But, in our case, we'll start with something relatively simple. We'll see what happens if we intentionally abort some of the network requests. We're going to terminate requests and see how our application behaves when that happens. We're not going to abort all the requests, but only some. Terminating 50% of requests should do.

What happens if 50% of the requests coming to our applications are terminated? Is our application resilient enough to survive that without negatively affecting users? As you can probably guess, we can check that through an experiment.

Let's take a look at yet another Chaos Toolkit definition.

```
cat chaos/network.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we abort responses
description: If responses are aborted, the dependant application should retry and/or\
 timeout requests
tags:
- k8s
- istio
- http
configuration:
  ingress_host:
      type: env
      key: INGRESS_HOST
steady-state-hypothesis:
  title: The app is healthy
  probes:
  - type: probe
    name: app-responds-to-requests
    tolerance: 200
    provider:
      type: http
      timeout: 5
      verify_tls: false
```

```
      url: http://${ingress_host}?addr=http://go-demo-8
      headers:
        Host: repeater.acme.com
  - type: probe
    tolerance: 200
    ref: app-responds-to-requests
  - type: probe
    tolerance: 200
    ref: app-responds-to-requests
  - type: probe
    tolerance: 200
    ref: app-responds-to-requests
  - type: probe
    tolerance: 200
    ref: app-responds-to-requests
method:
- type: action
  name: abort-failure
  provider:
    type: python
    module: chaosistio.fault.actions
    func: add_abort_fault
    arguments:
      virtual_service_name: go-demo-8
      http_status: 500
      routes:
        - destination:
            host: go-demo-8
            subset: primary
      percentage: 50
      version: networking.istio.io/v1alpha3
      ns: go-demo-8
  pauses:
    after: 1
```

At the top, we have general information like the `title` asking `what happens if we abort responses` and the `description` stating that `if responses are aborted, the dependant application should retry and/or timeout requests`. Those are reasonable questions and assumptions. If something bad happens with requests, we should probably retry or timeout them. We also have some `tags` telling us that the experiment is about `k8s`, `istio`, and `http`. Just as before, we have `configuration` that will allow us to convert the environment variable `INGRESS_HOST` into Chaos Toolkit variable `ingress_host`. And we have a `steady-state-hypothesis` that validates that the application is healthy. We're measuring that health by sending a request to our application and expecting that the return code is

200. We are, more or less, doing the same thing as before. However, this time, we are not sending a request to go-demo-8 but to the repeater.

Since we are going to abort 50% of the requests, having only one probe with a request might not actually produce the result that we want. That would rely on luck since we couldn't predict whether that request would fall into 50% that are aborted. To reduce the possibility of randomness having an influence on our steady-state hypothesis, we are going to repeat that request four more times. But, instead of defining the whole probe, we have a shortcut definition. The second probe also has the tolerance 200, but it is referencing the probe app-responds-to-requests. So, instead of repeating everything, we are just referencing the existing probe, and we are doing that four times.

All in all, we are sending requests, and we're expecting 200 response code five times.

Then we have a method with the action abort-failure. It's using the module chaosistio.fault.actions and the function add_abort_fault. It should be self-descriptive, and you should be able to guess that it will add abort faults into an Istio Virtual Service. We can also see that the action is targeting the Virtual Service go-demo-8.

All in all, the add_abort_fault function will inject HTTP status 500 to the Virtual Service go-demo-8 that is identified through the destination with the host set to go-demo-8 and the subset set to primary. Further on, we can see that we have the percentage set to 50. So, fifty percent of the requests to go-demo-8 will be aborted. We also have the version of Istio that we're using and the Namespace (ns) where that Virtual Service is residing.

So, we will be sending requests to the repeater, but we will be aborting those requests on the go-demo-8 API. That's why we added an additional application. Since the repeater forwards requests to go-demo-8, we will be able to see what happens when we interact with one application that interacts with another while there is a cut in that communication between the two.

After we inject the abort, just to be sure that we are not too hasty, we're going to give the system one second pause so that the abortion can be adequately propagated to the Virtual Service.

Now, let's see what happens when we run this experiment. Can you guess? It should be obvious what happens if we abort 50% of the responses, and we are validating whether our application is responsive. Will all five requests that will be sent to our application return status code 200?

Let's run the experiment and see.

```
1 chaos run chaos/network.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we abort responses
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Action: abort-failure
[... INFO] Pausing after activity for 1s...
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... CRITICAL] Steady state probe 'app-responds-to-requests' is not in the given tol\
erance so failing this experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: deviated
[... INFO] The steady-state has deviated, a weakness may have been discovered
```

Please note that the output in your case could be different.

The probe was executed five times successfully. Then the action added abort failures to the Istio Virtual Service. We were waiting for one second, and then we started re-running the probes.

We can see that, in my case, the first probe failed. I was unlucky. Given that approximately 50% should be unsuccessful, it could have been the second, or the third, or any other probe that failed. But my luck ran out right away. The first probe failed, and that was the end of the experiment. It first out of five post-action probes. That was to be expected, or, to be more precise, it was expected that one of those probes failed. That didn't have to be the first one, though.

Rolling Back Abort Failures

We are going to forget about experiments for a moment and see what happens if we send requests to the application ourselves. We're going to dispatch ten requests to `repeater.acme.com`, which is the same address as the address in the experiment. To be more precise, we'll fake that we're sending requests to `repeater.acme.com`, and the "real" address will be the Istio Gateway Ingress host.

```
for i in {1..10}; do
    curl -H "Host: repeater.acme.com" \
        "http://$INGRESS_HOST?addr=http://go-demo-8"
    echo ""
done
```

To make the output more readable, we added an empty line after requests.

The output, in my case, is as follows.

```
fault filter abort
Version: 0.0.1; Release: unknown

fault filter abort
fault filter abort
Version: 0.0.1; Release: unknown

fault filter abort
fault filter abort
Version: 0.0.1; Release: unknown

Version: 0.0.1; Release: unknown

fault filter abort
```

We can see that some of the requests returned `fault filter abort`. Those requests are the 50% that was aborted. Now, don't take 50% seriously because other requests are happening inside the cluster, and the number of those that failed in that output might not be exactly half. Think of it as approximately 50%.

What matters is that some requests were aborted, and others were successful. That is very problematic for at least two reasons. First, the experiment showed that our application cannot deal with network abortions. If a request is terminated (and that is inevitable), our app does not know how to deal with it. The second issue is that we did not roll back our change, so the injected faults are present even after the chaos experiment. We can see that through the 10 requests we just sent. Some of them aborted, while others were successful.

We'll need to figure out how to roll back that change at the end of the experiment. We need to remove the damage we're doing to our network. But, before we do that, let's confirm that our Virtual Service is indeed permanently messed up.

```
kubectl --namespace go-demo-8 \
    describe virtualservice go-demo-8
```

The output, limited to the relevant parts, is as follows.

```
...
Spec:
  Hosts:
    go-demo-8
  Http:
    Fault:
      Abort:
        Http Status:  500
        Percentage:
          Value:  50
...
```

We can see that, within the `Spec.Http` section, there is the `Fault.Abort` subsection, with `Http Status` set to `500` and `Percentage` to `50`.

Istio allows us to do such things. It will enable us, among many other possibilities, to specify how many HTTP responses should be aborted. Through Chaos Toolkit, we run an experiment that modified the definition of the Virtual Service by adding `Fault.Abort`. What we did not do is revert that change.

Everything we do with chaos experiments should always roll back after the experiment to what we had before unless it is a temporary change. So, for example, if we destroy a Pod, we expect Kubernetes to create a new one, so there is no need to revert such a change. But, injecting abort failures into our Virtual Service is permanent, and the system is not supposed to recover from that in any formal way. We need to roll it back.

We can explain the concept differently. We should revert changes we're doing to definitions of the components inside our cluster. If we only destroy something, that is not a change of any definition. We're not changing the desired state. Adding Istio abort failures is a change of a definition while terminating a Pod is not. Therefore, the former should be reverted, while the latter doesn't.

Before we improve our experiment, let's apply `istio.yaml` file again. That will restore our Virtual Service to its original definition.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/network/istio.yaml
```

We rolled back, not through a chaos experiment, but by re-applying the same definition `istio.yaml` that we used initially.

Now, let's describe the Virtual Service and confirm that re-applying the definition worked.

```
kubectl --namespace go-demo-8 \
    describe virtualservice go-demo-8
```

The output, limited to the relevant parts, is as follows.

```
...
Spec:
  Hosts:
    go-demo-8
  Http:
    Route:
      Destination:
        Host:  go-demo-8
        Port:
          Number:  80
        Subset:    primary
...
```

We can see that there is no `Fault.Abort` section. From now on, our Virtual Service should be working correctly with all requests.

Now that we got back to where we started, let's take a look at yet another chaos experiment.

```
cat chaos/network-rollback.yaml
```

As you can see from the output, there is a new section `rollbacks`. To avoid converting this into "where's Waldo" type of exercise, we'll output `diff` so that you can easily see the changes when compared to the previous experiment.

```
diff chaos/network.yaml \
    chaos/network-rollback.yaml
```

The output is as follows.

```
55a56,70
> rollbacks:
> - type: action
>   name: remove-abort-failure
>   provider:
>     type: python
>     func: remove_abort_fault
>     module: chaosistio.fault.actions
>     arguments:
>       virtual_service_name: go-demo-8
>       routes:
>         - destination:
>             host: go-demo-8
>             subset: primary
```

```
15 >       version: networking.istio.io/v1alpha3
16 >       ns: go-demo-8
```

And we can see, the only addition is the `rollbacks` section. It is also based on the module `chaosistio.fault.actions` but the `function` is `remove_abort_fault`. The arguments are basically the same as those we used to add the abort fault.

All in all, we are adding an abort fault through the action, and we are removing that same abort fault during the rollback phase.

Let's run this experiment and see what happens.

```
1 chaos run chaos/network-rollback.yaml
```

The output, without timestamps, is as follows.

```
1  [... INFO] Validating the experiment's syntax
2  [... INFO] Experiment looks valid
3  [... INFO] Running experiment: What happens if we abort responses
4  [... INFO] Steady state hypothesis: The app is healthy
5  [... INFO] Probe: app-responds-to-requests
6  [... INFO] Probe: app-responds-to-requests
7  [... INFO] Probe: app-responds-to-requests
8  [... INFO] Probe: app-responds-to-requests
9  [... INFO] Probe: app-responds-to-requests
10 [... INFO] Steady state hypothesis is met!
11 [... INFO] Action: abort-failure
12 [... INFO] Pausing after activity for 1s...
13 [... INFO] Steady state hypothesis: The app is healthy
14 [... INFO] Probe: app-responds-to-requests
15 [... INFO] Probe: app-responds-to-requests
16 [... CRITICAL] Steady state probe 'app-responds-to-requests' is not in the given tol\
17 erance so failing this experiment
18 [... INFO] Let's rollback...
19 [... INFO] Rollback: remove-abort-failure
20 [... INFO] Action: remove-abort-failure
21 [... INFO] Experiment ended with status: deviated
22 [... INFO] The steady-state has deviated, a weakness may have been discovered
```

The result is almost the same as before. The initial probes were successful, the action added the abort fault, and one of the after-action probes failed. All that is as expected, and as it was when we run the previous experiment.

What's new this time is the rollback. The experiment is failing just as it was failing before, but this time, there is the rollback action to remove whatever we did through actions. We added the abort

failure, and then we removed it. And to confirm that we successfully rolled back the change, we're going to send yet another 10 requests to the same address as before.

```
for i in {1..10}; do
    curl -H "Host: repeater.acme.com" \
        "http://$INGRESS_HOST?addr=http://go-demo-8"
done
```

This time the output is a stream of `Version: 0.0.1; Release: unknown` messages confirming that all the requests were successful. We did not improve our application just yet, but we did manage to undo the damage created during the experiment.

Let's describe the Virtual Service and double-check that everything indeed looks OK.

```
kubectl --namespace go-demo-8 \
    describe virtualservice go-demo-8
```

The output, limited to the relevant parts, is as follows.

```
...
Spec:
  Hosts:
    go-demo-8
  Http:
    Route:
      Destination:
        Host:  go-demo-8
        Port:
          Number:  80
        Subset:    primary
...
```

We can see that the `Fault.Abort` section is not there. Everything is normal. Whatever damage was done by the experiment action, was rolled back at the end of the experiment.

However, there's still at least one thing left for us to do. We still need to fix the application so that it can survive partial network failure. We'll do that next.

Making The Application Resilient To Partial Network Failures

How can we make our applications resilient to (some of) network issues? How can we deal with the fact that the network is not 100% reliable?

The last experiment is not creating complete outage but partial network failures. We could fix that in quite a few ways.

I already said that I will not show you how to solve issues by changing the code of your application. That would require examples in too many different languages. So, we'll look for a solution outside the application itself, probably inside Kubernetes. In this case, Istio is the logical place.

We're going to take a look at a modified version of our Virtual Service.

```
cat k8s/network/istio-repeater.yaml
```

The output is as follows.

```
---

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: repeater
spec:
  hosts:
  - repeater.acme.com
  - repeater
  gateways:
  - repeater
  http:
  - route:
    - destination:
        host: repeater
        subset: primary
        port:
          number: 80
    retries:
      attempts: 10
      perTryTimeout: 3s
      retryOn: 5xx
```

Most of the `VirtualService` definition is the same as before. It's called `repeater`, it has some hosts, and it is associated with the `repeater` Gateway. The `destination` is also the same. What is new is the `spec.http.retries` section.

We'll tell Istio Virtual Service to `retry` requests up to `10` times. If a request fails, it will be retried and, if it fails again, it will be retried again, and again, and again. The timeout is set to `3` seconds, and `retryOn` is set to `5xx`. That's telling Istio that it should retry failed requests up to `10` times with

a timeout of 3 seconds. It should retry them only if the response code is in the 500 range. If any of the 500 range response codes are received, Istio will repeat a request.

That should hopefully solve the problem of having our network failing sometimes.

Soon we'll check whether that's really really true. But, before we check whether our new definition works and before we even apply that definition, we'll take a quick look at the Envoy proxy documentation related to route filters. Istio uses Envoy, so the information about those `retryOn` codes can be found in its documentation.

If you are a Windows user, the `open` command might not work. If that's the case, please open the address manually in your favorite browser.

```
open https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/rou\
ter_filter#x-envoy-retry-on
```

You should see that the `5xx` code is one of many supported by Envoy. As you probably already know, Envoy is the proxy Istio injects into Pods as side-car containers. If, in the future, you want to fine-tune your Virtual Service definitions with other `retryOn` codes, that page gives you all the information you need. To be more precise, all router filters of the Envoy proxy are there.

Now that we know where to find the information about Envoy filters, we can go back to our improved definition of the Virtual Service and apply it. That should allow us to see whether our application is now resilient and highly available, even in case of partial failures of the network.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/network/istio-repeater.yaml
```

Now we're ready to re-run the same experiment and see whether our new setup works.

```
chaos run chaos/network-rollback.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we abort responses
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Action: abort-failure
[... INFO] Pausing after activity for 1s...
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Let's rollback...
[... INFO] Rollback: remove-abort-failure
[... INFO] Action: remove-abort-failure
[... INFO] Experiment ended with status: completed
```

We can see that the five initial probes were executed successfully and that the action injected abort failure set to 50%. After that, the same probes were re-run, and we can see that this time, all were successful. Our application is indeed retrying failed requests up to 10 times. Since approximately 50% of them are failing, up to ten repetitions are more than sufficient.

Everything is working. Our experiment was successful, and we can conclude that the `repeater` can handle partial network outages.

Increasing Network Latency

We saw how we can deal with network failures. To be more precise, we saw one possible way to simulate network failures, and one way to solve the adverse outcomes it produces. But it's not always going to be that easy. Sometimes the network does not fail, and requests do not immediately return 500 response codes. Sometimes there is a delay. Our applications might wait for responses for milliseconds, seconds, or even longer. How can we do deal with that?

Let's see what happens if we introduce a delay to our requests, or, to be more precise, if we introduce delay to responses of our requests. Is our application, in its current state, capable of handling that well and without affecting end users?

Let's take a look at yet another chaos experiment definition.

```
cat chaos/network-delay.yaml
```

This time, there are more than a few lines of changes.

We're keeping the part that aborts requests, and we added delays. So, instead of creating a separate experiment that would deal with delays, we're going to do both at the same time. We're keeping the abort failures, and we're adding delays. We're spicing it up, and that's not so far from the "real world" situation. When network failures are happening, some other requests might be delayed.

So let's take a look at the differences between this and the previous definition.

```
diff chaos/network-rollback.yaml \
    chaos/network-delay.yaml
```

The output is as follows.

```
2,3c2,3
< title: What happens if we abort responses
< description: If responses are aborted, the dependant application should retry and/\
or timeout requests
---
> title: What happens if we abort and delay responses
> description: If responses are aborted and delayed, the dependant application shoul\
d retry and/or timeout requests
20c20
<       timeout: 5
---
>       timeout: 15
53a54,69
> - type: action
>   name: delay
>   provider:
>     type: python
>     module: chaosistio.fault.actions
>     func: add_delay_fault
>     arguments:
>       virtual_service_name: go-demo-8
>       fixed_delay: 15s
>       routes:
>         - destination:
>             host: go-demo-8
```

```
>             subset: primary
>       percentage: 50
>       version: networking.istio.io/v1alpha3
>       ns: go-demo-8
70a87,100
> - type: action
>   name: remove-delay
>   provider:
>     type: python
>     func: remove_delay_fault
>     module: chaosistio.fault.actions
>     arguments:
>       virtual_service_name: go-demo-8
>       routes:
>         - destination:
>             host: go-demo-8
>             subset: primary
>       version: networking.istio.io/v1alpha3
>       ns: go-demo-8
```

We'll ignore the changes in the `title` and the `description` since they do not affect the output of an experiment.

We're increasing the `timeout` to `15` seconds. We're doing that not because I expect to have such a long timeout, but because it will be easier to demonstrate what's coming next.

We have two new actions.

The new action is using the function `add_delay_fault`, and the arguments are very similar to what we had before. It introduces a fixed delay of 15 seconds. So, when a request comes to this Virtual Service, it will be delayed for 15 seconds. And if we go back to the top, we can see that the timeout is also 15 seconds. So, our probe should fail because the delay of 15 seconds plus whatever number of milliseconds the request itself takes, is more than the timeout. The vital thing to note is that the `delay` is applied only to `50` percent of the requests.

Then we have a `rollback` action to remove that same delay.

All in all, we are adding a delay of 15 seconds on top of the abort faults. We are also introducing a rollback to remove that delay.

What do you think? Will the experiment fail, or will our application survive such conditions without affecting users (too much)? Let's take a look.

```
chaos run chaos/network-delay.yaml
```

The output is as follows.

```
[2020-03-13 23:45:33 INFO] Validating the experiment's syntax
[2020-03-13 23:45:33 INFO] Experiment looks valid
[2020-03-13 23:45:33 INFO] Running experiment: What happens if we abort and delay re\
sponses
[2020-03-13 23:45:34 INFO] Steady state hypothesis: The app is healthy
[2020-03-13 23:45:34 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:34 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:34 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:34 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:34 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:34 INFO] Steady state hypothesis is met!
[2020-03-13 23:45:34 INFO] Action: abort-failure
[2020-03-13 23:45:34 INFO] Action: delay
[2020-03-13 23:45:34 INFO] Pausing after activity for 1s...
[2020-03-13 23:45:35 INFO] Steady state hypothesis: The app is healthy
[2020-03-13 23:45:35 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:35 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:35 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:35 INFO] Probe: app-responds-to-requests
[2020-03-13 23:45:50 ERROR]   => failed: activity took too long to complete
[2020-03-13 23:45:50 WARNING] Probe terminated unexpectedly, so its tolerance could \
not be validated
[2020-03-13 23:45:50 CRITICAL] Steady state probe 'app-responds-to-requests' is not \
in the given tolerance so failing this experiment
[2020-03-13 23:45:50 INFO] Let's rollback...
[2020-03-13 23:45:50 INFO] Rollback: remove-abort-failure
[2020-03-13 23:45:50 INFO] Action: remove-abort-failure
[2020-03-13 23:45:50 INFO] Rollback: remove-delay
[2020-03-13 23:45:50 INFO] Action: remove-delay
[2020-03-13 23:45:50 INFO] Experiment ended with status: deviated
[2020-03-13 23:45:50 INFO] The steady-state has deviated, a weakness may have been d\
iscovered
```

The first five probes executed before the actions and confirmed that the initial state is as desired. The action introduced abort failure, just as before, as well as the delay.

In my case, and yours is likely different, the fourth probe failed. It could have been the first, or the second, or any other. But, in my case, the fifth one was unsuccessful. The message `activity took too long to complete` should be self-explanatory.

If we focus on the timestamp of the failed probe, we can see that there are precisely 15 seconds difference from the previous. In my case, the last successful probe started at 35 seconds, and then it failed at 50 seconds. The request was sent, and given that it has a timeout of 15 seconds, that's how much it waited for the response.

We can conclude that our application does not know how to cope with delays. What could be the fix for that?

Let's try to improve the definition of our Virtual Service. We'll output `istio-delay.yaml` and see what could be the change that might solve the problem with delayed responses.

```
cat k8s/network/istio-delay.yaml
```

The output is as follows.

```
---

apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: repeater
spec:
  hosts:
  - repeater.acme.com
  - repeater
  gateways:
  - repeater
  http:
  - route:
    - destination:
        host: repeater
        subset: primary
        port:
          number: 80
    retries:
      attempts: 10
      perTryTimeout: 2s
      retryOn: 5xx,connect-failure
    timeout: 10s
```

We still have the `retries` section with `attempts` set to `10` and with `perTryTimeout` set to `2` seconds. In addition, now we have `connect-failure` added to the `retryOn` values.

We are going to retry `10` times with a `2` second time out. We'll do that not only we have response codes in the five hundred range (`5xx`), but also when we have connection failures (`connect-failure`). That `2` seconds timeout is crucial in this case. If we send the request and it happens to be delayed, Istio Virtual Service will wait for `2` seconds, even though the delay is `15` seconds. It will abort that request after `2` seconds, and it will try again, and again, and again until it is successful, or until it

does it for 10 times. The total timeout is 10 seconds, so it might fail faster than that. It all depends on whether the timeout is reached first or the number of attempts.

Let's take a closer look at the `diff` between this definition and the previous one. That should give us a clearer picture of what's going on, and what changed.

```
diff k8s/network/istio-repeater.yaml \
    k8s/network/istio-delay.yaml
```

The output is as follows.

```
22,23c22,24
<       perTryTimeout: 3s
<       retryOn: 5xx
---
>       perTryTimeout: 2s
>       retryOn: 5xx,connect-failure
>     timeout: 10s
```

We can see that the `perTryTimeout` was reduced to `3` seconds, that we changed the `retryOn` codes to be not only `5xx` but also `connect-failure`, and that we introduced a timeout of `10` seconds. The retry process will be repeated up to 10 times and only for 10 seconds total.

Before we proceed, I must say that the timeout of 10 seconds is unrealistically high. Nobody should have 10 seconds as a goal. But, in this case, for the sake of simplifying everything, our expectation is that the application will be responsive, no matter whether we have partial delays or partial aborts within 10 seconds.

We're about to apply the new definition of this Virtual Service and re-run our experiment.

Will it work? Will our application be highly available and manage to serve our users, no matter whether requests and responses are aborted or delayed?

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/network/istio-delay.yaml

chaos run chaos/network-delay.yaml
```

The output of the latter command is as follows.

```
[2020-03-13 23:46:30 INFO] Validating the experiment's syntax
[2020-03-13 23:46:30 INFO] Experiment looks valid
[2020-03-13 23:46:30 INFO] Running experiment: What happens if we abort and delay re\
sponses
[2020-03-13 23:46:30 INFO] Steady state hypothesis: The app is healthy
[2020-03-13 23:46:30 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:30 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:30 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:30 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:30 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:30 INFO] Steady state hypothesis is met!
[2020-03-13 23:46:30 INFO] Action: abort-failure
[2020-03-13 23:46:31 INFO] Action: delay
[2020-03-13 23:46:31 INFO] Pausing after activity for 1s...
[2020-03-13 23:46:32 INFO] Steady state hypothesis: The app is healthy
[2020-03-13 23:46:32 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:38 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:44 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:46 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:48 INFO] Probe: app-responds-to-requests
[2020-03-13 23:46:54 INFO] Steady state hypothesis is met!
[2020-03-13 23:46:54 INFO] Let's rollback...
[2020-03-13 23:46:54 INFO] Rollback: remove-abort-failure
[2020-03-13 23:46:54 INFO] Action: remove-abort-failure
[2020-03-13 23:46:54 INFO] Rollback: remove-delay
[2020-03-13 23:46:54 INFO] Action: remove-delay
[2020-03-13 23:46:54 INFO] Experiment ended with status: completed
```

We can see that the execution of the five initial probes worked. After them, we run the two actions to add abort and delay failures, and then the same probes were executed successfully.

Pay attention to the timestamps of the after-action probes. In my case, if we focus on the timestamps from the first and the second post-action probe, we can see that the first one took around six seconds. There were probably some delays. Maybe there was one delay and one network abort. Or there could be some other combination. What matters is that it managed to respond within six seconds. In my case, the second request also took around 6 seconds, so we can guess that there were problems as well and that they were resolved. The rest of the probes were also successful even though they required varied durations to finish.

The responses to some, if not all the probes, took longer than usual. Nevertheless, they were all successful. Our application managed to survive delays and abort failures. Or, to be more precise, it survived in my case, and yours might be different.

After all the actions and the probes, the experiment rolled back the changes, and our system is back to the initial state. It's as if we never run the experiment.

Aborting All Requests

Sometimes we're unlucky, and we have partial failures with our network. But at other times, we're incredibly unfortunate, and the network is entirely down, or at least parts of the network related to one of the applications, or maybe a few of them, are down. What happens in that case? Can we recuperate from that? Can we make our applications resilient even in those situations?

We can run an experiment that will validate what happens in such a situation. It's going to be an easy one since we already have a very similar experiment.

```
cat chaos/network-abort-100.yaml
```

Given that experiment is almost the same as the one before, and that you'd have a tough time spotting the difference, we'll jump straight into the `diff` of the two.

```
diff chaos/network-rollback.yaml \
    chaos/network-abort-100.yaml
```

The output is as follows.

```
51c51
<       percentage: 50
---
>       percentage: 100
```

The difference is in the percent of abort failures. Instead of aborting 50% of the requests, we're going to abort all those destined for `go-demo-8`.

I'm sure that you know the outcome. Nevertheless, let's run the experiment and see what's happens.

```
chaos run chaos/network-abort-100.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we abort responses
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Action: abort-failure
[... INFO] Pausing after activity for 1s...
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... CRITICAL] Steady state probe 'app-responds-to-requests' is not in the given tol\
erance so failing this experiment
[... INFO] Let's rollback...
[... INFO] Rollback: remove-abort-failure
[... INFO] Action: remove-abort-failure
[... INFO] Experiment ended with status: deviated
[... INFO] The steady-state has deviated, a weakness may have been discovered
```

We modified the action to abort 100% of network requests destined to `go-demo-8`. As expected, the first probe failed. That's normal. Our application tried and tried and tried and tried. But after 10 times, it gave up, and we did not receive the response. It's a complete failure of the network associated with the `go-demo-8` API.

How should I fix that?

I will not show you how to fix that situation because the solution should most likely not be applied inside Kubernetes, but on the application level. In that scenario, assuming that we have other processes in place that deal with infrastructure when the network completely fails, it will be recuperated at one moment. We cannot expect Kubernetes and Istio and software around our applications to fix all of the problems. And this is the case where the design of our applications should be able to handle it.

Let's say that your frontend application is accessible, but that the backend is not. If, for example, your frontend application cannot, under any circumstance, communicate with the backend application, it should probably show a message like "shopping cart is currently not available, but feel free to browse our products" because they go to different backend applications. That's why we like microservices. The smaller the applications are, the smaller the scope of an issue. Or maybe your frontend application is not accessible, and then you would serve your users some static version of your frontend. There can be many different scenarios, and we won't go through them.

Similarly, there are many things that we can do to limit the blast radius. We won't go into those either

because most of the solutions for a complete collapse are related to how we code our applications, not what we do outside those applications. So think about how you would design your application to handle a total failure, similar to the one we just demonstrated through yet another chaos experiment.

Simulating Denial Of Service Attacks

Another scenario that could happen, among many others, is that we might be under attack. Somebody, intentionally or unintentionally, might be creating a Denial of Service Attack (DoS Attack). What that really means is that our applications, or even the whole cluster, might be under extreme load. It might be receiving such a vast amount of traffic that our infrastructure cannot handle it. It is not unheard of, not to say common, for a whole system to collapse when under a DoS attack. To be more precise, it is likely that a system will collapse if we don't undertake some precautionary measures.

Before we simulate such an attack, let's confirm that our application works more or less correctly when serving an increased number of requests. We're not going to go through simulating millions of requests. That would be out of the scope of this book. We're going to do a "poor man" equivalent of a Denial of Service Attack.

First, we'll put our application under test and see what happens if we keep sending 50 concurrent requests for, let's say, 20 seconds.

We'll use a tool called Siege that will be running as a container in a Pod.

Let's see what happens.

```
kubectl --namespace go-demo-8 \
    run siege \
    --image yokogawa/siege \
    --generator run-pod/v1 \
    -it --rm \
    -- --concurrent 50 --time 20S "http://go-demo-8"
```

We created a Pod called `siege` in the `go-demo-8` Namespace. It is based on the image `yokogawa/siege`. We used the `-it` argument (interactive, terminal), and we used `--rm` so that the Pod is deleted after the process is the only container inside that Pod is terminated. All those are uneventful. The interesting part of that command is the arguments we passed to the main process in the container.

The `--concurrent=50` and `--time 20S` argument tells Siege to run fifty concurrent requests for 20 seconds. The last argument is the address where Siege should be sending requests. This time we're skipping the `repeater` and sending them directly to `go-demo-8`.

Think of that command, and Siege in general, as being a very simple way to do performance testing. Actually, it's not even testing. We send a stream of concurrent requests, 50 to be more precise, for 20 seconds. I wanted us to confirm that the application can handle a high number of requests before

we jump into testing the behavior of the system when faced with a simulation of a Denial of Service Attack.

The output, limited to the relevant parts, is as follows.

```
...
Transactions:             1676 hits
Availability:            91.94 %
Elapsed time:            19.21 secs
Data transferred:         0.05 MB
Response time:            0.01 secs
Transaction rate:        87.25 trans/sec
Throughput:               0.00 MB/sec
Concurrency:              1.07
Successful transactions:  1676
Failed transactions:       147
Longest transaction:      0.08
Shortest transaction:     0.00
 ...
```

After twenty seconds, plus whatever time was needed to pull the image and run it, the siege ended. In my case, and yours will be different, I can see that `1676` hits were made. Most of them were successful. In my case, there was an availability of `91.94 %`. There were some failed transactions because this is not a robust solution. It's a demo application. Some were bound to fail, but ignore that. What matters is that most of the requests were successful. In my case, there were `1676` successful and `147` failed transactions.

Before we continue, let's take a quick look at the code of the application.

Don't worry if you're not proficient in Go. We're just going to observe something that could be useful to simulate Denial of Service Attacks.

```
cat main.go
```

The output, limited to the relevant parts, is as follows.

```
package main

import (
...
        "golang.org/x/time/rate"
...
)

...
var limiter = rate.NewLimiter(5, 10)
var limitReachedTime = time.Now().Add(time.Second * (-60))
var limitReached = false
...
func RunServer() {
...
        mux.HandleFunc("/limiter", LimiterServer)
...
}
...
func LimiterServer(w http.ResponseWriter, req *http.Request) {
        logPrintf("%s request to %s\n", req.Method, req.RequestURI)
        if limiter.Allow() == false {
                logPrintf("Limiter in action")
                http.Error(w, http.StatusText(500), http.StatusTooManyRequests)
                limitReached = true
                limitReachedTime = time.Now()
                return
        } else if time.Since(limitReachedTime).Seconds() < 15 {
                logPrintf("Cooling down after the limiter")
                http.Error(w, http.StatusText(500), http.StatusTooManyRequests)
                return
        }
        msg := fmt.Sprintf("Everything is OK\n")
        io.WriteString(w, msg)
}
...
```

We're going to simulate Denial of Service Attacks. For that, the application uses a Go library called `rate`. Further on, we have the `limiter` variable set to a `NewLimiter(5, 10)`. That means that it limits the application to have only five requests, with a burst of ten. No "real" application would ever be like that. But, we're using that to is simulate what happens if the number of requests is above the limit of what the application can handle. There are always limits of any application, we're just forcing this one to be very low.

Our applications should scale up and down automatically. But if we have a sudden drastic increase in the number of requests, that might produce some adverse effects. The application might not be able to scale up that fast.

Then we have the `LimiterServer` function that handles requests coming to the `/limiter` endpoint. It checks whether we reached that limit of five requests and, if we did, it sends `500` response code. Also, there is the additional logic that blocks all other requests for fifteen seconds after the limit is reached.

All in all, we're simulating the situation where our application reached the limit of what it can handle. And if it does, then it becomes unavailable for `15` seconds. It's a simple code that simulates what happens when an application starts receiving much more traffic than it can handle. If a replica of this app receives more than five simultaneous requests, it will be unavailable for fifteen seconds. That's (roughly) what would happen with requests when the application is under Denial of Service Attacks.

We'll execute another siege, but this time to the endpoint `/limiter`.

```
kubectl --namespace go-demo-8 \
    run siege \
    --image yokogawa/siege \
    --generator run-pod/v1 \
    -it --rm \
    -- --concurrent 50 --time 20S "http://go-demo-8/limiter"
```

The output, limited to the relevant parts, is as follows.

```
...
Transactions:              1845 hits
Availability:             92.02 %
Elapsed time:             19.70 secs
Data transferred:          0.04 MB
Response time:             0.01 secs
Transaction rate:         93.65 trans/sec
Throughput:                0.00 MB/sec
Concurrency:               1.12
Successful transactions:     20
Failed transactions:        160
Longest transaction:       0.09
Shortest transaction:      0.00
...
```

We can see that this time, the number of successful transactions is only `20`. It's not only five successful transactions, as you would expect, because we have multiple replicas of this application.

The exact number doesn't matter. What is important is that we can see that the number of successful transactions is much lower than before. In my case, that's only `20`. As a comparison, the first execution of the siege produced, in my case, `1676` successful transactions.

Running Denial Of Service Attacks

Now that we are familiar with Siege, and that we saw a "trick" baked in the `go-demo-8` app that allows us to limit the number of requests the application can handle, we can construct a chaos experiment that will check how the application behaves when under Denial of Service Attack.

Let's take a look at yet another chaos experiment definition.

```
cat chaos/network-dos.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we abort responses
description: If responses are aborted, the dependant application should retry and/or\
 timeout requests
tags:
- k8s
- pod
- deployment
- istio
configuration:
  ingress_host:
      type: env
      key: INGRESS_HOST
steady-state-hypothesis:
  title: The app is healthy
  probes:
  - type: probe
    name: app-responds-to-requests
    tolerance: 200
    provider:
      type: http
      timeout: 5
      verify_tls: false
      url: http://${ingress_host}?addr=http://go-demo-8/limiter
      headers:
        Host: repeater.acme.com
```

```
method:
- type: action
  name: abort-failure
  provider:
    type: process
    path: kubectl
    arguments:
    - run
    - siege
    - --namespace
    - go-demo-8
    - --image
    - yokogawa/siege
    - --generator
    - run-pod/v1
    - -it
    - --rm
    - --
    - --concurrent
    - 50
    - --time
    - 20S
    - "http://go-demo-8/limiter"
  pauses:
    after: 5
```

We have a steady-state hypothesis which validates that our application does respond with `200` on the endpoint `/limiter`. Further on, we have an action with the `type` of the `provider` set to `process`. That is the additional reason why I'm showing you that definition. Besides simulating Denial of Service and how to send increased number of concurrent requests to our applications, I'm using this opportunity to explore yet another `provider` type.

The `process` provider allows us to execute any command. That is very useful in cases when none of the Chaos Toolkit plugins will enable us to do what we need.

We can always accomplish goals that are not available through plugins by using the `process` provider, which can execute any command. It could be a script, a Shell command, or anything else, as long as it is executable. In this case, the `path` is `kubectl` (a command), followed by a list of `arguments`. Those are the same we just executed manually. We'll be sending `50` concurrent requests for `20` seconds to the `/limiter` endpoint.

Let's run this experiment and see what happens.

```
chaos run chaos/network-dos.yaml
```

```
[2020-03-13 23:51:28 INFO] Validating the experiment's syntax
[2020-03-13 23:51:28 INFO] Experiment looks valid
[2020-03-13 23:51:28 INFO] Running experiment: What happens if we abort responses
[2020-03-13 23:51:28 INFO] Steady state hypothesis: The app is healthy
[2020-03-13 23:51:28 INFO] Probe: app-responds-to-requests
[2020-03-13 23:51:28 INFO] Steady state hypothesis is met!
[2020-03-13 23:51:28 INFO] Action: abort-failure
[2020-03-13 23:51:52 INFO] Pausing after activity for 5s...
[2020-03-13 23:51:57 INFO] Steady state hypothesis: The app is healthy
[2020-03-13 23:51:57 INFO] Probe: app-responds-to-requests
[2020-03-13 23:51:57 CRITICAL] Steady state probe 'app-responds-to-requests' is not \
in the given tolerance so failing this experiment
[2020-03-13 23:51:57 INFO] Let's rollback...
[2020-03-13 23:51:57 INFO] No declared rollbacks, let's move on.
[2020-03-13 23:51:57 INFO] Experiment ended with status: deviated
[2020-03-13 23:51:57 INFO] The steady-state has deviated, a weakness may have been d\
iscovered
```

We can see that, after the initial probe was successful, we executed an action that ran the `siege` Pod. After that, the probe ran again, and it failed. Our application failed to respond because it was under heavy load, and it collapsed. It couldn't handle that amount of traffic. Now, the amount of traffic was ridiculously low, and that's why we're simulating DoS attack. But, in a "real world" situation, you would send high volume, maybe thousands or hundreds of thousands of concurrent requests, and see whether your application is responsive after that. In this case, we were cheating by configuring the application to handle a very low number of requests.

We can see that, in this case, the application cannot handle the load. The experiment failed.

The output in front of us is not very descriptive. We probably wouldn't be able to deduce the cause of the issue just by looking at it. Fortunately, that's only the list of events and their statuses, and more information is available. Every time we run an experiment, we get `chaostoolkit.log` file that stores detailed logs of what happened in case we need additional information. Let's take a look at it.

```
cat chaostoolkit.log
```

The output is too big to be presented in a book, so I'll let you explore it from your screen. You should see the (poorly formatted) output from `siege`. It gives us the same info as when we run it manually.

All in all, if you need more information, you can always find it in `chaostoolkit.log`. Think of it as debug info.

What would be the fix for this situation?

If you're waiting for me to give you the answer, you're out of luck. Just like the end of the previous chapter, I have a task for you. You're getting yet another homework.

Try to figure out how to would handle the situation we explored through the last experiment.

I will give you just a small tip that might help you know what to look for.

In Istio, we can use circuit breakers to limit the number of requests coming to an endpoint. In case of a Denial of Service Attack, or a sudden increase in the number of requests, we can use circuit breakers in Istio, or almost any other service mesh, to control what is the maximum number of concurrent requests that an application should receive.

Now it's your turn. Do the homework. Explore circuit breakers, and try to figure out how you would implement them for your applications. Use a chaos experiment to confirm that it fails before the changes and that it passes after. The goal is to figure out how to prevent that situation from becoming a disaster?

I know that your application has a limit. Every application does. How will you handle sudden outburst of requests that is way above what your app can handle at any given moment?

Destroying What We Created

That's it. Another chapter is finished, and now it's time for us to destroy the application by doing the same thing as before.

```
cd ..

kubectl delete namespace go-demo-8
```

We went out of the repository, and then we deleted the whole `go-demo-8` Namespace.

What comes next is your choice. Destroy the cluster if you do want to take a break. You will find instructions in the same Gists that you used to create it. Or keep the cluster and move forward right away. It's up to you.

Draining And Deleting Nodes

So far, we've been unleashing chaos on our applications and on networking. That's been very enlightening, at least to me. But there is still a lot that we should and that we can do.

One big topic that we haven't touched yet is related to nodes. What happens if there is something wrong with the nodes of our cluster? What happens if suddenly all the applications running in a node disappear? What happens if a node goes down or if it's destroyed? Those are the subjects we'll explore in this chapter. We're going to try to figure out how to create chaos that will generate issues related to nodes of our cluster.

Gist With The Commands

As you already know, every section that has hands-on exercises is accompanied by a Gist that allows you to copy and paste stuff instead of bothering to type. This chapter is not an exception.

All the commands from this chapter are available in the 06-node.sh[57] Gist.

Creating A Cluster

Just like always, we will have to create a cluster, unless you kept the one from before.

Unlike other chapters, there is a potential problem that we might face. It is related to the fact that we will not be able to run chaos experiments targeting nodes of our cluster on Minikube or Docker Desktop. If we drain or delete a node and that node is the only one in our cluster, then bad things will happen. In a "real" cluster like Azure, Google, AWS, DigitalOcean, or even on-prem, that shouldn't be a problem since we'd have multiple nodes, and we can scale our cluster up and down. But that is impossible, at least to my knowledge, with Minikube and Docker Desktop. Since they are a single node cluster, we will not be able to run experiments targeting nodes.

If you are a **Minikube** or **Docker Desktop** user, try to create a Kubernetes cluster somewhere else. As long as that cluster can run on multiple nodes, it should be fine. I prepared Gists to create a cluster in AWS (EKS), Google (GKE), Azure (AKS), so you can use those if you have an account in one of those providers. Alternatively, you can roll out your own. All the examples should work as long as that is not a Docker Desktop or Minikube cluster.

[57] https://gist.github.com/bc334351b8f5659e903de2a6eb9e3079

If, on the other hand, you cannot (or don't want to) run a Kubernetes cluster anywhere else, then the only thing you can do is to observe what I'm showing without doing the same actions yourself.

The good news is that this is likely the only chapter that has the restriction that we cannot use Docker Desktop or Minikube.

Once you create a cluster, you'll need the environment variable `INGRESS_HOST`. The Gists already have the instructions on how to define it. If you're creating a cluster on your own, make sure that it contains the address of Istio Gateway through which we can access applications.

Finally, there is a big warning.

Do NOT use a real production cluster.

I had to warn you, even though I'm sure you're having a demo cluster for the exercises in this book. What we're about to do will be very destructive. You might not be able to recuperate from the mayhem we are going to create. So make sure that the cluster is disposable.

Gists with the commands to create and destroy a Kubernetes cluster are as follows.

- GKE with Istio: gke-istio.sh[58]
- EKS with Istio: eks-istio.sh[59]
- AKS with Istio: aks-istio.sh[60]

Let's move on and deploy our demo application.

Deploying The Application

We need to deploy the demo application. This should be straightforward because we already did it a couple of times before. It's going to be the same application we used in the previous chapter, so we're going to go through it fast.

```
1 cd go-demo-8
2
3 git pull
```

We entered into the `go-demo-8` directory with the local copy of the repository, and we pulled the latest version of the code.

[58] https://gist.github.com/924f817d340d4cc52d1c4dd0b300fd20
[59] https://gist.github.com/3989a9707f80c2faa445d3953f18a8ca
[60] https://gist.github.com/c512488f6a30ca4783ce3e462d574a5f

```
kubectl create namespace go-demo-8

kubectl label namespace go-demo-8 \
    istio-injection=enabled
```

We created the Namespace `go-demo-8`, and we added the label `istio-injection=enabled`. From now on, Istio will be injecting proxy sidecars automatically.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/app-db

kubectl --namespace go-demo-8 \
    rollout status deployment go-demo-8
```

We applied the definitions from the `k8s/app-db` directory, and we waited until the application rolled out.

```
curl -H "Host: go-demo-8.acme.com" \
    "http://$INGRESS_HOST"
```

Finally, we sent a request to our application to confirm that it is accessible.

You might have noticed that we did not deploy the `repeater`. We don't need it. Other than that, it's still the same `go-demo-8` API and MongoDB.

That's about it. Now we can experiment with nodes of our cluster.

Draining Worker Nodes

We're going to try to drain everything from a random worker node.

Why do you think we might want to do something like that? One possible reason for doing that is in upgrades. The draining process is the same as the one we are likely using to upgrade our Kubernetes cluster.

Upgrading a Kubernetes cluster usually involves a few steps. Typically, we'd drain a node, we'd shut it down, and we'd replace it with an upgraded version of the node. Alternatively, we might upgrade a node without shutting it down, but that would be more appropriate for bare-metal servers that cannot be destroyed and created at will. Further on, we'd repeat the steps. We'd drain a node, we'd shut it down, and we'd create a new one based on an upgraded version. That would continue over and over again, one node after another, until the whole cluster is upgraded. The process is often called rolling updates (or rolling upgrades) and is employed by most Kubernetes distributions.

We want to make sure nothing wrong happens while or after upgrading a cluster. To do that, we're going to design an experiment that would perform the most critical step of the process. It will drain

a random node, and we're going to validate whether our applications are just as healthy before, as after that action.

If you're not familiar with the expression, draining means removing everything from a node.

Let's take a look at yet another definition of an experiment.

```
cat chaos/node-drain.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we drain a node
description: All the instances are distributed among healthy nodes and the applicati\
ons are healthy
tags:
- k8s
- deployment
- node
configuration:
  node_label:
      type: env
      key: NODE_LABEL
steady-state-hypothesis:
  title: Nodes are indestructible
  probes:
  - name: all-apps-are-healthy
    type: probe
    tolerance: true
    provider:
      type: python
      func: all_microservices_healthy
      module: chaosk8s.probes
      arguments:
        ns: go-demo-8
method:
- type: action
  name: drain-node
  provider:
    type: python
    func: drain_nodes
```

```
    module: chaosk8s.node.actions
    arguments:
      label_selector: ${node_label}
      count: 1
      pod_namespace: go-demo-8
      delete_pods_with_local_storage: true
  pauses:
    after: 1
```

The experiment has the typical `version`, `title`, `description`, and `tags`. We're not going to go through them.

Further on, in the `configuration` section, there is the `node_label` variable that will use the value of environment variable `NODE_LABEL`. I will explain why we need that variable in a moment. For now, just remember, there is a variable `node_label`.

Then we have the steady-state hypothesis. It has a single probe `all-apps-are-healthy`, which, as the name suggests, expects all applications to be healthy. We already used the same hypothesis, so there should be no need to explore it in more detail. It validates whether everything running in the `go-demo-8` Namespace is healthy.

The only `action` is in the `method` section, and it uses the function `drain_nodes`. The name should explain what it does. The function itself is available in `chaosk8s.node.actions` module, which is available through the Kubernetes plugin that we installed at the very beginning.

The action has a couple of arguments. There is the `label_selector` set to the value of the variable `${node_label}`. So, we are going to tell the system what are the nodes that are eligible for draining. Even though all your nodes are the same most of the time, that might not always be the case. Through that argument, we can select which nodes are eligible and which are not.

Further on, we have the `count` argument set to `1`, meaning that only one node will be drained.

There is also `pod_namespace`. That one might sound weird since we are not draining any Pods. We are draining nodes. Even though it might not be self-evident, that argument is instrumental. It tells the action to select a random node among those that have at least one Pod running in that Namespace. So, it will choose a random node among those where Pods inside the `go-demo-8` Namespace are running. That way, we can check what happens to the applications in that Namespace when one of the servers where they are running is gone.

Finally, we are going to pause for one second. That should be enough for us to be able to validate whether our applications are healthy soon after one of the nodes is drained.

Before we run that experiment, let me go back to the `node_label` variable.

We need to figure out what are the labels of the nodes of our cluster, and we can do that by describing them.

```
kubectl describe nodes
```

The output, limited to the relevant parts, is as follows.

```
...
Labels: beta.kubernetes.io/arch=amd64
        beta.kubernetes.io/fluentd-ds-ready=true
        beta.kubernetes.io/instance-type=n1-standard-4
        beta.kubernetes.io/os=linux
        cloud.google.com/gke-nodepool=default-pool
        cloud.google.com/gke-os-distribution=cos
        failure-domain.beta.kubernetes.io/region=us-east1
        failure-domain.beta.kubernetes.io/zone=us-east1-b
        kubernetes.io/arch=amd64
        kubernetes.io/hostname=gke-chaos-default-pool-2686de6b-nvzp
        kubernetes.io/os=linux
...
```

If you used one of my Gists to create a cluster, there should be only one node. If, on the other hand, you are using your own cluster, there might be others. As a result, you might see descriptions of more than one node.

In my case, I can use the `beta.kubernetes.io/os=linux` label as a node selector. If you have the same one, you can use it as well. Otherwise, you must find a label that identifies your nodes. That's the reason why I did not hard-code the selector. I did not want to risk the possibility that, in your cluster, that label might be different or that it might not even exist.

All in all, please make sure that you specify the correct label in the command that follows.

```
export NODE_LABEL="beta.kubernetes.io/os=linux"
```

We exported the `NODE_LABEL` variable with the label that identifies the nodes eligible for draining through experiments.

We are almost ready to run the experiment. But, before we do, stop for a moment and think what will be the result? Remember, we are running a cluster with a single node, at least if you used one of my Gists. What will be the outcome when we drain a single node? Will it be drained? The answer might seem obvious, but it probably isn't. To make it easier to answer that question, you might want to take another look at the Gist you used to create the cluster unless you rolled it out by yourself. Now, after you examined the Gist, think again. You might even need to look at the things running inside the cluster and examine them in more detail.

Let's say that you predicted the output of the experiment. Let's see whether you're right.

```
chaos run chaos/node-drain.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we drain a node
[... INFO] Steady state hypothesis: Nodes are indestructible
[... INFO] Probe: all-apps-are-healthy
[... INFO] Steady state hypothesis is met!
[... INFO] Action: drain-node
[... ERROR]   => failed: chaoslib.exceptions.ActivityFailed: Failed to evict pod ist\
io-ingressgateway-8577f4c6f8-xwcpb: {"kind":"Status","apiVersion":"v1","metadata":{}\
,"status":"Failure","message":"Cannot evict pod as it would violate the pod's disrup\
tion budget.","reason":"TooManyRequests","details":{"causes":[{"reason":"DisruptionB\
udget","message":"The disruption budget ingressgateway needs 1 healthy pods and has \
1 currently"}]},"code":429}
[... INFO] Pausing after activity for 1s...
[... INFO] Steady state hypothesis: Nodes are indestructible
[... INFO] Probe: all-apps-are-healthy
[... ERROR]   => failed: chaoslib.exceptions.ActivityFailed: the system is unhealthy
[... WARNING] Probe terminated unexpectedly, so its tolerance could not be validated
[... CRITICAL] Steady state probe 'all-apps-are-healthy' is not in the given t... th\
is experiment
[... INFO] Let's rollback...
[... INFO] No declared rollbacks, let's move on.
[... INFO] Experiment ended with status: deviated
[... INFO] The steady-state has deviated, a weakness may have been discovered
```

According to the output, the initial probe passed. All applications in the `go-demo-8` Namespace are healthy. That's a good thing.

Further on, the action was executed. It tried to drain a node, and it failed miserably.

It might have failed for a reason you did not expect.

We drained a node hoping to see what effect that produces on the applications in the `go-demo-8` Namespace. But, instead of that, we got an error stating that the node cannot be drained at all. It could not match the disruption budget of the `istio-ingressgateway`.

The Gateway is configured to have a disruption budget of `1`. That means that there must have at least one Pod running at any given moment. We, on the other hand, made a colossal mistake of deploying Istio without going into details. As a result, we are having a single replica of the Gateway.

All in all, the Gateway, in its current form, is having one replica. However, it has the disruption budget that prevents the system from removing a replica without guaranteeing that at least one Pod

is always running. That's a good thing. That design decision of Istio is correct because the Gateway should be running at any given moment. The bad thing is what we did. We installed Istio, or at least I told you how to install Istio, in a terrible way. We should have known better, and we should have scaled that Istio component to at least `2` or more replicas. We're going to do that soon. But, before that, we have a bigger problem on our plate.

We are running a single-node cluster. Or, to be more precise, you're running a single-node cluster if you used my instructions from one of the Gists. It will do us no good to scale Istio components to multiple replicas if they're all going to run on that single node. That would result in precisely the same failure. The system could not drain the node because that would mean that all the replicas of Istio components would need to be shut down, and they are configured with the disruption budget of `1`.

Uncordoning Worker Nodes

There are a couple of issues that we need to fix to get out of the bad situation we're in right now. But, before we start solving those, there is an even bigger problem created a few moments ago. I'm going to demonstrate that by retrieving the nodes.

```
kubectl get nodes
```

The output, in my case, is as follows (yours will be different).

```
NAME          STATUS                     ROLES  AGE VERSION
gke-chaos-... Ready,SchedulingDisabled <none> 13m v1.15.9-gke.22
```

You can see that the status of our single node is `Ready,SchedulingDisabled`. We run the experiment that failed to drain a node, and that is a process with two steps. First, the system disables scheduling on that node so that no new Pods are deployed. Then it drains that node by removing everything. The experiment managed to do the first step (it disabled scheduling), and then it failed on the second. As a result, now we have a cluster where we could not even schedule anything new. Based on our previous experience, we should have done it better from the start.

We should have created the rollback section that would un-cordon our nodes after the experiment, no matter whether it is successful or failed. So, our first mission will be to create the `rollbacks` section that will make sure that our cluster is in the correct state after the experiment. Or, in other words, we'll roll back whatever damage we do to the cluster through that experiment.

Let's take a look at yet another definition.

```
cat chaos/node-uncordon.yaml
```

You should see that there is a new section `rollbacks`. But, as in most other cases, we're going to `diff` this definition with the previous one and make sure that we see all the changes that we are making.

```
diff chaos/node-drain.yaml \
    chaos/node-uncordon.yaml
```

The output is as follows.

```
37a38,46
> rollbacks:
> - type: action
>   name: uncordon-node
>   provider:
>     type: python
>     func: uncordon_node
>     module: chaosk8s.node.actions
>     arguments:
>       label_selector: ${node_label}
```

We can see that the addition is only the `rollbacks` section with an action that relies on the `uncordon_node` function. Uncordon will undo the damage that we are potentially going to create by draining a node. So, if you think of draining as being cordoning or disabling scheduling plus draining, this rollback action will un-cordon or re-enable scheduling on that node. The rest will be taken by Kubernetes, which will be scheduling new Pods on that node.

The `rollbacks` section has a single argument `label_selector` that matches the label we're using in the cordoning action. It will un-cordon all the nodes with that label.

Now we can re-run the experiment.

```
chaos run chaos/node-uncordon.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we drain a node
[... INFO] Steady state hypothesis: Nodes are indestructible
[... INFO] Probe: all-apps-are-healthy
[... ERROR]   => failed: chaoslib.exceptions.ActivityFailed: the system is unhealthy
[... WARNING] Probe terminated unexpectedly, so its tolerance could not be validated
[... CRITICAL] Steady state probe 'all-apps-are-healthy' is not in the given toleran\
ce so failing this experiment
[... INFO] Let's rollback...
[... INFO] Rollback: uncordon-node
[... INFO] Action: uncordon-node
[... INFO] Experiment ended with status: failed
```

Just as before, it still fails because we did not solve the problem of not having enough nodes and not having enough replicas of our Istio components. The failure persists, and that's as expected. What we're getting now is the rollback action that will undo the damage created by the experiment. We can confirm that easily by outputting the nodes of the cluster.

```
kubectl get nodes
```

The output, in my case, is as follows.

```
NAME          STATUS ROLES  AGE VERSION
gke-chaos-... Ready  <none> 15m v1.15.9-gke.22
```

We can see that we still have the same single node but, this time, the status is `Ready`, while before it was `Ready,SchedulingDisabled`. We undid the damage created by the actions of the experiment.

We saw how we can un-cordon a node after cordoning it and after trying to drain it. Now, we can go into a more exciting part and try to figure out how to fix the problem. We cannot drain nodes. Therefore we cannot upgrade them since the process, in most cases, means draining first and upgrading second.

Making Nodes Drainable

Let's take a look at Istio Deployments.

```
kubectl --namespace istio-system \
    get deployments
```

The output is as follows.

```
NAME                 READY UP-TO-DATE AVAILABLE AGE
istio-ingressgateway 1/1   1          1         12m
istiod               1/1   1          1         13m
prometheus           1/1   1          1         12m
```

We can see that there are two components, not counting `prometheus`. If we focus on the `READY` column, we can see that they're all having one replica.

The two Istio components have a HorizontalPodAutoscaler (HPA) associated. They control how many replicas we'll have, based on metrics like CPU and memory usage. What we need to do is set the minimum number of instances to `2`.

Since the experiment revealed that `istio-ingressgateway` should have at least two replicas, that's the one we'll focus on. Later on, the experiment might reveal other issues. If it does, we'll deal with them then.

But, before we dive into scaling Istio, we are going to explore scaling the cluster itself. It would be pointless to increase the number of replicas of Istio components, as a way to solve the problem of not being able to drain a node, if that is the only node in a cluster. We need the Gateway not only scaled but also distributed across different nodes of the cluster. Only then can we hope to drain a node successfully if the Gateway is running in it. We'll assume that the experiment might shut down one replica, while others will still be running somewhere else. Fortunately for us, Kubernetes always does its best to distribute instances of our apps across different nodes. As long as it can, it will not run multiple replicas on a single node.

So, our first action is to scale our cluster. The problem is that scaling a cluster is not the same everywhere. Therefore, the commands to scale the cluster will differ depending on where you're running it.

To begin, we'll define an environment variable that will contain the name of our cluster.

Please replace `[...]` with the name of the cluster in the command that follows. If you're using one of my Gists, the name should be `chaos`.

```
export CLUSTER_NAME=[...] # e.g. `chaos`
```

From now on, the instructions will differ depending on the Kubernetes distribution you're running.

Please follow the instructions corresponding to your Kubernetes flavor. If you're not running your cluster inside one of the providers I'm using, you'll have to figure out the equivalent commands yourself.

Please execute the command that follows only if you are using **Google Kubernetes Engine (GKE)**. Bear in mind that, if you have a newly created account, the command might fail due to insufficient quotas. If that happens, follow the instructions in the output to request a quota increase.

```
gcloud container clusters \
    resize $CLUSTER_NAME \
    --zone us-east1-b \
    --num-nodes=3
```

Please execute the commands that follow only if you are using **AWS Elastic Kubernetes Service (EKS)**.

```
eksctl get nodegroup \
    --cluster $CLUSTER_NAME

export NODE_GROUP=[...] # Replace `[...]` with the node group

eksctl scale nodegroup \
    --cluster=$CLUSTER_NAME \
    --nodes 3 \
    $NODE_GROUP
```

Please execute the commands that follow only if you are using **Azure Kubernetes Service (AKS)**.

```
az aks show \
    --resource-group chaos \
    --name chaos \
    --query agentPoolProfiles

export NODE_GROUP=[...] # Replace `[...]` with the `name` (e.g., `nodepool1`)

az aks scale \
    --resource-group chaos \
    --name chaos \
    --node-count 3 \
    --nodepool-name $NODE_GROUP
```

Depending on where you're running your Kubernetes cluster, the process can take anything from seconds to minutes. We'll be able to continue once the cluster is scaled.

To be on the safe side, we'll confirm that everything is OK by outputting the nodes.

```
kubectl get nodes
```

The output, in my case, is as follows.

```
NAME          STATUS ROLES  AGE   VERSION
gke-chaos-... Ready  <none> 60s   v1.15.9-gke.22
gke-chaos-... Ready  <none> 60s   v1.15.9-gke.22
gke-chaos-... Ready  <none> 3h28m v1.15.9-gke.22
```

Repeat the previous command if there are less than three nodes with the status `Ready`.

In my case, you can see that there are three worker nodes. That does not include the control plane, which is out of reach when we're using GKE (like I do) and most other managed Kubernetes clusters.

Now that we have a cluster with multiple nodes, we can figure out how to scale Istio components. Or, to be more precise, we'll change the minimum number of replicas defined in the HPA associated with the Gateway.

Let's start by taking a quick look at what we have.

```
kubectl --namespace istio-system \
    get hpa
```

The output is as follows.

```
NAME                   REFERENCE                         TARGETS MINPODS MAXPODS REPLICA\
S AGE
istio-ingressgateway Deployment/istio-ingressgateway 6%/80%  1       5       1      \
  70s
istiod               Deployment/istiod               0%/80%  1       5       1      \
  62m
```

We can see that the minimum number of Pods of those HPAs is `1`. If we can change the value to `2`, we should always have two replicas of each while still allowing the components to scale to up to `5` replicas if needed.

Typically, we should create a new Istio manifest with all the changes we need. However, in the interest of moving fast, we'll apply a patch to that HPA. As I already mentioned, this book is not dedicated to Istio, and I'll assume that you'll check the documentation if you don't already know how to update it properly.

```
kubectl --namespace istio-system \
    patch hpa istio-ingressgateway \
    --patch '{"spec": {"minReplicas": 2}}'
```

Now that we modified Istio in our cluster, we're going to take another look at the HPAs.

```
kubectl --namespace istio-system \
    get hpa
```

The output is as follows.

```
NAME                 REFERENCE                         TARGETS MINPODS MAXPODS REPLICA\
S AGE
istio-ingressgateway Deployment/istio-ingressgateway 7%/80%  2       5       2      \
  13m
istiod               Deployment/istiod               0%/80%  1       5       1      \
  75m
```

We can see that `istio-ingressgateway` now has the minimum and the actual number of Pods set to `2`. If, in your case, the number of replicas is still `1`, the second Pod is still not up and running. If that's the case, wait for a few moments and repeat the command that retrieves all the HPAs.

Now let's take a quick look at the Pods in `istio-system`. We want to confirm that they're running on different nodes.

```
kubectl --namespace istio-system \
    get pods \
    --output wide
```

The output is as follows.

```
NAME                     READY STATUS  RESTARTS AGE   IP         NODE              \
...
istio-ingressgateway-... 1/1   Running 0        6m59s 10.52.1.3  gke-chaos-...-kqgk \
...
istio-ingressgateway-... 1/1   Running 0        16m   10.52.1.2  gke-chaos-...-kjkz \
...
istiod-...               1/1   Running 0        77m   10.52.0.14 gke-chaos-...-kjkz \
...
prometheus-...           2/2   Running 0        77m   10.52.0.16 gke-chaos-...-kjkz \
...
```

If we take a look at `istio-ingressgateway` Pods, we can see that they are running on different nodes. A quick glance tells us that they're all distributed across the cluster. Each of its replicas running on different servers.

Let's re-run our experiment and see what we are getting.

We are going to execute exactly the same experiment as before. It failed before due to a very unexpected reason. It was not unsuccessful because of the problems with `go-demo-8`, but it failed because of Istio itself. So let's see what we'll get now.

```
chaos run chaos/node-uncordon.yaml
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we drain a node
[... INFO] Steady state hypothesis: Nodes are indestructible
[... INFO] Probe: all-apps-are-healthy
[... INFO] Steady state hypothesis is met!
[... INFO] Action: drain-node
[... INFO] Pausing after activity for 1s...
[... INFO] Steady state hypothesis: Nodes are indestructible
[... INFO] Probe: all-apps-are-healthy
[... INFO] Steady state hypothesis is met!
[... INFO] Let's rollback...
[... INFO] Rollback: uncordon-node
[... INFO] Action: uncordon-node
[... INFO] Experiment ended with status: completed
```

Once the node was drained, the experiment waited for one second, and then it confirmed that the steady-state hypothesis is valid. It passed. So, this time, nodes seem to be drainable.

In the end, the experiment rolled back so that there is no permanent negative effect. We can confirm that by listing all the nodes.

```
kubectl get nodes
```

In my case, the output is as follows.

```
NAME          STATUS ROLES  AGE   VERSION
gke-chaos-... Ready  <none> 5m25s v1.15.9-gke.22
gke-chaos-... Ready  <none> 5m25s v1.15.9-gke.22
gke-chaos-... Ready  <none> 3h32m v1.15.9-gke.22
```

The status of all the nodes is `Ready`. That's awesome. Not only that we managed to make the nodes of our cluster drainable, but we also managed to roll back at the end of the process and make them all "normal" again.

Deleting Worker Nodes

After resolving a few problems, we are now able to drain nodes. We discovered those issues through experiments. As a result, we should be able to upgrade our cluster without doing something terribly wrong and, hopefully, without negatively affecting our applications.

Draining nodes is, most of the time, a voluntary action. We tend to drain our nodes when we choose to upgrade our cluster. So, the previous experiment was beneficial since it gives us the confidence

to upgrade the cluster without (much) fear. However, there is still something worse that can happen to our nodes.

More often than not, nodes will fail without our consent. They will not drain. They will just get destroyed, they will get damaged, they will go down, and they will be powered off. Bad things will happen to nodes, whether we like it or not.

Let's see whether we can create an experiment that will validate how our cluster behaves when such things happen.

As always, we're going to take a look at yet another experiment.

```
cat chaos/node-delete.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we delete a node
description: All the instances are distributed among healthy nodes and the applicati\
ons are healthy
tags:
- k8s
- deployment
- node
configuration:
  node_label:
      type: env
      key: NODE_LABEL
steady-state-hypothesis:
  title: Nodes are indestructible
  probes:
  - name: all-apps-are-healthy
    type: probe
    tolerance: true
    provider:
      type: python
      func: all_microservices_healthy
      module: chaosk8s.probes
      arguments:
        ns: go-demo-8
method:
- type: action
  name: delete-node
  provider:
```

```
    type: python
    func: delete_nodes
    module: chaosk8s.node.actions
    arguments:
      label_selector: ${node_label}
      count: 1
      pod_namespace: go-demo-8
  pauses:
    after: 10
```

We can see that we replaced the `draining-node` method with `delete-node`. There are a few other changes, so let's look at the `diff` that should allow us to see better what really changed when compared with the previous definition.

```
diff chaos/node-uncordon.yaml \
    chaos/node-delete.yaml
```

```
2c2
< title: What happens if we drain a node
---
> title: What happens if we delete a node
26c26
<   name: drain-node
---
>   name: delete-node
29c29
<     func: drain_nodes
---
>     func: delete_nodes
35d34
<       delete_pods_with_local_storage: true
37,46c36
<     after: 1
< rollbacks:
< - type: action
<   name: uncordon-node
<   provider:
<     type: python
<     func: uncordon_node
<     module: chaosk8s.node.actions
<     arguments:
<       label_selector: ${node_label}
```

```
26 ---
27 >     after: 10
```

We can see that the `title` and the `name` changed, but that's not important. The function is different. It is `delete_nodes`, instead of `drain_nodes`. As you can see, all the other parts of that action are the same because nothing else changed, except the `delete_pods_with_local_storage` which should be self-explanatory. In other words, draining and deleting a node takes exactly the same arguments. It's just a different function.

Another thing that changed is that we removed the `rollbacks` section. The previous rollback that un-cordons nodes is not there anymore. The reason is simple. We are not draining, and we are not cordoning nodes. We're deleting a node, and there is no rollback from that. Or, to be more precise, there could be a rollback, but we're not going to implement it. It is a similar situation like destroying a Pod. Except that, this time, we're not terminating a Pod. We're killing a node, and our cluster should be able to recuperate from that. We're not changing the desired state, and the cluster should recover from a loss of a node without us rolling back anything. At least, that should be the goal. All in all, there is no rollback.

Let's run this experiment and see what we're getting.

```
1 chaos run chaos/node-delete.yaml
```

The output is as follows.

```
1  [2020-03-17 22:59:26 INFO] Validating the experiment's syntax
2  [2020-03-17 22:59:26 INFO] Experiment looks valid
3  [2020-03-17 22:59:26 INFO] Running experiment: What happens if we delete a node
4  [2020-03-17 22:59:26 INFO] Steady state hypothesis: Nodes are indestructible
5  [2020-03-17 22:59:26 INFO] Probe: all-apps-are-healthy
6  [2020-03-17 22:59:27 INFO] Steady state hypothesis is met!
7  [2020-03-17 22:59:27 INFO] Action: delete-node
8  [2020-03-17 22:59:29 INFO] Pausing after activity for 10s...
9  [2020-03-17 22:59:39 INFO] Steady state hypothesis: Nodes are indestructible
10 [2020-03-17 22:59:39 INFO] Probe: all-apps-are-healthy
11 [2020-03-17 22:59:39 INFO] Steady state hypothesis is met!
12 [2020-03-17 22:59:39 INFO] Let's rollback...
13 [2020-03-17 22:59:39 INFO] No declared rollbacks, let's move on.
14 [2020-03-17 22:59:39 INFO] Experiment ended with status: completed
```

In your case, the output might show that the experiment failed. If that's the case, you were unlucky, and the node that hosted the `go-demo-8` DB was destroyed. We already discussed the problems with the database, and we're not going to revisit them here. You'll have to imagine that it worked and that your output matches mine.

We can see, at least from my output, that the initial steady-state hypothesis was met, and the action to delete a node was executed. Then we were waiting for 10 seconds, and after that, the node was destroyed. The post-action steady-state hypothesis was successful, so we can conclude that our applications running in `go-demo-8` Namespace are still functioning. They are healthy. Actually, that's not true. There are other problems, but we are not discovering those problems with this experiment. We're going to get to them later in one of the next sections. For now, at least according to the experiment, everything was successful.

All in all, we confirmed that all apps were healthy, and then we destroyed a node and confirmed that they're still healthy.

Let's take a look at the nodes.

```
kubectl get nodes
```

The output, in my case, is as follows.

```
NAME          STATUS ROLES  AGE   VERSION
gke-chaos-... Ready  <none> 7m25s v1.15.9-gke.22
gke-chaos-... Ready  <none> 3h34m v1.15.9-gke.22
```

Now we have two nodes, while before we had three. The cluster is not recuperating from this because we did not set autoscaling and quite a few other things. But the good news is that our app is still running.

There are a couple of things that you should note. First of all, we did not really destroy a node. We could do that, but we didn't. Instead, we removed the node from the Kubernetes cluster. So, from the Kubernetes perspective, the node is gone. However, the virtual machine is most likely still running, and you might want to delete that machine yourself. If you do wish to annihilate it, you can go to console or use the CLI from your favorite cloud provider and delete it. I will not provide instructions for that.

What matters is that this is one way to see what happens when a node is deleted or terminated. To be more precise, we saw what happens when a node is, for one reason or another, not registered by Kubernetes anymore. From the Kubernetes perspective, that node was gone. It's a separate question whether it is physically running or not. We do not know if what was running on that node is still there, but hidden from Kubernetes. That's the assignment for you.

Figure out what happens with the stuff running on that node. Think of it as homework.

Before we go on, let's check the pods and see whether they're all running.

```
kubectl --namespace go-demo-8 \
    get pods
```

The output, in my case, is as follows.

```
NAME              READY STATUS  RESTARTS AGE
go-demo-8-...     2/2   Running 2        3m35s
go-demo-8-...     2/2   Running 1        34s
go-demo-8-...     2/2   Running 0        33s
go-demo-8-db-...  2/2   Running 0        3m34s
```

We can see that the Pods in the `go-demo-8` Namespace are indeed running. We can see that, in my case, two of them are relatively young (`33s` and `34s`), meaning that they were probably running in the node that was removed from Kubernetes. They were distributed across the two healthy nodes. So our application is fine if we ignore the possibility that additional replicas might be running on the node that went rogue.

I have a second assignment for you. Create an experiment that will destroy a node instead of removing it from Kubernetes. Create an action that will shut it down.

I will not provide instructions on how to do that for two reasons. First of all, you should have assignments instead of just copying and pasting commands. The second reason lies in the complexity of doing something like that without knowing what your hosting provider is. I cannot provide the instructions as an experiment that works for all since it would have to differ from one provider to another. Your hosting provider might already be supported through one of the Chaos Toolkit modules. You'd need to install a plugin for your specific provider. Or you might be using one of the providers that are not supported through any of the plugins. If that's the case, you can accomplish the same thing by telling Chaos Toolkit to execute a command that will destroy a node. Later on, I'm going to show you how to run random commands. That's coming.

We're done for now. We saw that nothing wrong happens with our cluster if we remove a node (ignoring the DB). However, if you would remove one more node, then your cluster might not have enough capacity to host everything we need. We would be left with only one server. And if you would remove the third node, that would be a real problem.

The primary issue is that our cluster is not scalable. At least, not if you created it through one of the Gists. We're going to (briefly) discuss that next.

Destroying Cluster Zones

As I mentioned before, nothing happen if we destroy only one out of three nodes (again, ignoring the DB). But if we would continue doing that, bad things could happen.

Here goes yet another assignment. Figure out what would happens if we destroy a whole data center?

There are at least two critical things that we didn't do, but we should. First of all, we should have created our Kubernetes cluster with Cluster Autoscaler so that it automatically scales up and down depending on the traffic. In that case, not only that it would scale up and down to accommodate an increase and decrease in the workload, but when a node goes down, it would be recreated by Cluster Autoscaler. It would figure out that there is not enough capacity. Cluster Autoscaler itself

would solve fully (or partly) the problem that we could have encountered if we continued running the previous experiment and continued deleting nodes.

The second issue is that we are running a zonal cluster. If you followed my Gists, your cluster is running in a single zone, and that means that it is not fault-tolerant. If that zone (data center) goes down, we'd be lost. So the second change we should have done to our cluster is to make it regional. It should span multiple zones within the same region. It shouldn't run in different regions because that would increase latency unnecessarily. Every cloud provider, at least the big three, have a concept of a region, even though it might be called differently. By region, what I mean is a group of zones (data centers) that are close enough to each other so that there is no high latency, while they still operate as entirely separate entities. Failure of one should not affect the other. Or, at least, that's how it should be in theory.

So, we should make our cluster regional, and we should make it scalable. Since the steps differ from one provider to another, I will not show you how to create such a cluster, but I will provide the Gists. Please consult them to see how to create a better Kubernetes cluster in Google, Azure, or AWS.

- Regional and scalable GKE: https://gist.github.com/88e810413e2519932b61d81217072daf
- Regional and scalable EKS: https://gist.github.com/d73fb6f4ff490f7e56963ca543481c09
- Regional and scalable AKS: https://gist.github.com/b068c3eadbc4140aed14b49141790940

We'll start the next chapter by creating the cluster in that way. It will scale automatically, and it will be regional.

That's about it. There are many other things we could do with nodes, but we explored just enough for you to have at least the basic understanding of how chaos engineering applied to Kubernetes nodes works.

Destroying What We Created

Just as before, at the end of each chapter, we'll delete everything we did.

```
cd ..

kubectl delete namespace go-demo-8
```

We went one directory back, and then we deleted the Namespace.

Unlike other sections, this time, it is probably not optional whether to destroy the cluster. We removed a node in a cluster that is not auto-scalable, and that is a terrible idea. We could modify the cluster to be auto-scalable, and then also change it to be regional, but that would be too much work. Instead, make sure that this time, you do destroy the cluster, even if you're not taking a break. Starting from the next chapter, we will be creating a cluster in a much better and more resilient way by making it regional and making it scalable.

Creating Chaos Experiment Reports

We saw how we can rain destruction on many different levels.

We started small by destroying pods. Then we moved into destroying higher-level constructs and into measuring the health of our applications. We were also messing with the network and the nodes, and we did quite a few other things in between.

We're going to calm down for a while. We'll stop the chaos, or, at least, we will not create new types of destruction. Instead, we'll focus on figuring out how to create informative reports that can be distributed throughout the company. Instead of creating more mayhem, we'll figure out how we can create reports and how we can provide data to other teams that will inform them about the shortcomings of their applications and their clusters.

Gist With The Commands

As you already know, every section that has hands-on exercises is accompanied by a Gist that allows you to copy and paste commands instead of bothering to type. This chapter is not an exception.

All the commands from this chapter are available in the 07-reporting.sh[61] Gist.

Creating A Cluster

As always, we will need a cluster. However, this time, it is going to be a bit better defined than before. In the previous chapter, we gained some insights that should help us create a more robust cluster. It will be regional and scalable. Or, to be more precise, it will be like that if that's possible. If you're using Docker Desktop or Minikube, it will not be regional nor scalable because those are single-node clusters. Nobody's perfect.

Just remember that, no matter how you create the cluster, we will need the `INGRESS_PORT` variable with the Istio Ingress Gateway address. That, as you can probably guess, means that the cluster should have Istio up-and-running.

Gists with the commands to create and destroy a Kubernetes cluster are as follows.

- Docker Desktop with Istio: docker-istio.sh[62]

[61] https://gist.github.com/d81f114a887065a375279635a66ccac2
[62] https://gist.github.com/9a9752cf5355f1b8095bd34565b80aae

- Minikube with Istio: minikube-istio.sh[63]
- Regional and scalable GKE with Istio: gke-istio-full.sh[64]
- Regional and scalable EKS with Istio: eks-istio-full.sh[65]
- Regional and scalable AKS with Istio: aks-istio-full.sh[66]

Now that we have a cluster, we'll deploy our demo application. And, after that, we'll be able to run some experiments. This time, we will not run them for the sake of exploring new ways to destroy stuff. Instead, we'll run experiments so that we can gather some data which we'll use to generate reports.

Deploying The Application

The only thing left for us to do, before we dive into reporting, is to deploy the demo application. It'll be the same one we used before, so what follows will not be a mystery. Let's go through it fast.

We'll go to the `go-demo-8` directory and pull the latest version of the repository.

```
cd go-demo-8

git pull
```

Next, we'll create a Namespace called `go-demo-8` and add the label so that Istio knows that it should inject proxy sidecars automatically.

```
kubectl create namespace go-demo-8

kubectl label namespace go-demo-8 \
    istio-injection=enabled
```

We're going to apply the definition from the directory `k8s/app-full` and, once all the resources are created, we're going to wait until the Deployment of the API rolls out.

[63] https://gist.github.com/a5870806ae6f21de271bf9214e523b53
[64] https://gist.github.com/88e810413e2519932b61d81217072daf
[65] https://gist.github.com/d73fb6f4ff490f7e56963ca543481c09
[66] https://gist.github.com/b068c3eadbc4140aed14b49141790940

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/app-full

kubectl --namespace go-demo-8 \
    rollout status deployment go-demo-8
```

Finally, to be sure that it works fine, we're going to send a request to the newly deployed demo application.

```
curl -H "Host: repeater.acme.com" \
    "http://$INGRESS_HOST?addr=http://go-demo-8"
```

Now we are ready to generate some data based on experiments. Once we have the information we need, we're going to try to figure out how we can generate reports. After all, you'll need something like a PDF document to give to your manager and make him happy. More importantly, we want to generate something that you can give to the rest of the teams. It should contain, in an easy to digest format, information about what's wrong, what failed, and what was successful in chaos experiments.

Exploring Experiments Journal

Chaos Toolkit allows us to create a journal file with information generated during an experiment.

To generate data, we need to run an experiment. Let's take a look at the definition of a familiar one.

```
cat chaos/health-http.yaml
```

We already explored that experiment, so there's probably no need to go through it. Even if you forgot it, you should have enough experience to deduce what it does yourself. Remember, the subject of this chapter is not to learn new ways to define experiments and destroy stuff, but to create reports. For that, we need data or, if we switch to Chaos Toolkit terminology, we need a journal.

We'll run the experiment in a similar way as before, but with a twist.

```
chaos run chaos/health-http.yaml \
    --journal-path journal-health-http.json
```

We added a new argument `--journal-path`, set to the path to the file `journal-health-http.json`. That's where the journal data containing the information gathered throughout the experiment was stored. And, as you can guess from the file extension, that data was stored in the JSON format.

Let's take a look at the file.

```
cat journal-health-http.json
```

The output is too big to be presented in a book, and I will let you explore it from your screen. I don't think you need me to guide you through it. With your current experience, you should be able to deduce what each of the sections in the journal are.

I don't expect many people to want to read JSON output. It could be useful in its current form if we'd like machines to interpret it and, maybe, perform some additional actions. Machines love JSON, humans don't, at least not for reading.

So, if we're looking for a machine-readable output of the outcomes of an experiment, the journal is excellent. It is very easy to parse. But, in our case, that's not the final outcome we're looking for. We're yet to discover how we can convert this into something that is more user friendly and easier to read by humans. So, in the next section, we're going to try to figure out how to convert this JSON into a report, which could be in different formats. We're yet to see which one we're going to use.

Creating Experiment Report

Now that we have the journal with all the information, let's convert JSON into something more humanly readable. We're going to choose PDF, even though we could create some other formats. It could be, for example, HTML, which could be useful if we would like to publish it online. But, we're going to choose PDF, since I will imagine that the goal is to hand over a document, either to your colleagues or to managers.

Now we're getting to the potentially problematic part.

To create a report, we need to install quite a few dependencies, and they could be tricky to install. So, we're going to skip trying to figure out all the dependencies we might need to install and configure. Instead, we're going to run it as a container. To be more precise, we're going to run a process to create a report through Docker.

Make sure that you have Docker up or running on your laptop. If you don't have it already, go to the Docker Engine Overview[67] page and install the version for your platform.

Once Docker is up and running, we can create a container that will generate a report based on the journal file we created earlier.

The command we are about to execute will be slightly longer than we're used to. If that's too much for you and your colleagues, you can convert it into a script, so that it's a bit easier to run.

[67] https://docs.docker.com/install/

```
docker container run \
    --user $(id -u) \
    --volume $PWD:/tmp/result \
    -it \
    chaostoolkit/reporting \
     -- report \
     --export-format=pdf \
    journal-health-http.json \
    report.pdf
```

Lo and behold, a report was generated.

We executed `docker container run`.

The `--user` argument made sure that the output file is made by the same user as the one you're using.

Further on, we used `--value` to ensure that the report that we generated inside the container is available on our local hard disk after the container was destroyed. We mounted the current directory (`$PWD`) as the volume `/tmp/result`.

What also used `-it`, which is short for interactive and terminal. That way, we could see the output from that container on the screen.

The image used to create the container is `chaostoolkit/reporting`. That is the image created and maintained by the Chaos Toolkit community.

Finally, we specified a few arguments of the entry point command. The first one is `report` telling `chaos` what we want to create. The `--export-format` made sure that it is PDF (it could be something else). Further on, we set the location of the journal (`journal-health-http.json`), and the very last argument is the path where the report should be generated (`report.pdf`).

We can see from the output that the `report` was `generated as 'report.pdf'`.

Let's open the newly generated file and see how it looks like.

Remember, if you're a **Windows user**, you will need to open the PDF manually if the `open` command that follows does not work. You might need to go to the directory and double click, or do whatever you usually do when opening files in Windows. Mac and Linux users can use the `open` command as-is.

```
open report.pdf
```

The report should be easier to read than if we observe the data from the journal file. You can send it to your manager and say, "Hey, I generated a report that shows potential problems in our cluster."

I'll let you explore the report yourself. You'll see that it contains all the information we might need in an easy-to-digest format.

The report contains data from a single run of Chaos Toolkit. We're going to see how we can create reports based on multiple experiments later. For now, we have a single-experiment report.

Creating A Multi-Experiment Report

Assuming that we want to see how we can generate reports based on multiple experiments, the first thing we need to do is to run a second experiment. Otherwise, we'd be left with data (journal) from a single experiment.

So, we are going to execute yet another experiment that will generate a second journal file. After that, we'll try to figure out how to create a report based on both journals.

Let's start by taking a quick look at yet another definition.

```
cat chaos/network-delay.yaml
```

The output is as follows.

```
version: 1.0.0
title: What happens if we abort and delay responses
description: If responses are aborted and delayed, the dependant application should \
retry and/or timeout requests
tags:
- k8s
- istio
- http
configuration:
  ingress_host:
      type: env
      key: INGRESS_HOST
steady-state-hypothesis:
  title: The app is healthy
  probes:
  - type: probe
    name: app-responds-to-requests
    tolerance: 200
    provider:
      type: http
      timeout: 15
      verify_tls: false
```

```
      url: http://${ingress_host}?addr=http://go-demo-8
      headers:
        Host: repeater.acme.com
  - type: probe
    tolerance: 200
    ref: app-responds-to-requests
  - type: probe
    tolerance: 200
    ref: app-responds-to-requests
  - type: probe
    tolerance: 200
    ref: app-responds-to-requests
  - type: probe
    tolerance: 200
    ref: app-responds-to-requests
method:
- type: action
  name: abort-failure
  provider:
    type: python
    module: chaosistio.fault.actions
    func: add_abort_fault
    arguments:
      virtual_service_name: go-demo-8
      http_status: 500
      routes:
        - destination:
            host: go-demo-8
            subset: primary
      percentage: 50
      version: networking.istio.io/v1alpha3
      ns: go-demo-8
- type: action
  name: delay
  provider:
    type: python
    module: chaosistio.fault.actions
    func: add_delay_fault
    arguments:
      virtual_service_name: go-demo-8
      fixed_delay: 15s
      routes:
        - destination:
```

```
            host: go-demo-8
            subset: primary
      percentage: 50
      version: networking.istio.io/v1alpha3
      ns: go-demo-8
  pauses:
    after: 1
rollbacks:
- type: action
  name: remove-abort-failure
  provider:
    type: python
    func: remove_abort_fault
    module: chaosistio.fault.actions
    arguments:
      virtual_service_name: go-demo-8
      routes:
        - destination:
            host: go-demo-8
            subset: primary
      version: networking.istio.io/v1alpha3
      ns: go-demo-8
- type: action
  name: remove-delay
  provider:
    type: python
    func: remove_delay_fault
    module: chaosistio.fault.actions
    arguments:
      virtual_service_name: go-demo-8
      routes:
        - destination:
            host: go-demo-8
            subset: primary
      version: networking.istio.io/v1alpha3
      ns: go-demo-8
```

That is the same experiment as one of those we run while we were experimenting with networking. There's nothing new here. We are reusing the experiment we're familiar with, so I'll skip explaining what that does. Instead, we'll jump straight into running it.

```
chaos run chaos/network-delay.yaml \
    --journal-path journal-network-delay.json
```

The output, without timestamps, is as follows.

```
[... INFO] Validating the experiment's syntax
[... INFO] Experiment looks valid
[... INFO] Running experiment: What happens if we abort and delay responses
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Action: abort-failure
[... INFO] Action: delay
[... INFO] Pausing after activity for 1s...
[... INFO] Steady state hypothesis: The app is healthy
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Probe: app-responds-to-requests
[... INFO] Steady state hypothesis is met!
[... INFO] Let's rollback...
[... INFO] Rollback: remove-abort-failure
[... INFO] Action: remove-abort-failure
[... INFO] Rollback: remove-delay
[... INFO] Action: remove-delay
[... INFO] Experiment ended with status: completed
```

The experiment was successful, and you already know what it does. Even if your memory does not serve you well, you should be able to deduce what's it about by reading the definition and comparing it with the outcome of the execution. What matters is that we stored the result in a different journal file.

Let's run another container to generate a report.

```
docker container run \
    --user $(id -u) \
    --volume $PWD:/tmp/result \
    -it \
    chaostoolkit/reporting \
     -- report \
     --export-format=pdf \
    journal-health-http.json \
    journal-network-delay.json \
    report.pdf
```

We created a container with almost the same arguments. The only difference is that we added the path to the second journal file (`journal-network-delay.json`).

We can generate a report based on as many experiments as we want, as long as each produced a different journal file. All we have to do is specify all those journals as arguments. It's up to Chaos Toolkit to assemble the information from all the specified experiments and create a report.

Let's take a look at the report.

```
open report.pdf
```

We can see that this time, the report contains the outcomes from both experiments.

As long as we store results of each experiment in a separate journal file, we can combine them all into one big report and distribute it to whoever is interested.

Destroying What We Created

This was a short and uneventful chapter. I thought you might use a break from destruction and do something other than staring at the terminal all the time. Nevertheless, this chapter should be beneficial since we learned how to generate outputs (reports) that can be distributed throughout the rest of your organization.

Now, just as with any other chapter, we are going to remove everything we created.

```
cd ..

kubectl delete namespace go-demo-8
```

We went one directory back, and we deleted the namespace `go-demo-8`.

You might want to stop Docker demon since we will not need it anymore unless you're using Docker Desktop as your Kubernetes cluster.

Now you need to make a decision. Do you want to go straight into the next chapter? If you do, just go there. On the other hand, you might want to take a break. If that's what you prefer, I suggest you destroy the cluster. You can use the commands from the same Gist you used to create it.

The choice is yours. Destroy the cluster and take a break or move forward to the next section right away.

Running Chaos Experiments Inside A Kubernetes Cluster

We saw that we can run experiments targeting applications running in Kubernetes. We were messing with networking, and we saw how to shut down and manipulate nodes of a cluster. However, we were doing all that by running those experiments locally. Our experiments were targeting a remote Kubernetes cluster.

What we did so far was useful as a way to figure out how chaos experiments work. Now is the time to move the execution of those experiments away from our laptop and into a Kubernetes cluster. We want not only to target a Kubernetes cluster but also to run the experiments inside it and target the components inside the same cluster where chaos experiments are running.

In this chapter, we are going to explore how to transfer the execution of our chaos experiments from our laptops into a Kubernetes cluster.

Gist With The Commands

As you already know, every section that has hands-on exercises is accompanied by a Gist that allows you to copy and paste commands instead of bothering to type. This chapter is not an exception.

All the commands from this chapter are available in the 08-k8s.sh[68] Gist.

Creating A Cluster

We need to have a cluster where we'll run the demo application as well as the experiments. Whether you have it or not depends on whether you destroyed your cluster at the end of the previous chapter. If you didn't destroy it, just skip to the next section right away. If you did destroy the cluster, feel free to use one of the Gists that follow to create a new one. Alternatively, you can roll out your own cluster. It's up to you.

- Docker Desktop with Istio: https://gist.github.com/9a9752cf5355f1b8095bd34565b80aae (docker-istio.sh)

[68] https://gist.github.com/015845b599beca995cdd8e67a7ec99db

- Minikube with Istio: https://gist.github.com/a5870806ae6f21de271bf9214e523b53 (minikube-istio.sh)
- Regional and scalable GKE with Istio: https://gist.github.com/88e810413e2519932b61d81217072daf (gke-istio-full.sh)
- Regional and scalable EKS with Istio: https://gist.github.com/d73fb6f4ff490f7e56963ca543481c09 (eks-istio-full.sh)
- Regional and scalable AKS with Istio: https://gist.github.com/b068c3eadbc4140aed14b49141790940 (aks-istio-full.sh)

Deploying The Application

In the previous chapters, we deployed the demo application, which we targeted in our experiments. This one is not an exception. We are going to deploy the same demo application that we used before. Given that you are already familiar with it, we'll go quickly through the commands without much explanation.

We'll go to the `go-demo-8` directory and pull the latest version of the repository.

```
cd go-demo-8

git pull
```

Next, we'll create a Namespace called `go-demo-8` and add the label so that Istio knows that it should inject proxy sidecars automatically.

```
kubectl create namespace go-demo-8

kubectl label namespace go-demo-8 \
    istio-injection=enabled
```

We're going to apply the definition from the directory `k8s/app-full` and, once all the resources are created, we're going to wait until the Deployment of the API rolls out.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/app-full

kubectl --namespace go-demo-8 \
    rollout status deployment go-demo-8
```

Finally, to be sure that it works fine, we're going to send a request to the newly deployed demo application.

```
curl -H "Host: repeater.acme.com" \
    "http://$INGRESS_HOST?addr=http://go-demo-8"
```

Now we can turn our attention towards setting up the resources we'll need to run experiments inside the Kubernetes cluster.

Setting Up Chaos Toolkit In Kubernetes

Before we start running chaos experiments inside the Kubernetes cluster, we'll need to set up at least two things.

We'll need experiment definitions stored somewhere in the cluster. The most common and the most logical way to define a configuration in Kubernetes is to use ConfigMap. Apart from having experiment definitions readily available, we will also need to create a ServiceAccount that will provide the necessary privileges to processes that will run the experiments.

Let's start with the ConfigMap.

```
cat k8s/chaos/experiments.yaml
```

The output, limited to the first experiment, is as follows.

```
...
apiVersion: v1
kind: ConfigMap
metadata:
  name: chaostoolkit-experiments
data:
  health-http.yaml: |
    version: 1.0.0
    title: What happens if we terminate an instance of the application?
    description: If an instance of the application is terminated, the applications a\
s a whole should still be operational.
    tags:
    - k8s
    - pod
    steady-state-hypothesis:
      title: The app is healthy
      probes:
      - name: app-responds-to-requests
        type: probe
        tolerance: 200
```

```
        provider:
          type: http
          timeout: 3
          verify_tls: false
          url: http://go-demo-8.go-demo-8/demo/person
          headers:
            Host: go-demo-8.acme.com
    method:
    - type: action
      name: terminate-app-pod
      provider:
        type: python
        module: chaosk8s.pod.actions
        func: terminate_pods
        arguments:
          label_selector: app=go-demo-8
          rand: true
          ns: go-demo-8
      pauses:
        after: 2
...
```

We can see that it is a "standard" Kubernetes ConfigMap that contains a few experiments. I did not put all those we explored so far since we won't need them all in this chapter. Nevertheless, you should be able to add additional experiments, as long as you don't reach the limit of 1MB. That's the limitation of etcd, which is the registry where Kubernetes stores its objects. If you do need more than that, you can use multiple ConfigMaps.

The vital part of that definition is the `data` section.

We can see that there is the `health-http.yaml` key, which, as you hopefully know, will be transformed into a file with the same name when we mount that ConfigMap. The value is a definition of an experiment. The steady `steady-state-hypothesis` is checking whether the application is healthy. It does that through the probe that sends requests to the app. The `action` of the `method` is terminating one of the Pods of the `go-demo-8` application.

Further down, we can see that there are a few other experiments. We won't go through them because all those defined in that ConfigMap are the same as the ones we used before. In this chapter, we are not trying to figure out new ways to create chaos, but rather how to run experiments inside a Kubernetes cluster.

All in all, our experiments will be defined as ConfigMaps, and they will be available to the processes inside the cluster.

Let's apply that definition and move forward.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/chaos/experiments.yaml
```

Next, to be on the safe side, we are going to describe the ConfigMap and confirm that it is indeed created and available inside our cluster.

```
kubectl --namespace go-demo-8 \
    describe configmap chaostoolkit-experiments
```

We can see that the output is, more or less, the same as the content that we already saw in the YAML file that we used to create that ConfigMap.

The next one in line is the ServiceAccount. We need it to provide sufficient permissions to `chaos` processes that we'll run in the `go-demo-8` Namespace.

Let's take a look at the `sa.yaml` file.

```
cat k8s/chaos/sa.yaml
```

The output is as follows.

```
---

apiVersion: v1
kind: ServiceAccount
metadata:
  name: chaostoolkit

---

apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: chaostoolkit
rules:
- apiGroups:
  - ""
  - "extensions"
  - "apps"
  resources:
  - pods
  - deployments
  - jobs
```

```
23    verbs:
24    - list
25    - get
26    - watch
27    - delete
28  - apiGroups:
29    - ""
30    resources:
31    - "persistentvolumeclaims"
32    verbs:
33    - list
34    - get
35    - create
36    - delete
37    - update
38    - patch
39    - watch
40
41  ---
42
43  apiVersion: rbac.authorization.k8s.io/v1beta1
44  kind: RoleBinding
45  metadata:
46    name: chaostoolkit
47  roleRef:
48    apiGroup: rbac.authorization.k8s.io
49    kind: Role
50    name: chaostoolkit
51  subjects:
52    - kind: ServiceAccount
53      name: chaostoolkit
```

We can see that there are a few things defined there. To begin with, we have the ServiceAccount called `chaostoolkit`. As you hopefully already know, ServiceAccounts themselves are just references to something, and they don't serve much of a purpose without binding them to roles which, in turn, define permissions. So, that YAML definition contains a `Role` with the `rules` that determine which actions (`verbs`) can be executed on specific types of `apiGroups` and `resources`. Further on, we have a `RoleBinding` that binds the Role to the `ServiceAccount`.

The short explanation of that definition is that it will allow us to do almost anything we might ever need to do, but limited to a specific Namespace.

In a real-world situation, you might want to be more restrictive than that. Those permissions will effectively allow processes in that Namespace to do anything inside it. On the other hand, it is tough

to be restrictive with permissions we need to give to our chaos experiments. Theoretically, we might want to affect anything inside a Namespace or even the whole cluster through experiments. So, no matter how strong our desire is to be restrictive with the permissions in general, we might need to be generous to chaos experiments. For them to work correctly, we most likely need to allow a wide range of permissions. As a minimum, we have to permit them to perform the actions we decided we'll run.

From permissions point of view, the only real restriction that we're setting up is that we are creating the `RoleBinding` and not a `ClusterRoleBinding`. Those permissions will be assigned to the `ServiceAccount` inside that Namespace. As a result, we'll limit the capability of Chaos Toolkit to that Namespace, and it will not be able to affect the whole cluster.

Later on, we might dive into an even wider range of permissions. We might choose to create a subset of experiments that are operating on the level of the whole cluster. But, for now, limiting permissions to a specific Namespace should be enough. Those permissions, as I already mentioned, will allow Chaos Toolkit to do almost anything inside a particular Namespace.

All that is left, for now, is to apply that definition and create the `ServiceAccount`, the `Role`, and the `RoleBinding`.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/chaos/sa.yaml
```

Now we should be able to run our experiments inside the Kubernetes cluster. We have the ConfigMap that contains the definitions of the experiments, and we have the ServiceAccount bound to the Role with sufficient permissions.

Types Of Experiment Executions

Generally speaking, there are two ways we can run experiments. Or, to be more concrete, we'll explore two even though are certainly other methods.

We can run one-shot experiments by executing experiments on demand. We can say, "let's run that experiment right now." I'm calling them "one-shot" because such experiments would not be recurring. We'll be able to choose when to execute them, and they will exist only while they're running. We'll be able to monitor the progress and observe what's going on. We could run those experiments manually, or we could hook them into our continuous delivery. In the latter case, they would be part of our pipelines and, for example, executed whenever we deploy a new release.

Later on, we're going to explore how to create scheduled experiments that will run periodically. We might choose to do chaos every few minutes, every few hours, once a day, or whatever the schedule we'll define is.

Running One-Shot Experiments

For now, we're going to focus on one-shot experiments that we'll run manually, or through pipelines, or through any other means that we might see fit.

Let's take a look at yet another Kubernetes YAML file.

```
cat k8s/chaos/once.yaml
```

The output is as follows.

```
---

apiVersion: batch/v1
kind: Job
metadata:
  name: go-demo-8-chaos
spec:
  activeDeadlineSeconds: 600
  backoffLimit: 0
  template:
    metadata:
      labels:
        app: go-demo-8-chaos
      annotations:
        sidecar.istio.io/inject: "false"
    spec:
      serviceAccountName: chaostoolkit
      restartPolicy: Never
      containers:
      - name: chaostoolkit
        image: vfarcic/chaostoolkit:1.4.1-2
        args:
        - --verbose
        - run
        - /experiment/health-http.yaml
        env:
        - name: CHAOSTOOLKIT_IN_POD
          value: "true"
        volumeMounts:
        - name: config
          mountPath: /experiment
```

```
32            readOnly: true
33          resources:
34            limits:
35              cpu: 20m
36              memory: 64Mi
37            requests:
38              cpu: 20m
39              memory: 64Mi
40        volumes:
41        - name: config
42          configMap:
43            name: chaostoolkit-experiments
```

What matters is that we are defining a Kubernetes `Job`.

"A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As Pods successfully complete, the Job tracks the successful completions. When a specified number of successful completions is reached, the task (Job) is complete. Deleting a Job will clean up the Pods it created."

We can see from that description (taken from Kubernetes docs) that a Job is an excellent candidate to run something once while being able to track the status and, especially, exit codes of the processes inside it. If we limit ourselves to core Kubernetes components, Jobs are probably the most appropriate type of Kubernetes resources we can use when we want to run something like an experiment. The processes inside containers in a Job will start, run, and finish. Unlike most other Kubernetes resources, Jobs are not restarted or recreated when they are successful or when Pods are deleted. That's one of the main differences when compared to, let's say, Deployments and StatefulSets.

On top of that, we can configure Jobs not to restart on failure. That's what we're doing in that definition by setting `spec.template.spec.restartPolicy` to `Never`. Experiments can be successful or failed, and no matter the outcome of an experiment, the Pod created by that Job will run only once.

Most of that specification is not very exciting. What matters is that we set the `serviceAccountName` to `chaostoolkit`. That should give the Job sufficient permissions to do whatever we need it to do.

We defined only one container. We could have more if we'd like to run multiple experiments. But, for our purposes, one should be more than enough to demonstrate how experiments defined as Jobs work.

I encourage you to create your own container image that will contain Chaos Toolkit, the required plugins, and anything else that you might need. But, to simplify things, we'll use the one I created (`vfarcic/chaostoolkit`). We'll discuss that image soon.

We can see from the `args` that we want the output to be verbose, and that it should `run` the experiment defined in `/experiment/health-http.yaml`. If you're wondering where does that file come from, the short answer is "from the ConfigMap". We'll discuss it in a moment.

Then we have the `env` variable `CHAOSTOOLKIT_IN_POD` set to `true`. Chaos Toolkit might need to behave slightly differently when running inside a Pod, so we're using that variable to ensure it knows where it is.

Further on, we have `volumeMounts`. We are mounting something called `config` as the directory `/experiment`. That "something" is defined in the `volumes` section, and it is referencing the ConfigMap `chaostoolkit-experiments` we created earlier. That way, all the entries from that ConfigMap will be available as files inside the `/experiment` directory in the container.

Finally, we also defined the `resources` so that Kubernetes knows how much memory and CPU we're requesting, and what should be the `limits`.

Before we move on, let's take a quick look at the definition of the image `vfarcic/chaostoolkit`.

If you are a **Windows user**, the `open` command might not work. If that's the case, please copy the address from the command that follows, and paste it into your favorite browser.

```
open "https://github.com/vfarcic/chaostoolkit-container-image"
```

That repository has only `Dockerfile`. Open it, and you'll see the definition of the container image. It is as follows.

```
FROM python:3.8.1-alpine

LABEL maintainer="Viktor Farcic <viktor@farcic.com>"

RUN apk add --no-cache --virtual build-deps gcc g++ git libffi-dev linux-headers pyt\
hon3-dev musl-dev && \
    pip install --no-cache-dir  -q -U pip && \
    pip install --no-cache-dir chaostoolkit && \
    pip install --no-cache-dir chaostoolkit-kubernetes && \
    pip install --no-cache-dir chaostoolkit-istio && \
    pip install --no-cache-dir chaostoolkit-slack && \
    pip install --no-cache-dir slackclient==1.3.2 && \
    apk del build-deps

ENTRYPOINT ["/usr/local/bin/chaos"]
CMD ["--help"]
```

I will assume that you are already familiar with Dockerfile format and that you know that it is used to define instructions that tools can use to build container images. There are quite a few builders that use such definitions, Docker and Kaniko being two among many.

As you can see, the definition of the image (`Dockerfile`) is relatively simple and straightforward. It is based on `python` since that's a requirement for Chaos Toolkit, and it installs the CLI and the plugins we'll need. The entry point is the path to the `chaos` binary. By default, it'll output help if we do not overwrite the command (`CMD`).

Now that we explored the Job that will run the experiment, we are ready to apply it and see what happens.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/chaos/once.yaml
```

We can see that the Job `go-demo-8-chaos` was `created`.

Next, we'll take a look at the Pods in that Namespace. To be more specific, we're going to retrieve all the Pods and filter them by using selector `app=go-demo-8-chaos` since that's the name of the label that we put to the template of the Pod that will be created by the Job.

```
kubectl --namespace go-demo-8 \
    get pods \
    --selector app=go-demo-8-chaos
```

The output is as follows.

```
NAME                READY STATUS  RESTARTS AGE
go-demo-8-chaos-... 1/1   Running 0        14s
```

We can see that, in my case, the Pod created by the Job is `Running`. If that's what you see on your screen, please repeat that command until the `STATUS` is `Completed`. That should take around a minute. When it's finished, the output should be similar to the one that follows.

```
NAME                READY STATUS    RESTARTS AGE
go-demo-8-chaos-... 0/1   Completed 0        75s
```

The execution of the Pod created through the Job finished. It is `Completed`, meaning that the experiment was successful.

Let's output the logs and confirm that the experiment was indeed successful.

```
kubectl --namespace go-demo-8 \
    logs --selector app=go-demo-8-chaos \
    --tail -1
```

Since we wanted to output the logs only of the Pod that executed the experiment, we used the `selector` as a filter.

I won't present the output in here since it is too big to fit into the book. You should see on your screen the events that we already explored countless times before. The only substantial difference between now and then is that we run the experiment from a container, instead of a laptop. As a result, Kubernetes added timestamps and log levels to each output entry.

We managed to run an experiment from a Kubernetes cluster in a very similar way as if we'd run it from a laptop.

Before we proceed, we'll delete the Job. We won't need it anymore since we are about to switch to the scheduled execution of experiments.

```
kubectl --namespace go-demo-8 \
    delete --filename k8s/chaos/once.yaml
```

That's it. The Job is no more.

As you saw, running one-shot experiments inside a Kubernetes cluster is straightforward. I encourage you to hook them into your continuous delivery pipelines. All you have to do is really create a Job.

In the next section, we're going to take a look at how we can schedule our experiments to be executed periodically instead of running one-shot executions.

Running Scheduled Experiments

In some cases, one-shot experiments are useful. You might want to trigger an experiment based on specific events (e.g., deployment of a new release), or as a scheduled exercise with "all hands on deck". However, there are situations when you might want to run chaos experiments periodically. You might, for example, decide to execute them once a day at a specific hour. Or you might even choose to randomize that and run them periodically at a random time of a day.

We want to test the system and be objective. This might sound strange, but being objective with chaos engineering often means being, more or less, random. If we know when something potentially disrupting might happen, we might react differently than when in unexpected situations. We might be biased and schedule the execution of experiments at the time when we know that there will be no adverse effect on the system. Instead, it might be a good idea to run experiments at some random intervals during the day or a week so that we cannot easily predict what will happen. We often don't control when will "bad" things happen in production. Most of the time, we don't know when a node will die, and we often cannot guess when a network will stop being responsive.

Similarly, we should try to include some additional level of randomness to the experiments. If we run them only when we deploy a new release, we might not discover the adverse effects that might be produced hours later. We can partly mitigate that by running experiments periodically.

Let's see how we can create periodic execution of chaos experiments. To do that, we are going to take a look at yet another YAML file.

```
cat k8s/chaos/periodic.yaml
```

The output is as follows.

```
---

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: go-demo-8-chaos
spec:
  concurrencyPolicy: Forbid
  schedule: "*/5 * * * *"
  jobTemplate:
    metadata:
      labels:
        app: go-demo-8-chaos
    spec:
      activeDeadlineSeconds: 600
      backoffLimit: 0
      template:
        metadata:
          labels:
            app: go-demo-8-chaos
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          serviceAccountName: chaostoolkit
          restartPolicy: Never
          containers:
          - name: chaostoolkit
            image: vfarcic/chaostoolkit:1.4.1-2
            args:
            - --verbose
            - run
            - --journal-path
            - /results/journal-health-http.json
            - /experiment/health-http.yaml
            env:
            - name: CHAOSTOOLKIT_IN_POD
```

```
              value: "true"
            volumeMounts:
            - name: experiments
              mountPath: /experiment
              readOnly: true
            - name: results
              mountPath: /results
              readOnly: false
            resources:
              limits:
                cpu: 20m
                memory: 64Mi
              requests:
                cpu: 20m
                memory: 64Mi
          volumes:
          - name: experiments
            configMap:
              name: chaostoolkit-experiments
          - name: results
            persistentVolumeClaim:
              claimName: go-demo-8-chaos

---

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: go-demo-8-chaos
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

That YAML is slightly bigger than the previous one. The major difference is that this time we are not defining a Job. Instead, we have a `CronJob`, which will create the Jobs in scheduled intervals.

If you take a closer look, you'll notice that the CronJob is almost the same as the Job we used before. There are a few significant differences, though.

First of all, we probably don't want to run the same Jobs concurrently. Running one copy of an experiment at a time should be enough. So, we set `concurrencyPolicy` to `Forbid`.

The `schedule`, in this case, is set to `*/5 * * * *`. That means that the Job will run every five minutes unless the previous one did not yet finish since that would contradict the `concurrencyPolicy`.

If you're not familiar with the syntax like `*/5 * * * *`, it is the standard syntax from `crontab` available in (almost) every Linux distribution. You can find more info in the CRON expression[69] section of the Cron entry in Wikipedia.

In "real world" situations, running an experiment every five minutes might be too frequent. Something like once an hour, once a day, or even once a week is more appropriate. But, for the purpose of this demonstration, I want to make sure that you're not waiting for too long to see the results of an experiment. So, we will run our experiments every five minutes.

The `jobTemplate`, as the name suggests, defines the template that will be used to create Jobs. Whatever is inside, it is almost the same as what we had in the Job we created earlier.

The significant difference between now and then is that this time we're using CronJob to create Jobs at scheduled intervals. The rest is, more or less, the same. The one difference is that when we run one-shot experiments, we're in control. We can tail the logs because we are observing what's happening, or we're letting pipelines decide what to do next. However, the in case of scheduled periodic experiments, we probably want to store the results somewhere. We most likely want to write journal files that can be converted into reports later. For that, we are mounting an additional volume besides the ConfigMap. It is called `results`, and it references `PersistentVolumeClaim` called `go-demo-8-chaos`.

If we go back to the `args`, we can see that we are setting `--journal-path` argument to the value with the path to `/results/journal-health-http.json`. Since `/results` is the directory that will be mounted to an external drive, our journals will be persisted. Or, to be more precise, the last journal will be stored there. Given that the name of the journal is always the same, new ones will always overwrite the older one. We could randomize that, but, for our purposes, the latest journal should be enough. Remember, we'll run the experiments every five minutes, but, in the real-world situation, the frequency would be much lower, and you should have enough time to explore the journal if an experiment fails before the execution of a new one starts. Anyways, what matters is that we will not only have a CronJob that creates Jobs periodically but that we will be storing journal files of our experiments in an external drive. From there, you should be able to fetch those journals and convert them into PDFs, just as you did before.

Let's apply that definition and see what we'll get.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/chaos/periodic.yaml
```

Next, we'll retrieve the CronJobs and wait until the one we just created runs a Job.

[69] https://en.wikipedia.org/wiki/Cron#CRON_expression

```
kubectl --namespace go-demo-8 \
    get cronjobs
```

The output, in my case, is as follows.

```
NAME            SCHEDULE    SUSPEND ACTIVE LAST SCHEDULE AGE
go-demo-8-chaos */5 * * * * False   0      <none>        15s
```

If, in your case, the `LAST SCHEDULE` is also set to `<none>`, you'll need a bit of patience. Jobs are created every five minutes. Keep re-running the previous command until the status of the `LAST SCHEDULE` column changes from `<none>` to something else. The output should be similar to the one that follows.

```
NAME            SCHEDULE    SUSPEND ACTIVE LAST SCHEDULE AGE
go-demo-8-chaos */5 * * * * False   1      6s            84s
```

In my case, the CronJob created a Job six seconds ago.

Let's retrieve the Jobs and see what we'll get.

```
kubectl --namespace go-demo-8 \
    get jobs
```

The output, in my case, is as follows.

```
NAME                COMPLETIONS DURATION AGE
go-demo-8-chaos-... 0/1         19s      19s
```

We can see that, in my case, there is only one Job because there was not sufficient time for the second to run.

What matters is that this Job was created by the CronJob, and that it did not finish executing. It is still running. If that's the situation on your screen as well, you'll need to repeat the previous command until the `COMPLETIONS` column is set to `1/1`. Remember, it takes around a minute or two for the experiment to execute. After a while, the output of `get jobs` should be similar to the one that follows.

```
NAME                COMPLETIONS DURATION AGE
go-demo-8-chaos-... 1/1         59s      72s
```

The `COMPLETIONS` column now says `1/1`. In my case, it took around a minute to execute it.

Given that Jobs create Pods, we'll retrieve all those in the `go-demo-8` Namespace and see what we'll get.

```
kubectl --namespace go-demo-8 \
    get pods
```

```
NAME                READY STATUS    RESTARTS AGE
go-demo-8-...       2/2   Running   2        14m
go-demo-8-...       2/2   Running   0        30s
go-demo-8-chaos-... 0/1   Completed 0        82s
go-demo-8-db-...    2/2   Running   0        14m
repeater-...        2/2   Running   0        14m
repeater-...        2/2   Running   0        14m
```

We can see that we have the Pods of our application, which are all `Running`. Alongside them, there is the `go-demo-8-chaos` Pod with the `STATUS` set to `Completed`. Since Jobs start processes in containers one by one and do not restart them when finished, the `READY` column is set to `0/1`. The processes in that Pod finished, and now none of the containers is running.

Let's summarize.

We created a scheduled CronJob. Every five minutes, a new Job is created, which, in turn, creates a Pod with a single container that runs our experiments. For now, there is only one container, but we could just as well add more.

Next, we'll explore what happens if an experiment fails.

Running Failed Scheduled Experiments

Let's see what happens when an experiment fails. You probably already know the outcome, or, at least, you should be able to guess. Nevertheless, we'll simulate a failure of the experiment by deleting the Deployment `go-demo-8`. As a result, the Pods of the application that is used as the target of the experiment will be terminated, and the experiment will inevitably fail.

```
kubectl --namespace go-demo-8 \
    delete deployment go-demo-8
```

The target of the experiment (the application) is gone. We're going to retrieve the Pods of the experiment created by the CronJob.

```
kubectl --namespace go-demo-8 \
    get pods \
    --selector app=go-demo-8-chaos
```

Keep repeating that command until the output is similar to the one that follows. Remember, new experiments are being created every five minutes. On top of that, we need to wait for a minute or two until the `chaos` process finishes running. The end result should be the creation of a new Pod, which, when the experiment inside it is finished, should have the `STATUS` set to `error`.

```
NAME                READY STATUS    RESTARTS AGE
go-demo-8-chaos-... 0/1   Completed 0        6m8s
go-demo-8-chaos-... 0/1   Error     0        67s
```

The experiment failed because we deleted the application it was targeting. The process running inside the container failed and returned an exit code other than zero. That was the indication to Kubernetes to treat it as an error. The process inside the container running inside that Pod did not terminate successfully; hence the experiment failed.

There's probably no need to go deeper. I just wanted to show you how both successful and unsuccessful experiments look like when scheduled periodically through a CronJob.

If we'd like to generate reports, we have the journal file with the information from the last execution. It is stored in the PersistentVolume. All we'd have to do is retrieve it from that volume. From there on, we should be able to generate the report just as we did in the past.

I will not show you how to retrieve something from persistent volumes. Instead, I will assume that you already know how to do that. If you don't, you are likely new to Kubernetes. If that's the case, I'm surprised that you got this far into the book. Nevertheless, if you are indeed new to Kubernetes and you're not sure how to handle persistent volumes, I recommend you read The DevOps 2.3 Toolkit: Kubernetes[70]. Read that book or the official documentation to get more familiar with Kubernetes.

On the other hand, if you already have experience with Kubernetes, retrieving data from a volume should not be a problem. In either case, I will not explain that here. Kubernetes itself is not the subject of this book. Instead, I'll just give you a tip by saying that I'd create a report by creating a new Pod that has the same volume attached, and I'd run the process of creating the report from there. After that, I'd push the report to some external storage that is easy to access like, for example, a Git repository, an S3 bucket, Artifactory, or whatever else I might have at my disposal.

In any case, let's confirm that the PersistentVolume is indeed there.

```
kubectl get pv
```

The output is as follows.

```
NAME    CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM                     STORAG\
ECLASS REASON AGE
pvc-... 1Gi      RWO          Delete         Bound  go-demo-8/go-demo-8-chaos standa\
rd            7m44s
pvc-... 8Gi      RWO          Delete         Bound  go-demo-8/go-demo-8-db    standa\
rd            19m
```

As we can see, one of the two volumes is claimed by `go-demo-8-chaos`. That's where the journals are being stored.

[70] https://www.devopstoolkitseries.com/posts/devops-23/

There are still a couple of things we need to figure out when running chaos experiments inside a Kubernetes cluster. But, before we move on, we're going to re-apply the definition of the demo application. We destroyed the app so that we ensure that the experiment will fail. Now we're going to recreate it.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/app-full
```

To be on the safe side, we'll wait until it rolls out.

```
kubectl --namespace go-demo-8 \
    rollout status deployment go-demo-8
```

There is at least one more important thing that we should explore. In the next section, we'll try to figure out how to send notifications from the experiments.

Sending Experiment Notifications

At the time of this writing (March 2020), Chaos Toolkit allows us to send notifications to at least two different targets (there may be others by the time you're reading this). We can send notifications to Slack, or to any HTTP endpoint.

We'll focus on Slack. If that's not your favorite communication tool, you should be able to change the configuration to use anything else, as long as that destination can receive HTTP requests. Since almost every tool designed in this century has an API exposed through HTTP, we should be able to send notifications to any destination. The question is whether we do that through a plugin, just like the one we're about to use, or through the generic HTTP notification mechanism available in Chaos Toolkit.

We're going to explore how to send notifications to Slack. Later on, if you decide that Slack is not something you want to use, you should be able to adapt the examples to HTTP notifications.

For you to be able to see the notifications we are about to send, you'll have to do one of two things. You can join DevOps20 Slack workspace[71]. If you don't want to do that and, instead, you prefer to use your own Slack workspace, you will need to change a few commands. Specifically, you'll have to change the Slack API token in some of the YAML definitions we're about to explore.

So, either use my Slack workspace, and for that, you'll have to join DevOps20 Slack workspace[72] or modify some of the YAML definitions that I prepared.

Let's take a look at `settings.yaml` that contains a few things that we will need.

[71] http://slack.devops20toolkit.com/
[72] http://slack.devops20toolkit.com/

```
cat k8s/chaos/settings.yaml
```

The output is as follows.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: chaostoolkit-settings
data:
  settings.yaml: |
    notifications:
    - type: plugin
      module: chaosslack.notification
      token: xoxb-@102298415591@-@902246554644@-@xa4zDR3lcyvleRymXAFSifLD@
      channel: tests
```

That's a simple ConfigMap with a single key `settings.yaml`. Normally, we use ConfigMaps in Kubernetes for all sorts of configurations. The exceptions would be confidential information that can be stored as Kubernetes Secrets, in HashiCorp Vault, or in any other secrets management solution. But, we won't dive into secrets since that would result in a big topic by itself and would not be directly related to chaos experiments. We'd need not only to create secrets (that part is easy) but also to figure out how to inject them into that ConfigMap. We'd probably end up with temporary settings (e.g.`settings.yaml.tmp` and a secret, and we'd combine them into the final `settings.yaml` file during boot of the container with Chaos Toolkit. Instead, we're taking a shortcut by adding the `token` inside the ConfigMap.

Let's get back to the task at hand.

Chaos Toolkit expects `.chaostoolkit/settings.yaml` file inside the home directory of the current user. We'll deal with that later. For now, what matters is that the ConfigMap defines the `settings.yaml` file with everything we'll need to send notifications to Slack.

The `settings.yaml` file contains a single entry in the `notification` section that specifies that we want to use the `chaosslack.notification` module, which is available through the Slack plugin that is already available in the container image we explored earlier. Additionally, we have the `token` and the `channel`. The former is, as you can guess, used to authenticate to Slack, while the latter specifies which channel within the workspace should receive notifications.

If you already used Slack tokens, you might have noticed that the on in that YAML is a bit strange. I could not put the "real" token because GitHub would not allow me to push confidential information. Instead, I "tricked" the system by adding the `@` character in a few places inside the token. That way, GitHub does not recognize it as a Slack token, while it's still relatively easy to transform it into the "real" one. All we have to do is remove `@` characters. We'll use `sed` for that.

```
cat k8s/chaos/settings.yaml \
    | sed -e "s|@||g" \
    | kubectl --namespace go-demo-8 \
    apply --filename -
```

We output the content of the `settings.yaml` file and used it as the input to the `sed` command. It, in turn, replaced all occurrences of `@` with an empty string. The transformed definition was sent to the `kubectl apply` command, which created the modified ConfigMap inside the `go-demo-8` Namespace.

Now that we have the ConfigMap with `settings.yaml`, which defines the target (Slack) that should receive the notifications, we can include it into our CronJob.

```
cat k8s/chaos/periodic-slack.yaml
```

The output, limited to the relevant parts, is as follows.

```
---

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: go-demo-8-chaos
spec:
  ...
  jobTemplate:
    ...
    spec:
      ...
      template:
        ...
        spec:
          ...
          containers:
          - name: chaostoolkit
            ...
            volumeMounts:
            ...
            - name: settings
              mountPath: /root/.chaostoolkit
              readOnly: true
            ...
          volumes:
          - name: experiments
```

```
28             configMap:
29               name: chaostoolkit-experiments
30           ...
```

The new definition is almost the same as the CronJob currently running in our cluster. The only novelty is the addition of the ConfigMap. The CronJob has a new entry in the `volumeMounts` section referencing the volume `settings`. It mounts it as the `/root/.chaostoolkit` directory, which is the location Chaos Toolkit expects to find `settings.yaml`. Further down, we can see those same `settings` defined as the `volume` that is the `configMap` with the name `chaostoolkit-experiments`.

To be sure that we did not miss any other change when compared with the previous definition, we're going to output the differences.

```
diff k8s/chaos/periodic.yaml \
    k8s/chaos/periodic-slack.yaml
```

The output is as follows.

```
41a42,44
>   - name: settings
>     mountPath: /root/.chaostoolkit
>     readOnly: true
55a59,61
>   - name: settings
>     configMap:
>       name: chaostoolkit-settings
```

As you can see, the entries to mount the ConfigMap are the only changes between now and then.

Let's `apply` the new definition.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/chaos/periodic-slack.yaml
```

Now that the CronJob was reconfigured, we should start seeing notifications in the DevOps20 Slack workspace, unless you chose to use your own token.

Go to the DevOps Slack workspace and choose the channel tests. You will see the output of your chaos experiment soon.

Don't get confused if you don't see only the results of your experiments. You might notice notifications from experiments run by other people that might be going through the same exercises as you. It all depends on how many people are exploring notifications at the same time. What matters, though, is that we are going to see our notifications popping up every five minutes and that they might be mixed with notifications coming from others.

You'll see a notification when the next experiment starts (`run-started`), as well as when it ends (`run-completed`). You should be able to observe something similar to *figure 8-1*.

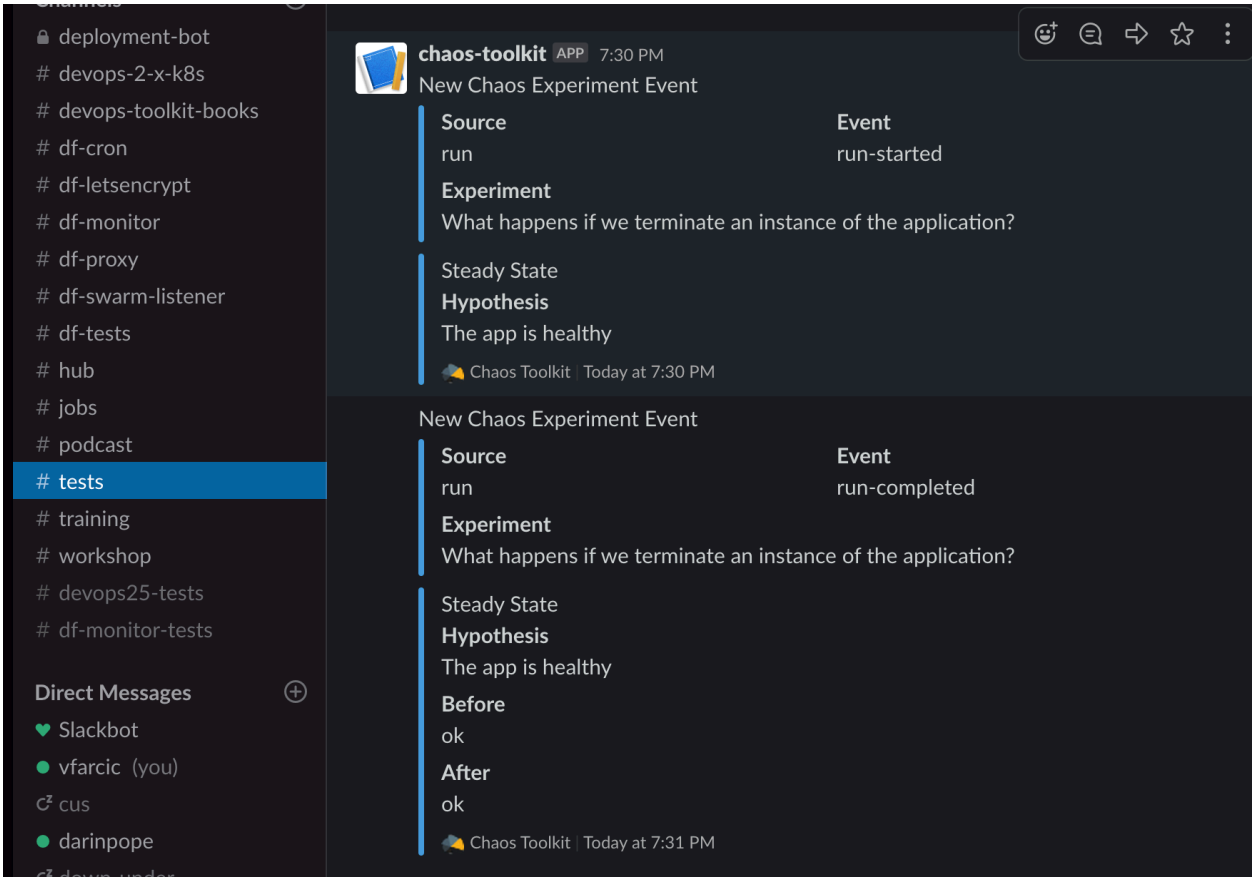

Figure 8-1: Chaos Toolkit Slack notifications

I won't insult your intelligence by going through the information we see in those notifications. They are the same pieces of data we've seen countless times before. The only difference is that they're now in Slack.

If you keep the CronJob running, the same notification will pop up every five minutes. Just don't get confused if you see an occasional unexpected notification. Others might be ahead of you.

What is interesting about the notifications is that they are not specific to a failure or a success. We are, right now, receiving notifications, no matter the outcome of the experiments. We'll change that soon. We'll try to figure out how to filter the notifications so that, for example, we receive only unsuccessful experiments. But, for now, what we are having in front of us are the results of the experiments, no matter whether they're successful or failed.

We can confirm the same thing if we output the Pods.

```
kubectl --namespace go-demo-8 \
    get pods \
    --selector app=go-demo-8-chaos
```

The output, in my case, is as follows.

```
NAME                  READY   STATUS      RESTARTS   AGE
go-demo-8-chaos-...   0/1     Error       0          16m
go-demo-8-chaos-...   0/1     Completed   0          11m
go-demo-8-chaos-...   0/1     Completed   0          6m14s
go-demo-8-chaos-...   0/1     Completed   0          83s
```

Just as before, we can see that the new Pod was `completed`.

If you are curious to see more information about how to set up notifications, please go to the Get Notifications From The Chaos Toolkit Flow[73] page in the documentation. In the next section, we'll go slightly deeper into notifications and try to filter them a bit more.

Sending Selective Notifications

We are now receiving notifications whenever an experiment starts and whenever it ends, no matter whether it is successful or failed. That, however, might be too much. We can easily be overwhelmed with too many notifications. As a result, we are likely going to start ignoring the notifications and might easily miss those that do matter. Instead, I prefer to receive notifications only when something terrible happens. I prefer that I receive notifications from the system only when they are critical. That, more often than not, means that we might want to receive only notifications that should result in some actions. What's the point of knowing than an experiment was successful just as many others were successful before. No news can be interpreted as good news. Instead, I prefer to receive notifications that indicate that there is an action that I should perform. A failed experiment should be a clear indication that the system does not behave as it should, and that we should do our best to improve it so that the same experiment is successful the next time it runs.

We're going to modify the `notifications` section in Chaos Toolkit settings so that we are notified only when experiments fail. That way, we'll be ignorant when they're successful, and we'll be ready to act when they fail.

Let's take a look at yet another YAML.

```
cat k8s/chaos/settings-failure.yaml
```

The output is as follows.

[73] https://docs.chaostoolkit.org/reference/usage/notification/

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: chaostoolkit-settings
data:
  settings.yaml: |
    notifications:
    - type: plugin
      module: chaosslack.notification
      token: xoxb-@102298415591@-@902246554644@-@xa4zDR3lcyvleRymXAFSifLD@
      channel: tests
      events:
      - discover-failed
      - run-failed
      - validate-failed
```

That `ConfigMap` is very similar to the one we used before. The only difference is in the additional section `events`. Over there, we're specifying that we want to send notifications only if certain events happen. In our case, those are all related to failures. We'll be sending notifications if there is an issue detected during discovery (`discover-failed`), during running (`run-failed`), or during validation (`validate-failed`) phases. Those three cover all failure scenarios in Chaos Toolkit, so we'll be notified every time something goes wrong.

If you're curious about which other events you can use, please go to the Chaos Toolkit Flow Events[74] section in the documentation.

To be on the safe side, we'll confirm that those are indeed the only changes to the ConfigMap definition by outputting the `diff` with the previous definition.

```
diff k8s/chaos/settings.yaml \
    k8s/chaos/settings-failure.yaml
```

The output is as follows.

```
11a12,15
>       events:
>       - discover-failed
>       - run-failed
>       - validate-failed
```

As you can see, the `events` section is indeed the only addition to the ConfigMap.

Let's modify the `token` by removing the `@` characters and applying the result with `kubectl`, just as we did before.

[74] https://docs.chaostoolkit.org/reference/usage/notification/#chaos-toolkit-flow-events

```
cat k8s/chaos/settings-failure.yaml \
    | sed -e "s|@||g" \
    | kubectl --namespace go-demo-8 \
    apply --filename -
```

Next, we'll start fetching the Pods and waiting until the CronJob spins up a new one.

```
kubectl --namespace go-demo-8 \
    get pods \
    --selector app=go-demo-8-chaos
```

The output, in my case, is as follows.

```
NAME                READY STATUS    RESTARTS AGE
go-demo-8-chaos-... 0/1   Error     0        26m
go-demo-8-chaos-... 0/1   Completed 0        11m
go-demo-8-chaos-... 0/1   Completed 0        6m21s
go-demo-8-chaos-... 0/1   Completed 0        81s
```

We can see from my output that the last job was created eighty-one seconds ago. So, I'll have to wait approximately three and a half minutes until the next Job is created, and a minute or two until it finishes executing. In your case, the remaining time will be different depending on when the last job was created.

Keep repeating the `kubectl get` command until a new Pod (created by a new Job) starts running, and it completes the execution of the experiment.

Please go back to Slack once the new Pod is created and with the `STATUS` set to `Completed`.

If you ignore potential notifications coming from other readers, you'll see that there's nothing new. No notifications are coming from you. The process didn't create a new notification because the experiment for successful, and we configured Chaos Toolkit to send notification only if an experiment fails. From now on, there will be no notifications coming from you. Or, to be more precise, until we do something that will make one of the experiments fail. Just remember what I said before. Other people might be running experiments while following this book, so you might see notifications from others. What matters is that your notification is not there. The experiment was successful.

Let's change the situation. Let's confirm not only that we don't receive notifications when experiments are successful, but also that we do get new messages in Slack when an experiment fails. We're going to do that in the same way as we did before. We're going to simulate failure by removing the Deployment of the application. That will inevitably result in a failed experiment since it will try to confirm that the app that does not exist is healthy.

```
kubectl --namespace go-demo-8 \
    delete deployment go-demo-8
```

We removed the `go-demo-8` Deployment, and that, as you already know, will terminate the ReplicaSets, which, in turn, will eliminate the Pods. As a result, the experiment should fail to confirm that the application is healthy.

Now comes the waiting time. We need to be patient until the next experiment is executed. Please go back to Slack and wait for a while.

After a while, you should see a new notification similar to the one in *figure 8-2*.

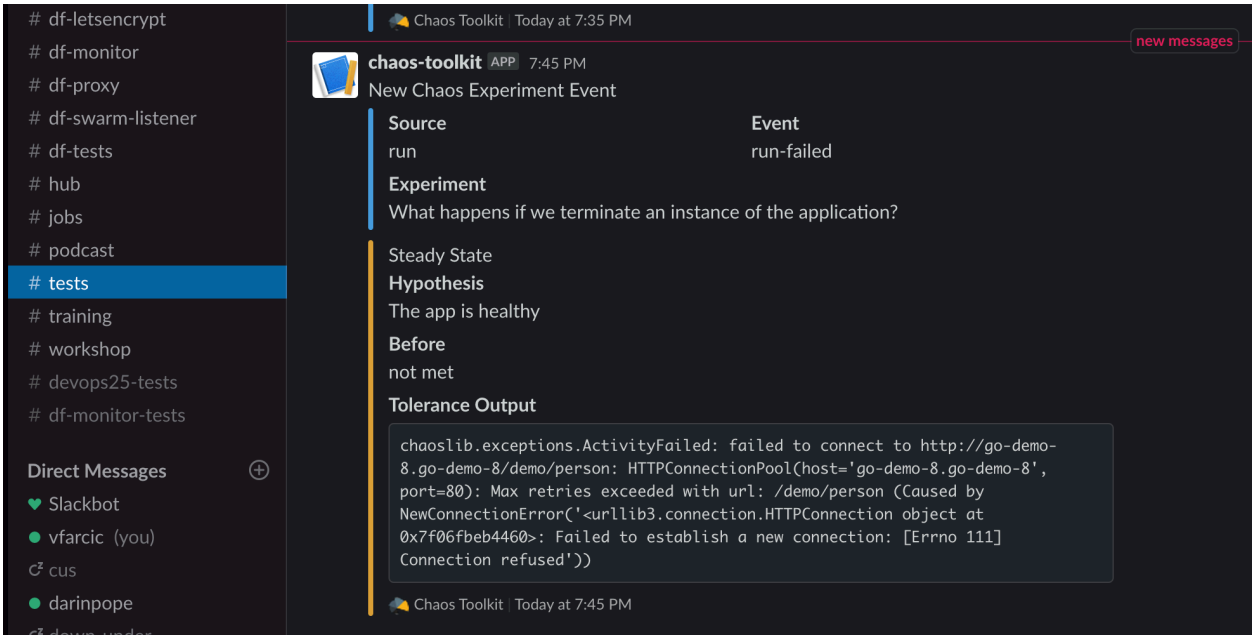

Figure 8-2: Chaos Toolkit Slack notifications about a failed experiment

From now on, we are receiving notifications only when experiments fail, instead of being swarmed with, more or less, useless information about successful executions. While it might be useful to see notifications of successful experiments a few times, sooner or later, we'd get tired of that, and we'd start ignoring them. So, we configured our Chaos Toolkit CronJob to send notifications only when one of the experiments fail.

Destroying What We Created

That's it. Yet another chapter is finished, and we are going to remove everything we created by deleting the Namespace `go-demo-8`.

```
cd ..

kubectl delete namespace go-demo-8
```

Everything we did is gone. Both the demo application and the CronJob were terminated, and you know what comes next. You can continue straight to the next chapter. If that's what you want, just go straight there. On the other hand, if you're going to take a break, it might be useful to destroy the cluster. You'll find the instructions at the bottom of the Gist you used to create it.

Executing Random Chaos

We saw how we can run chaos experiments limited to specific applications or specific conditions. We were, for example, removing instances of an application, and that was partly randomized. We were terminating random Pods, but they were still tied to a specific application. Similarly, we were messing with networking but, again, limited and filtered by a particular app. We were destroying nodes as well, but they were those where specific applications were running.

Now we're ready to spice it up a bit. We're going to make our experiments even more random. Instead of being limited to destructions and interruptions of specific components, we're going to start affecting completely random things. And that will bring us an additional level of insight that we might not get when we are limited to specific applications. We'll explore what happens when we exercise, more or less, random interruptions. For that, we will need to make some changes to our experiments if we are not going to limit them to a single application. We could start affecting the system on the cluster level, but that might be too big of a jump.

We'll start by trying to figure out how to affect random resource on the Namespace level. After that, we might move onto the cluster level. Through such goals, we might discover that the probes and steady-state hypotheses defined in Chaos Toolkit might not be sufficient. How do we know what the state that we want to observe is if that state is much bigger than a single application is? To be successful at a larger scale, we might need to bring additional tools into the mix. We might want to re-evaluate how we observe the state and how we'll get notified if something is wrong inside our cluster after doing completely random destruction.

All in all, we'll explore how to make chaos experiments more destructive, more random, and more unpredictable.

Gist with the commands

As you already know, every section that has hands-on exercises is accompanied by a Gist that allows you to copy and paste commands instead of bothering to type. This chapter is not an exception.

All the commands from this chapter are available in the 09-total.sh[75] Gist.

[75] https://gist.github.com/69bfc361eefd0a64cd92fdc4840f7aed

Creating A Cluster

We need a cluster to deploy our demo application and to run experiments. Whether you already have it or not depends on whether you destroyed your cluster at the end of the previous chapter. If you didn't destroy it, just skip to the next section right away. If you did destroy the cluster, feel free to use one of the Gists that follow. Or roll out your own cluster. It's up to you.

Docker Desktop and Minikube can be used only in some of the exercises.

- Docker Desktop with Istio: docker-istio.sh[76]
- Minikube with Istio: minikube-istio.sh[77]
- Regional and scalable GKE with Istio: gke-istio-full.sh[78]
- Regional and scalable EKS with Istio: eks-istio-full.sh[79]
- Regional and scalable AKS with Istio: aks-istio-full.sh[80]

Deploying The Application

Just like always, we need not only a cluster, but also the demo application. This chapter is not an exception.

We are going to deploy the same demo application that we used before, so we are going to go through this fast. There's nothing new in it.

We'll go to the `go-demo-8` directory and pull the latest version of the repository.

```
cd go-demo-8

git pull
```

Next, we'll create a Namespace called `go-demo-8` and add the label so that Istio knows that it should inject proxy sidecars automatically.

[76] https://gist.github.com/9a9752cf5355f1b8095bd34565b80aae
[77] https://gist.github.com/a5870806ae6f21de271bf9214e523b53
[78] https://gist.github.com/88e810413e2519932b61d81217072daf
[79] https://gist.github.com/d73fb6f4ff490f7e56963ca543481c09
[80] https://gist.github.com/b068c3eadbc4140aed14b49141790940

```
kubectl create namespace go-demo-8

kubectl label namespace go-demo-8 \
    istio-injection=enabled
```

We're going to apply the definition from the directory `k8s/app-full` and, once all the resources are created, we're going to wait until the Deployment of the API rolls out.

```
kubectl --namespace go-demo-8 \
    apply --filename k8s/app-full

kubectl --namespace go-demo-8 \
    rollout status deployment go-demo-8
```

Finally, to be sure that it works fine, we're going to send a request to the newly deployed demo application.

```
curl -H "Host: repeater.acme.com" \
    "http://$INGRESS_HOST?addr=http://go-demo-8"
```

Now we can switch back to chaos experiments.

Deploying Dashboard Applications

We're going to take chaos experiments to the next level. We'll randomize the targets by making them independent of specific applications. To do that, we need to rethink how do we validate the results of such experiments. Steady-state hypotheses in Chaos Toolkit will not get us there. They assume that we know in advance what to measure and what the desired state is, both before and after the experiments. If, for example, we go crazy and start removing random nodes, we will likely not be able to define in advance the state of the whole cluster or even the entire Namespace. Or, at least, we are probably not going to be able to do that in Chaos Toolkit format. That would be impractical, if not impossible.

What we need is a proper monitoring and alerting system that will allow us to observe the cluster as a whole and (potentially) get alerts when something goes wrong in any part of the system. With proper monitoring, dashboards, and alerting in place, we can start doing random destruction and catch many (if not all) anomalies in the system. So, our first action will be to introduce a few additional tools.

We're going to use Prometheus[81] for gathering and querying metrics. It's already running, so there's no need to deploy it. We're going to install Grafana[82] for visualizing those metrics, and we're going

[81] https://prometheus.io/
[82] https://grafana.com/

to use Kiali[83] for visualizing service mesh with Istio. But, before we do any of those things, we need to generate some traffic in our cluster that, in turn, will create metrics that we'll be able to observe. Otherwise, if our cluster is dormant and nothing is happening inside, we would not be able to see those metrics in action and explore monitoring and alerting.

So, the first thing we are going to do is to generate some traffic and, for that, we need to know the IP address through which we can send requests to the cluster in general, and the demo application in particular. We already have Istio Gateway, and you should have defined the environment variable `INGRESS_HOST`. Or, at least, you should have those if you run the commands from the Gists with the instructions on how to create a cluster. To be on the safe side, we're going to output the value of the `INGRESS_HOST` variable and confirm that it looks like a valid address or an IP.

```
1 echo $INGRESS_HOST
```

In my case, the output is `34.74.135.234`. If, in your case, that variable is empty, go back to the Gist for creating the cluster. Over there, you will find the instructions on how to assign a value into the `INGRESS_GATEWAY` variable.

Now that we know what the entry point to the cluster is, we're going to open a second terminal. Soon you'll see why we need it.

I tend to have two terminal windows occupying half of the screen each. That way, I can observe the outputs from both. Nevertheless, it's up to you how you'll arrange them.

We're going to define the same environment variable `INGRESS_HOST` in the second terminal window.

Please replace `[...]` with the value of the `echo` command from the first terminal.

```
1 export INGRESS_HOST=[...]
```

Next, we are going to create an infinite loop of requests to the demo application.

```
1 while true; do
2     curl -i -H "Host: repeater.acme.com" \
3         "http://$INGRESS_HOST?addr=http://go-demo-8/demo/person"
4     sleep 1
5 done
```

From now on, we're sending a stream of requests to the `repeater` with a one-second pause between each. The output should be a constant flow of responses similar to the one that follows.

[83] https://kiali.io/

```
HTTP/1.1 200 OK
date: Tue, 24 Mar 2020 03:26:47 GMT
content-length: 0
x-envoy-upstream-service-time: 5
server: istio-envoy
```

We are generating traffic, which, as you will soon see, is creating metrics.

Leave the loop running and go back to the first terminal.

Fortunately for us, Prometheus is already installed through Istio. But, Grafana and Kiali are missing. Nevertheless, that's not a big problem since both can be added to the Istio manifest by setting a few additional values.

```
istioctl manifest install \
    --set values.grafana.enabled=true \
    --set values.kiali.enabled=true \
    --skip-confirmation
```

I would not recommend that you change Istio manifest in that way. Instead, you should define the whole manifest in a YAML file, store it in Git, and track changes. However, this is not a book about Istio, and this is not a production cluster, so you'll forgive me for taking a shortcut.

What matters, in this context, is that now we have both Grafana and Kiali up and running in our cluster. Or, to be more precise, Kubernetes was instructed to create the needed resources.

We'll check the rollout status to be on the safe side and make sure that Grafana is indeed fully operational.

```
kubectl --namespace istio-system \
    rollout status deployment grafana
```

The output should state that the `deployment "grafana"` was `successfully rolled out`.

Grafana is up and running, and we can start using it.

Exploring Grafana Dashboards

Before we start exploring this specific setup of Grafana and Kiali, there are a couple of things that you should know. First of all, the default setup that we are using is not creating Ingress Gateways for those tools. Typically, you would create them so that there is proper access to those tools through a domain (or a subdomain). But, in the interest of brevity, we are going to skip that. This is not

really a book about Istio, nor about monitoring. Instead, we're going to use the `istioctl dashboard grafana` command that will create a temporary tunnel through which we'll be able to access the UI.

So, let's open the tunnel to the dashboard.

```
1 istioctl dashboard grafana
```

If, in your case, the dashboard doesn't open automatically, copy the address from the output and paste it into your favorite browser.

You should see the Grafana home page on your screen. Select the *dashboards* icon from the left-hand menu and select the *Manage* item. You should see the directory *istio*. Open it, and you will observe quite a few Istio-specific dashboards available.

We are not going to go through all those dashboards because, again, monitoring and Istio are not the subjects of this book. Instead, we'll have a rapid overview of one of the dashboards. Please select the *Istio Mesh Dashboard*.

What we can see in front of us is a high-level overview of the services controlled by Istio. There aren't many since we're running only two applications. The database is excluded since it doesn't have Istio VirtualService associated with it.

That dashboard provides an overview of some basic metrics. In the top part, we have *Global Request Volume*, *Global Request Rate* (non-5xx responses), and so on. Everything seems to be OK. For example, *100%* of the requests are successful. More importantly, there is no data for 4xx and 5xx responses. If our application would start generating errors, then we would see data in those boxes. In the bottom part, there is a list of services; *go-demo-8* and *repeater*. They are accompanied by additional information like, for example, latency and success rate.

If there were something wrong with the network traffic, we would like to be able to see that through a dashboard like that one. Actually, we would likely be able to detect other anomalies not directly related to networking. For example, if an application is unavailable, it would not be able to process requests, and that would result in 5xx responses. Quite a few other things can be deduced through networking, no matter whether issues are directly related, or not.

If we'd detect an anomaly, we could select one of those services and gather more information.

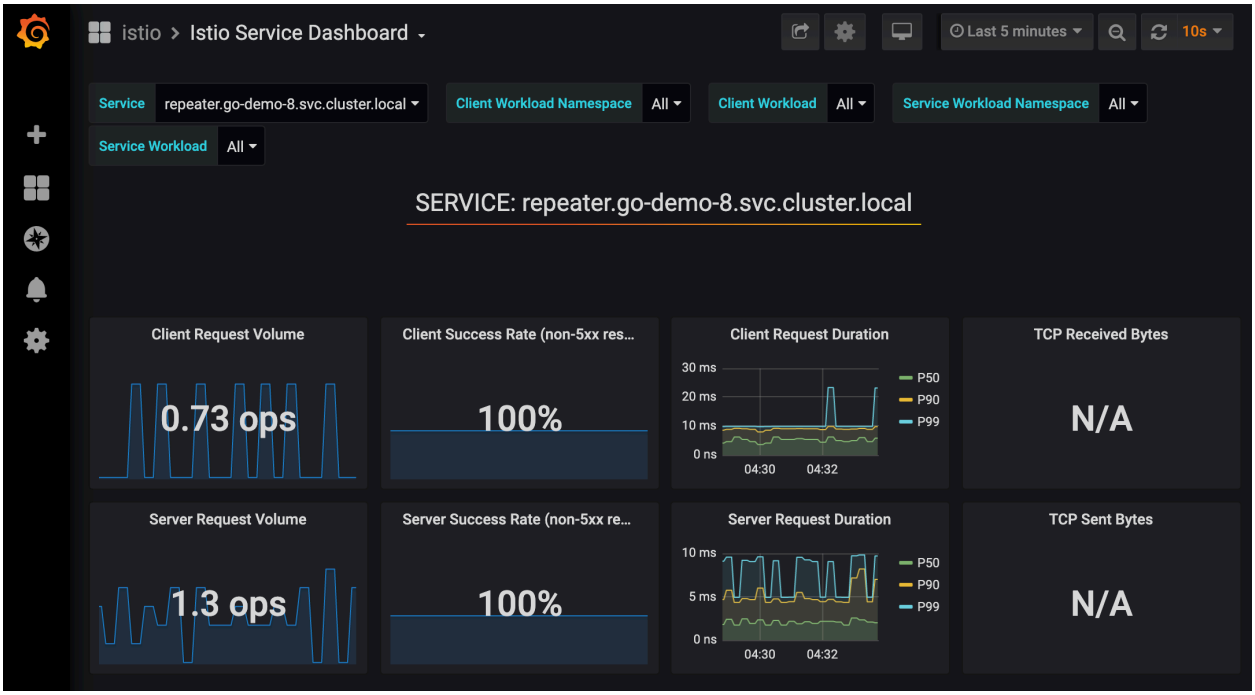

Figure 9-1: Grafana dashboard

I'll let you explore the dashboards yourself. For our purposes, that main screen on the dashboard should be enough.

Exploring Kiali Dashboards

We're about to explore Kiali as an alternative, or a complement, to Grafana dashboards. But, before we do that, please cancel the tunnel to Grafana by pressing *ctrl+c*

Just like with Grafana, we won't go into details, but only take a quick look at Kiali. I will assume that, later on, you'll explore it in more detail. After all, it's just a UI.

Unlike Grafana, which is a general dashboarding solution that works with many different sources of metrics, Kiali is specific to service meshes. It visualizes network traffic.

Let's open it and see what we'll get.

```
1 istioctl dashboard kiali
```

Just like with Grafana, we created a tunnel to Kiali, and you should see it in your browser.

We're presented with a screen that expects us to enter a username and password. Unlike Grafana that comes without authentication by default, Kiali requires us to log in. However, we cannot do that just yet because we did not create a username and password. Kiali expects to find authentication info in a Kubernetes Secret, so we need to create it. To do that, we'll stop the tunnel first.

Press *ctrl+c* to stop the tunnel and to release the terminal input, before we create the Secret.

```
echo "apiVersion: v1
kind: Secret
metadata:
  name: kiali
  labels:
    app: kiali
type: Opaque
data:
  username: $(echo -n "admin" | base64)
  passphrase: $(echo -n "admin" | base64)" \
    | kubectl --namespace istio-system \
    apply --filename -
```

The command we just executed output a YAML definition of a Kubernetes Secret. We used `base64` to encode the `username` and the `passphrase` since that's the format Secrets expect. The output was piped into the `kubectl apply` command. As a result, now we have the Secret that will be used by Kiali.

However, Kiali can detect authentication Secrets only during boot, so we'll need to restart it.

```
kubectl --namespace istio-system \
    rollout restart deployment kiali
```

Kiali was restarted, and, as a result, it should have caught the Secret. We should be able to log in.

Let's go back to the dashboard and see whether we can authenticate.

```
istioctl dashboard kiali
```

Please login with *admin* as both the username and the password.

Now that we're authenticated, we can see the overview screen.

Click the *Graph* item from the left-hand menu and select the Namespace *go-demo-8*.

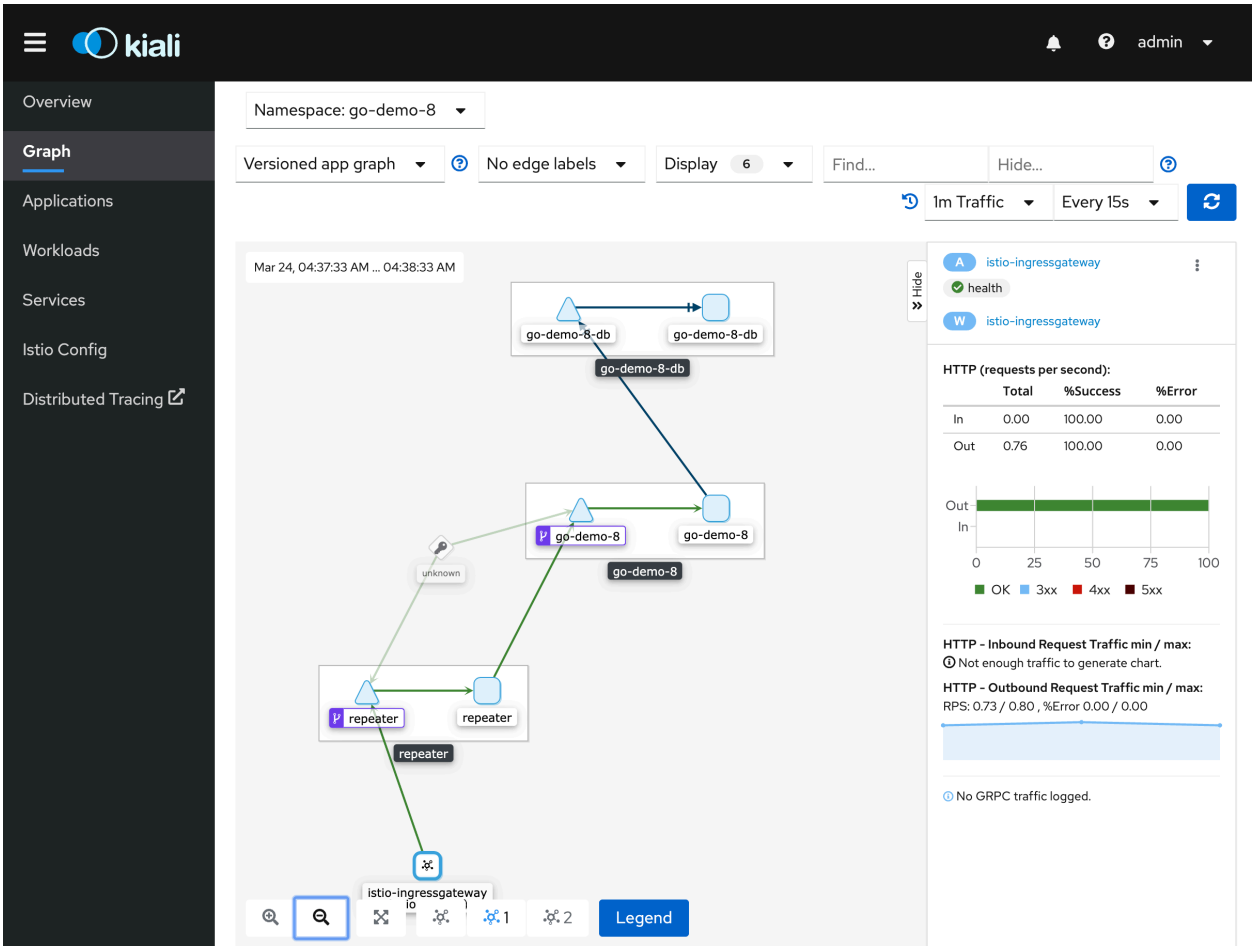

Figure 9-2: Kiali dashboard

We can see all the applications in the selected Namespace. There are three of them, and we can see that the traffic flows from the Gateway to the *repeater*. From there, it's being forwarded to *go-demo-8*, which communicates with the database (*go-demo-8-db*). It is a handy way to deduce the flow of traffic and potential problems related to the communication between different components.

Just like with Grafana, the goal is not to show you everything you can do with Kiali. I believe that you are capable of exploring it in more detail yourself.

What matters, for now, is that now we have the tools that allow us to visualize what is happening on the networking (Kiali), as well as on any other levels (Grafana). Since Kiali is focused only on service mesh and networking, it is probably a better source of information related to the flow of traffic. Grafana, on the other hand, is more general, and it can visualize any metric stored in quite a few different sources. In our case, that's Prometheus, but it could be others, like, for example, Elasticsearch. Both have their strengths and weaknesses, and, for the purpose of our experiments, we'll use both.

Soon we'll jump back to Grafana, so please stop the tunnel to Kiali by pressing *ctrl+c*.

Before we proceed, I must clarify something. We'll be watching dashboards for a while. However,

they are not the end goal, but, instead, a means to an end. We should use them to detect patterns. They are helpful to find out what goes where, and how, and when. But, once we identify the patterns, we might stop watching dashboards and create alerts. The ultimate goal should not be to stare at screens with "pretty colors", but rather to be notified when events that require our attention happen.

My preference for alerting is Alert Manager[84] that is available in Prometheus. That does not mean that other alerting solutions are not good, but rather that I feel more comfortable with it than with other similar applications. Being integrated with Prometheus helps.

Anyways, the goal is not to watch dashboards indefinitely, but only to observe them as a learning experience that will allow us to create alerts. The final objective is to be notified whenever there is something wrong inside of our clusters.

Now that we added a few new tools to the mix, even though we didn't explore them in detail, we'll see how we can use them in conjunction with chaos experiments.

Preparing For Termination Of Instances

Now we have metrics stored in Prometheus, and we can visualize them using Grafana and Kiali. We should be ready to create even more mayhem than before.

What can we do?

Let's not introduce anything drastically new. Let's destroy a Pod. Now you might say, "Hey, I already know how to destroy a Pod. You showed me that." If that's what you're thinking, you're right. Nevertheless, we are going to terminate, but, this time, we are not going to target a specific app. We are going to destroy a completely random Pod in the `go-demo-8` Namespace.

You will not see a significant difference between destroying a Pod of the `go-demo-8` application and destroying a completely random Pod from the `go-demo-8` Namespace. We are not running much in that Namespace right now. But, in a real-world situation, you would have tens, or even hundreds of applications running in, let's say, the production Namespace, In such a case, destroying a Pod selected randomly among many applications could have quite unpredictable effects. But, we don't have that many. We have only three apps (`repeater`, `go-demo-8`, and MongoDB). Still, even with only those three, randomizing which Pod will be terminated might result in unexpected results.

We'll leave speculations for some other time, and we'll go ahead and destroy a random Pod. As always, we'll start by taking a quick look at the definition we're going to apply.

```
1 cat k8s/chaos/experiments-any-pod.yaml
```

The output is as follows.

[84] https://prometheus.io/docs/alerting/alertmanager/

```
---

apiVersion: v1
kind: ConfigMap
metadata:
  name: chaostoolkit-experiments
data:
  health-instances.yaml: |
    version: 1.0.0
    title: What happens if we terminate an instance?
    description: Everything should continue working as if nothing happened
    tags:
    - k8s
    - pod
    - deployment
    steady-state-hypothesis:
      title: The app is healthy
      probes:
      - name: all-apps-are-healthy
        type: probe
        tolerance: true
        provider:
          type: python
          func: all_microservices_healthy
          module: chaosk8s.probes
          arguments:
            ns: go-demo-8
    method:
    - type: action
      name: terminate-app-pod
      provider:
        type: python
        module: chaosk8s.pod.actions
        func: terminate_pods
        arguments:
          rand: true
          ns: go-demo-8
      pauses:
        after: 10
```

That ConfigMap is very similar to those we used in the previous chapter. It defines a single experiment that will be available to processes inside the cluster. The hypothesis of the experiment is the same old one that validates whether all the applications in the `go-demo-8` Namespace are

healthy. Then, we have the `method` that will destroy a random Pod inside the `go-demo-8` Namespace, and pause for 10 minutes. What matters is that, this time, we are not limiting the Pod selector to a specific application. Any Pod in that Namespace will be eligible for termination.

Before we `apply` that definition, we'll need to create a Namespace. Since we are going to destroy a random Pod from the `go-demo-8` Namespace, it probably wouldn't be a good idea to run our experiments there. We'd risk terminating a Pod of the experiment. Also, if we are exploring how to make the destruction more randomized, we might choose to run experiments across more than one Namespace. So, we are going to create a new Namespace that will be dedicated to chaos experiments.

```
kubectl create namespace chaos
```

Now we can apply the definition with the ConfigMap that has the experiment we want to run.

```
kubectl --namespace chaos apply \
    --filename k8s/chaos/experiments-any-pod.yaml
```

Remember that experiment validates whether all the applications in the `go-demo-8` Namespace are running correctly and that the `method` will terminate a random Pod from that namespace. We're not choosing from which Deployment or StatefulSet that Pod should come from. Also, please note that the verification relies on Kubernetes health checks, which are not very reliable.

We want to run an experiment that will perform actions inside a different Namespace. We want to separate the experiments from the resources manipulated through actions. For that, we'll need to define a ServiceAccount which, this time, will be slightly different than the one we used before.

```
cat k8s/chaos/sa-cluster.yaml
```

The output is as follows.

```
---

apiVersion: v1
kind: ServiceAccount
metadata:
  name: chaostoolkit

---

apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: chaostoolkit
```

```
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: chaostoolkit
    namespace: chaos
```

This time, we're using `ClusterRoleBinding` instead of RoleBinding. In the past, our experiments were running in the same Namespace as the applications that were targeted. As a result, we could use RoleBinding, which is namespaced. But now we want to be able to run the experiments in the `chaos` Namespace and allow them to execute some actions on resources in other Namespaces. By binding the `ServiceAccount` to the `ClusterRoleBinding`, we're defining cluster-wide permissions. In turn, that binding is using pre-defined `ClusterRole` called `cluster-admin`, which is available in every Kubernetes distribution (that I know of). It should be easy to guess the level of permissions a role called `cluster-admin` provides.

All in all, with that `ServiceAccount`, we'll be able to do almost anything anywhere inside of the cluster.

Let's `apply` the definition so that the `ServiceAccount` is available for our future experiments.

```
kubectl --namespace chaos apply \
    --filename k8s/chaos/sa-cluster.yaml
```

The last thing we need is a definition of a CronJob that will run our experiments.

```
cat k8s/chaos/periodic-fast.yaml
```

```
---

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: health-instances-chaos
spec:
  concurrencyPolicy: Forbid
  schedule: "*/2 * * * *"
  jobTemplate:
    metadata:
      labels:
        app: health-instances-chaos
```

```
14      spec:
15        activeDeadlineSeconds: 600
16        backoffLimit: 0
17        template:
18          metadata:
19            labels:
20              app: health-instances-chaos
21            annotations:
22              sidecar.istio.io/inject: "false"
23          spec:
24            serviceAccountName: chaostoolkit
25            restartPolicy: Never
26            containers:
27            - name: chaostoolkit
28              image: vfarcic/chaostoolkit:1.4.1
29              args:
30              - --verbose
31              - run
32              - --journal-path
33              - /results/health-instances.json
34              - /experiment/health-instances.yaml
35              env:
36              - name: CHAOSTOOLKIT_IN_POD
37                value: "true"
38              volumeMounts:
39              - name: experiments
40                mountPath: /experiment
41                readOnly: true
42              - name: results
43                mountPath: /results
44                readOnly: false
45              resources:
46                limits:
47                  cpu: 20m
48                  memory: 64Mi
49                requests:
50                  cpu: 20m
51                  memory: 64Mi
52            volumes:
53            - name: experiments
54              configMap:
55                name: chaostoolkit-experiments
56            - name: results
```

```
            persistentVolumeClaim:
              claimName: chaos

---

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: chaos
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

That CronJob is very similar to the one we used in the previous chapter. However, this time, it'll be scheduled to run every two minutes since I wanted to save you even more from waiting for the outcomes. The only other significant difference is that it'll run a different experiment. It'll periodically execute `health-instances.yaml`, which is defined in the ConfigMap we created earlier.

Terminating Random Application Instances

Let's apply the CronJob that we just explored and see what'll happen.

```
kubectl --namespace chaos apply \
    --filename k8s/chaos/periodic-fast.yaml
```

Next, we'll output all the CronJobs from the Namespace `chaos`.

```
kubectl --namespace chaos get cronjobs
```

If the `LAST SCHEDULE` is set to `<none>`, you might need to wait for a while longer (up to two minutes), and re-run the previous command. Once the first Job is created, the output should be similar to the one that follows.

```
NAME                   SCHEDULE    SUSPEND ACTIVE LAST SCHEDULE AGE
health-instances-chaos */2 * * * * False   1      8s            107s
```

Next, we'll take a look at the Jobs created by that CronJob.

```
kubectl --namespace chaos get jobs
```

Just as we had to wait until the CronJob creates the Job, not we need to wait until the Job creates the Pod and the experiment inside it finishes executing. Keep re-running the previous command until the `COMPLETIONS` column is set to `1/1`. The output should be similar to the one that follows.

```
NAME                         COMPLETIONS DURATION AGE
health-instances-chaos-... 1/1         93s      98s
```

From this moment on, the results I will present might differ from what you'll observe on your screen. Ultimately, we'll end up with the same result, even though the time you might need to wait for that might differ.

Finally, we'll retrieve the Pods in the `chaos` Namespace and check whether there was a failure.

```
kubectl --namespace chaos get pods
```

The output is as follows.

```
NAME                         READY STATUS    RESTARTS AGE
health-instances-chaos-... 0/1   Completed 0        107s
```

In my case, it seems that the first run of the experiment was successful. To be on the safe side, we'll take a look at the Pods of the demo applications.

```
kubectl --namespace go-demo-8 \
    get pods
```

The output is as follows.

```
NAME             READY STATUS  RESTARTS AGE
go-demo-8-...    2/2   Running 2        27m
go-demo-8-...    2/2   Running 3        27m
go-demo-8-db-... 2/2   Running 0        27m
repeater-...     2/2   Running 0        27m
repeater-...     2/2   Running 0        53s
```

All the Pods are running, and, at least in my case, it seems that the experiment did not detect any anomaly. We can confirm that it indeed terminated one of the Pods by observing the `AGE` column. In my case, one of the `repeater` Pods is fifty-three seconds old. That must be the one that was created

after the experiment removed one of the replicas of the `repeater`. The experiment indeed chose a random Pod, and, in my case, the system seems to be resilient to such actions.

It might conclude that the experiment did not uncover any weakness in the system. That's excellent news. Isn't it?

I'd say that we deserve a break. We'll open the dashboard in Grafana and let it stay on the screen while we celebrate the success. We finally have an experiment that does not uncover any fault in the system.

```
istioctl dashboard grafana
```

Please open the *Istio Mesh Dashboard*. You should know how to find it since we already used it in the previous chapters.

In my case, everything is "green", and that is yet another confirmation that my system was not affected negatively by the experiment. Not only that the probe passed, but I can see that all the metrics are within the thresholds. There are no responses with *4xx* nor *5xx* codes. The *Success Rate* is *100%* for both Services. Life is good, and I do deserve a break as a reward. I'll leave the dashboard on the screen, while the experiment is being executed every two minutes. Since it destroys a random Pod, next time, it could be any other. It's not always going to be the `repeater`.

Right now, you must use your imagination and picture me working on something else using my primary monitor, while Grafana keeps running in my secondary display.

Minutes are passing. The CronJob keeps creating Jobs every two minutes, and the experiments are running one after another. I'm focused on my work, confident that everything is OK. I know that there's nothing to worry about because everything is "green" in Grafana. My system is rock-solid. Everything works. Life is good.

A while later, my whole world turns upside down. All of a sudden, I can see an increase in *5xx* responses. The "Success Rate" turned red for both *repeater* and *go-demo-8* Services. Something is terribly wrong. I have an issue that was not detected by any of the previous experiments. Even the one that I'm running right now uncovered an issue only after it was executed quite a few times. To make things more complicated, the experiments are successful, and the problem can be observed only through the Grafana dashboard.

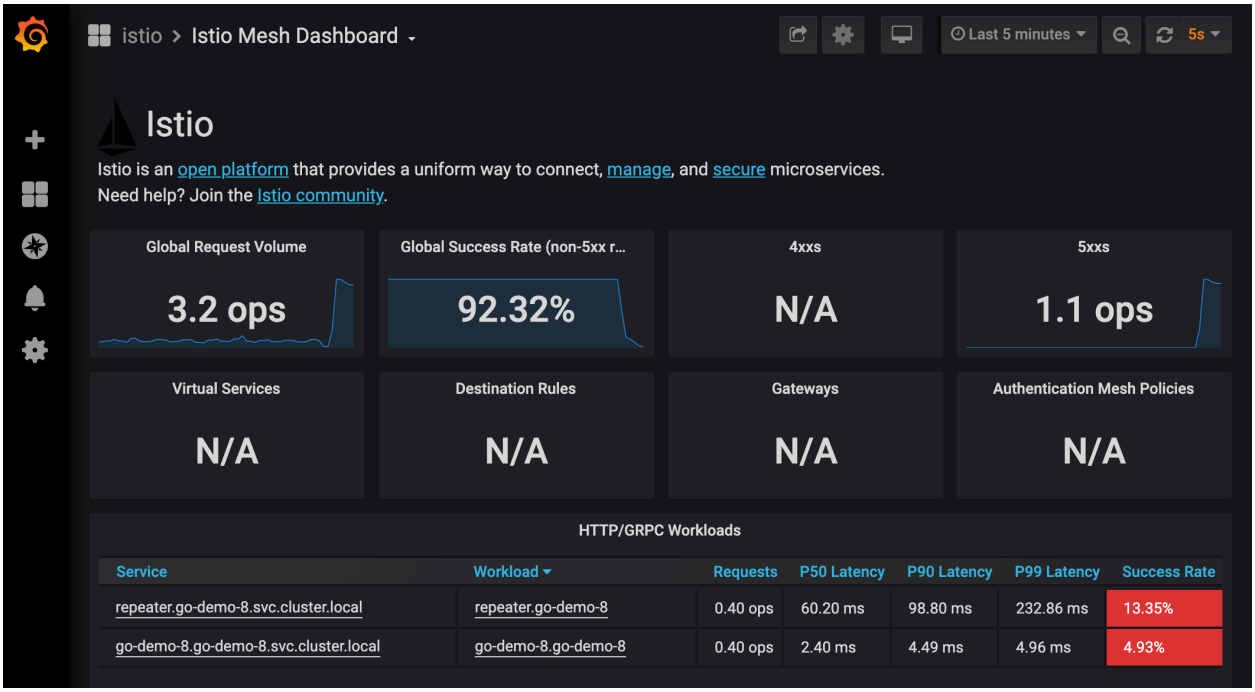

Figure 9-3: Grafana dashboard after an unsuccessful experiment

You might have reached the same point as I did, or everything might still be green. In the latter case, be patient. Your system will fail, as well. It's just a matter of time. Be patient.

Unlike the previous experiments, this time, we did not destroy or disrupt a specific target. We did not predict what will be the adverse effect. But we did observe through the dashboard that there is a problem affecting both the *repeater* and *go-demo-8*. Nevertheless, we did uncover a potential issue. But why did it appear after so much time? Why wasn't it discovered right away from the first execution of the experiment? After all, we were destroying Pods before, and we did fix all the issues we found. Yet, we're in trouble again.

So far, we know that there is a problem detected through the Grafana dashboard. Let's switch to Kiali and see whether we'll see something useful there.

Please go back to the terminal, cancel the tunnel towards Grafana by pressing *ctrl+c*, and execute the command that follows to open Kiali.

```
1 istioctl dashboard kiali
```

What do we have there? We can see that there are three applications. Two of them are red.

Select *Graph* from the left-hand menu, and choose the *go-demo-8* Namespace from the top. You should see something similar to figure 9-4.

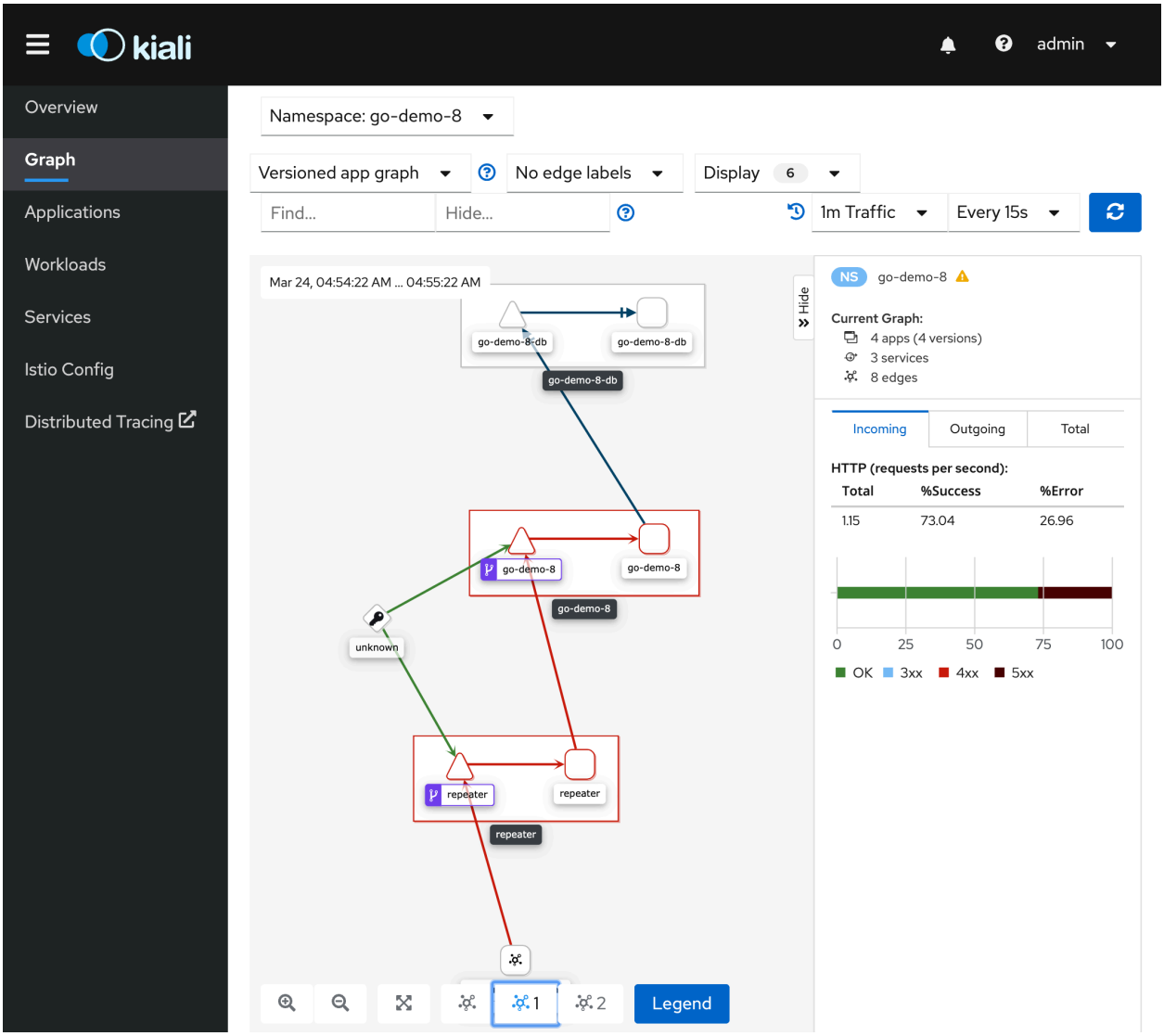

Figure 9-4: Kiali dashboard after an unsuccessful experiment

We can see that the traffic is blocked. It's red all around. External traffic goes to the *repeater*. From there, it goes to *go-demo-8*, and then it fails to reach the database. The problem might be that our database was destroyed.

Our database is not replicated. It makes sense that one of the experiments eventually terminated the database. That could result in the crash of all the applications that, directly or indirectly, depend on it. Destroying a Pod of the *repeater* is okay because there are two replicas of it. The same can be said for *go-demo-8* as well. But the database, with the current setup, is not highly available. We could predict that destroying the only Pod of the DB results in downtime.

All in all, eventually, one of the experiments randomly terminated the database, and that resulted in failure. The issue should be temporary, and the system should continue operating correctly as soon as Kubernetes recreates the failed Pod. It shouldn't take more than a few seconds, a minute at the

most, to get back to the desired state. Yet, more time already passed, and things are still red in Kiali. Or, at least, that's what's happening on my screen.

Let's check the Pods and see whether we can deduce what's going on.

Please go back to the terminal, cancel the tunnel towards Kiali by pressing *ctrl+c*, and execute the command that follows to retrieve the Pods.

```
kubectl --namespace go-demo-8 \
    get pods
```

The output is as follows.

```
NAME              READY STATUS  RESTARTS AGE
go-demo-8-...     2/2   Running 0        5m2s
go-demo-8-...     2/2   Running 3        37m
go-demo-8-db-...  2/2   Running 0        3m1s
repeater-...      2/2   Running 0        37m
repeater-...      2/2   Running 0        57s
```

Everything looks fine, even though we know that there's something wrong. All the Pods are running, and, in my case, the database was recreated three minutes ago.

Let's try to figure out what's the cause of the issue.

The database was destroyed, and Kubernetes recreated it a few seconds later. The rest of the Pods is running, and, in my case, the last one terminated was the *repeater*. How did we observe that the system is messed up even after everything went back to normal? The system is not operational, even though the database is up and running. What could be the most likely the cause of that issue?

When *go-demo-8* boots, it connects to the database. My best guess is that the likely culprit is the logic of the code of the demo application. It probably does not know how to re-establish the connection to the database after it gets back online. The code of the app was perhaps written in a way that, if the connection is broken, it does not have sufficient logic to re-establish it later. That's a bad design. We have a problem with the application itself. We probably found an issue that we did not capture in the past experiments.

We already demonstrated in the previous chapters that, when the database is terminated, it produces a short downtime since it is not replicated. The new finding is that the code of the application itself (*go-demo-8*) does not know how to re-establish the connection. So, even if we would make the database fault-tolerant, and even if Kubernetes would recreate failed replicas of the database, the damage is permanent due to bad code. We would need to re-design the application to make sure that when the connection to the database is lost, the app itself tries to re-establish it.

Even better, we could make the application fail when the connection to the database is lost. That would result in the termination of the container where it's running, and Kubernetes would recreate

it. That process might repeat itself a few times but, once the database is up-and-running again, the restart of the container would result in a new boot of the application that would be able to re-establish the connection. Or, we can opt for a third solution.

We could also change the health check. Right now, the address is not targeting an endpoint that uses the database. As a result, the health check seems to be working fine, and Kubernetes sees no reason to terminate containers of *go-demo-8*.

No matter the solution we should employ, the vital thing to note is that we found yet another weak spot in our system. Knowing what might go wrong is the most important thing. Solving an issue is often not a problem, as long as we know what the issue is.

There's no need for us to continue terminating random Pods since now we know that there is a problem that needs to be resolved. So, we're going to delete the CronJob.

```
kubectl --namespace chaos delete \
    --filename k8s/chaos/periodic-fast.yaml
```

I'll leave it to you to think which proposed solution you'd employ, if any. Think of it as yet another homework. For now, we're going to restart *go-demo-8*. That should re-establish the connection to the database and fix the issue. To be more precise, that will not solve the problem but act as a workaround that will allow us to move to the next subject. The "real" fix is the task I already assigned to you.

```
kubectl --namespace go-demo-8 \
    rollout restart deployment go-demo-8
```

Disrupting Network Traffic

We were terminating random Pods, no matter which application they belong to. Can we do something similar to networking? Can we disrupt a random network or, to be more precise, a random Istio VirtualService? Yes, we can, but not by using the out-of-the-box solution in Chaos Toolkit. At the time of this writing (April 2020), the Istio plugin does not allow us to operate on a random VirtualService. So, we cannot say disrupt a random network. If we would like to do that, we would need to create our own implementation, which should be relatively straightforward. We could write the script that would, let's say, retrieve all the VirtualServices, pick a random one, and disrupt it. Writing such a script should not be a big deal. Or, even better, we should contribute to the project by implementing such a feature in Chaos Toolkit Extension for Istio Fault Injection[85].

As you already know, every once in a while, I will give you tasks to do yourself, and I promised that they are not going to be trivial.

Your next task is to disrupt a random Istio VirtualService. You can do this by writing a script that, first of all, lists all the VirtualServices in a Namespace, then selects a random one, and, finally, it

[85] https://github.com/chaostoolkit-incubator/chaostoolkit-istio

disrupts it in one way or another. Explore how you can modify Istio VirtualService and do some nasty things to it. If you choose to accept the challenge, once you're done, try to implement the same in the Istio plugin. Make a pull request and contribute back to the community.

If you do not feel confident with Istio VirtualServices, you might want to check the course Canary Deployments To Kubernetes Using Istio and Friends[86] on Udemy. The course does not speak directly about Istio, but it does show in more detail what we're trying to do right now.

In any case, now you have a task. Create a script or contribute to the plugin. Use it inside a chaos experiment. Come back when you're done, and we're going to explore more stuff.

Preparing For Termination Of Nodes

Now we know how to affect not only individual applications, but also random ones running in a Namespace, or even the whole cluster. Next, we'll explore how to randomize our experiments on the node level as well.

In the past, we were terminating or disrupting nodes where a specific application was running. What we're going to do next is try to figure out how to destroy a completely random node. It will be without any particular criteria. We'll just do random stuff and see how your how that affects our cluster. If we're lucky, such actions will not result in any adverse result. Or maybe they will. We'll soon find out.

The reason we can do that now, and we couldn't do that before, is because the steady-state hypothesis of our experiments was not enough. If we destroy something (almost) completely random, then any part of the system can be affected. We cannot use the Chaos Toolkit hypothesis to predict what should be the initial state, nor what should be the state after some destructive cluster-wide actions. To be more precise, we could do that, but it would be too complicated, and we would be trying to solve the problem with the wrong tool.

Now we know that we can use Prometheus to store metrics and that we can monitor our system through dashboards like Grafana and Kiali. We could, and should, go further and, for example, create alerts that will notify us when any part of the system is misbehaving.

Now we are ready to go full throttle and run our experiments on the cluster level.

Let's take a look at yet another YAML definition.

```
1 cat k8s/chaos/experiments-node.yaml
```

That definition should not contain anything truly new. Nevertheless, there are a few details worth explaining. To make it simpler, we'll take a look at the `diff` between that and the previous definition. That will help us spot the differences between the two.

[86] https://www.udemy.com/course/canary-deployments-to-kubernetes-using-istio-and-friends/?referralCode=75549ECDBC41B27D94C4

```
diff k8s/chaos/experiments-any-pod.yaml \
    k8s/chaos/experiments-node.yaml
```

The output is as follows.

```
40c40,68
<
---
>   node.yaml: |
>     version: 1.0.0
>     title: What happens if we drain a node
>     description: All the instances are distributed among healthy nodes and the app\
lications are healthy
>     tags:
>     - k8s
>     - node
>     method:
>     - type: action
>       name: drain-node
>       provider:
>         type: python
>         func: drain_nodes
>         module: chaosk8s.node.actions
>         arguments:
>           label_selector: beta.kubernetes.io/os=linux
>           count: 1
>           delete_pods_with_local_storage: true
>       pauses:
>         after: 180
>     rollbacks:
>     - type: action
>       name: uncordon-node
>       provider:
>         type: python
>         func: uncordon_node
>         module: chaosk8s.node.actions
>         arguments:
>           label_selector: beta.kubernetes.io/os=linux
```

We can see that we have a completely new experiment called `node.yaml`, with the `title`, the `description`, and all the other things we normally have in experiments. It doesn't have any steady-state hypothesis, because we really don't know what the state of the whole cluster should be. So we

are skipping the steady-state, but we are using the `method`. It will remove one of the nodes that have a certain label selector.

The only reason why we're setting the `label_selector` to `beta.kubernetes.io/os=linux` is to avoid draining windows nodes if there are any. We don't have them in the cluster (if you used the Gist I provided). Nevertheless, since you are not forced to use those Gists to create a cluster, I couldn't be sure that you do not have it mixes with Windows servers. Our focus is only on Linux.

To be on the safe side, describe one of your nodes, and confirm that the label indeed exists. If that's not the case, please replace its value with whichever label is used in your case.

Further down, we can see that we also set `delete_pods_with_local_storage` to `true`. With it, we'll ensure that Pods with the local storage will be deleted before the node is drained. Otherwise, the experiment would not be able to perform the action since Kubernetes does not allow draining of nodes with local storage, given that they are tied to that specific node.

All in all, we'll drain a random node (as long as it's based on Linux). And then, we're going to pause for `180` seconds. Finally, we'll roll back by un-cordoning the nodes, and, that way, we'll restore them to their original state.

There's one potentially important thing you should know before we proceed. In hindsight, I should have said it at the beginning of this section. Nevertheless, better late than never.

The experiment we are about to run **will not work with Docker Desktop or Minikube**. If that's the Kubernetes distribution you're running, you can observe the output in here, since you will not be able to run the experiment. Docker Desktop and Minikube have only one node, so draining one would mean draining the whole cluster, including the control plane.

Let's `apply` this definition and update our existing ConfigMap.

```
kubectl --namespace chaos apply \
    --filename k8s/chaos/experiments-node.yaml
```

Next, we're going to take a look at yet another CronJob.

```
cat k8s/chaos/periodic-node.yaml
```

The output, limited to the relevant parts, is as follows.

```
---

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: nodes-chaos
spec:
  concurrencyPolicy: Forbid
  schedule: "*/5 * * * *"
  jobTemplate:
    ...
    spec:
      activeDeadlineSeconds: 600
      backoffLimit: 0
      template:
        metadata:
          labels:
            app: health-instances-chaos
          annotations:
            sidecar.istio.io/inject: "false"
        spec:
          serviceAccountName: chaostoolkit
          restartPolicy: Never
          containers:
          - name: chaostoolkit
            image: vfarcic/chaostoolkit:1.4.1
            args:
            - --verbose
            - run
            - --journal-path
            - /results/node.json
            - /experiment/node.yaml
            env:
            - name: CHAOSTOOLKIT_IN_POD
              value: "true"
            volumeMounts:
            - name: experiments
              mountPath: /experiment
              readOnly: true
            - name: results
              mountPath: /results
              readOnly: false
            resources:
```

```
44                limits:
45                  cpu: 20m
46                  memory: 64Mi
47                requests:
48                  cpu: 20m
49                  memory: 64Mi
50            volumes:
51            - name: experiments
52              configMap:
53                name: chaostoolkit-experiments
54            - name: results
55              persistentVolumeClaim:
56                claimName: chaos
57
58  ---
59
60  kind: PersistentVolumeClaim
61  apiVersion: v1
62  metadata:
63    name: chaos
64  spec:
65    accessModes:
66      - ReadWriteOnce
67    resources:
68      requests:
69        storage: 1Gi
```

We can see that it is, more or less, the same CronJob as the one we used before. There are only a few minor differences.

The `schedule` is now increased to five minutes because it takes a while until we drain and, later on, un-cordon a node. I raised it from two minutes we had before to account for the fact that the experiment will take longer to run. The other difference is that, this time, we are running the experiment defined in `node.yaml` residing in the newly updated ConfigMap.

Apart from having a different `schedule` and running an experiment defined in a different file, that CronJob is exactly the same as the one we used before.

Terminating Random Nodes

Let's apply the new definition of the CronJob.

```
kubectl --namespace chaos apply \
    --filename k8s/chaos/periodic-node.yaml
```

Assuming that you left the loop that sends requests running in the second terminal, we should be able to observe that the demo application keeps responding with `200`. At the moment, the demo application seems to be working correctly.

As you already know, we'll need to wait for a while until the first Job is created, and the experiment is executed.

We'll retrieve CronJobs to make it more interesting than staring at the blank screen.

```
kubectl --namespace chaos get cronjobs
```

After a while, and a few repetitions of the previous command, the output should be similar to the one that follows.

```
NAME        SCHEDULE    SUSPEND ACTIVE LAST SCHEDULE AGE
nodes-chaos */5 * * * * False   0      1m01s         5m2s
```

Once there is something other than `<none>` in the `LAST SCHEDULE` column, we can proceed knowing that a Job was created. That means that our experiment is running right now.

Let's retrieve Jobs and confirm that everything looks correct so far.

```
kubectl --namespace chaos get jobs
```

The output is as follows.

```
NAME            COMPLETIONS DURATION AGE
nodes-chaos-... 1/1         2m5s     4m46s
```

In my case, it is running for over two minutes (`2m5s`), so I (and probably you as well) will need a bit more patience. We need to wait until the Job is finished executing. If you're in the same position, keep repeating the previous command.

Next, to be entirely on the safe side, we'll retrieve Pods as well.

```
kubectl --namespace chaos get pods
```

The output is as follows.

```
NAME             READY STATUS            RESTARTS AGE
nodes-chaos-... 0/1   Completed         0        4m55s
nodes-chaos-... 0/1   ContainerCreating 0        4s
```

Sufficient time passed, and, in my case, the first experiment finished, and the next one is already running for four seconds.

Finally, the last thing we need to do is retrieve the nodes and see what's going on over there.

```
kubectl get nodes
```

The output is as follows.

```
NAME    STATUS ROLES  AGE   VERSION
gke-... Ready  <none> 3m37s v1.15.9-gke.22
gke-... Ready  <none> 55m   v1.15.9-gke.22
gke-... Ready  <none> 55m   v1.15.9-gke.22
gke-... Ready  <none> 55m   v1.15.9-gke.22
```

In my case, we can see that I have four nodes. Everything looks healthy, at least from nodes perspective. The experiment drained a node, and then it un-cordoned it, thus enabling it back for scheduling.

Now comes the most critical step when running experiments that are not focused on a single application.

Let's take a look at Grafana and see what we have over there.

```
istioctl dashboard grafana
```

What can we see from Grafana? Is there anything useful there?

Please open the *Istio Mesh Dashboard*.

Everything seems to be working correctly. Nothing terrible happened when we drained a node. Or, at least, everything seems to be working correctly from the networking perspective.

Next, we'll try to see whether we can spot anything from Kiali.

Please stop the tunnel to Grafana by pressing *ctrl+c*, and open Kiali through the command that follows.

```
istioctl dashboard kiali
```

Go through different graphs, and iterate through different Namespaces. If your situation is the same as mine, you should see that everything seems to be okay. Everything works well. There are no issues.

Please stop the tunnel to Kiali by pressing *ctrl+c*.

Draining a completely random node could affect anything in a cluster. Yet, we could not prove that such actions are disastrous. In parallel, the experiment keeps being executed every five minutes, so the nodes keep being drained. We can confirm that by retrieving the nodes through the `kubectl get nodes` command. If we keep doing that, we'll see that every once in a while, a node is drained. Then, a while later, it's being restored to normal (un-cordoned).

We can see that the measures and the improvements to the system that we did so far were successful. The cluster and the applications in it are more robust then they were when we started with the first experiment. But that's not entirely true. I was kind of lucky, and you might not have been. The database is still running as a single replica, and that is the only weak point that we still have. Because of that, you might have seen a failure that I did not experience myself.

Monitoring And Alerting With Prometheus

As I already mentioned, the critical ingredient that Chaos Toolkit does not provide is notifications whether a part of the system failed. Steady-state hypotheses are focused on what we know, and they are usually limited to a single application, network, storage, or node. By their nature, they are limited in their scope.

As you already know, we do need a proper monitoring system. We need to gather the metrics, and we are already doing that through Prometheus. We need to be able to observe those metrics to deduce the state of everything in our cluster. We can do that through dashboards. But that often proves to be insufficient beyond the initial stages. We need a proper alerting system that we will improve over time. Whenever we detect something through dashboards, we probably want to convert that discovery into an alert. If we do that, we will not need to keep staring at the monitor filled with "pretty colors" forever.

I prefer to use AlertManager[87]. You might not have the same taste and choose to use something else. It could be, for example, DataDog[88], or anything else that suits your needs. Nevertheless, this is a book about chaos engineering. Even though monitoring is an essential part of it, it is still not in the scope. So, I will not go into monitoring in more detail because that would require much than a few paragraphs. A whole chapter would not be able to even scratch the surface. If you do feel you need more, you might want to check The DevOps 2.5 Toolkit: Monitoring, Logging, and Auto-Scaling Kubernetes[89].

You should be comfortable with monitoring and other related subjects. Without robust monitoring and alerting, there is no proper chaos engineering. Or, to be more precise, I don't think that you

[87] https://prometheus.io/docs/alerting/alertmanager
[88] https://www.datadoghq.com/
[89] https://www.devopstoolkitseries.com/posts/devops-25/

should jump into chaos engineering without first mastering monitoring and alerting. So, figure out how to properly monitor, observe, and alert based on some thresholds in your system. Only after that, you'll be able to do chaos experiments successfully.

Destroying What We Created

That's it. Yet another chapter is finished, and we are going to destroy the things we created.

This time, the instructions will be slightly different.

Go to the second terminal and stop sending requests to the demo application by pressing *ctrl+c*.

Next, please go back to the first terminal.

This time we'll need to delete two Namespaces.

```
cd ..

kubectl delete namespace go-demo-8

kubectl delete namespace chaos
```

The only thing left is to destroy the cluster itself, unless you choose to leave it running. If you do want to destroy it, you will find the instructions at the end of the Gists you used to create it.

Until The Next Time

That's it, the book is finished.

I might extend it over time. But, at this moment, if you went through all the exercises and you did the homework, you hopefully learned what chaos engineering is. You saw some of the benefits, the upsides and the downsides, and the traps behind it. I hope that you found it useful.

Now, let's go one more time through our checklist.

At the very beginning of the course, we defined a list of the things we'll try to accomplish. So let's see whether we fulfilled those. As a refresher, the list of the tasks we defined at the very beginning as follows.

- Terminate instance of an app
- Partially terminate network
- Increase latency
- Simulate Denial of Service (DoS) attacks
- Drain a node
- Delete a node
- Create reports
- Send notifications
- Run the experiments inside a Kubernetes cluster

The first item was to terminate an instance of an app. We went beyond that. We were terminating not only instances of an application, but also of its dependencies. We even created experiments that terminate random instances of random apps.

The next item was to partially terminate a network. We used Istio VirtualServices to define network routes. It was the right choice since Istio provides the means to manipulate the behavior of networking that can be leveraged by chaos experiments. As a result, we run experiments that terminate some of the network requests. That gave us an insight into the problems networking might cause to our applications. We did not stop at discovering some of the network-related issues. We also fixed the few problems we uncovered.

We also decided that we'll explore what happens when we increase latency. Since, by that time, we were already committed to Istio, we used it for the experiments related to latency as well. We saw that the demo application was not prepared to deal with delayed responses, so we modified the definition of a few resources. As a result, the demo application became tolerant to increased latency.

We also simulated a Denial of Service (DoS) attacks. Just as with other network-related experiments, we used Istio for that as well.

Then we moved into nodes. We saw how we can drain them, and we observed the adverse effects that might cause.

As if draining wasn't enough, we also deleted a node. That, like most other experiments, provided some lessons that helped us improve our cluster and the applications running in it.

We tried quite a few other things, which I'm probably forgetting mention.

Once we got comfortable with the scope limited to the demo application, we explored how to run experiments on the Namespace and the cluster level.

Since destruction is not the goal in itself, we dived into the generation of reports and notifications. We used Slack, and you should be able to extend that knowledge to send notifications somewhere as well.

We learned how to run the experiments inside a Kubernetes. We defined Jobs, for one-shot experiments, that could be hooked into our continuous delivery pipelines. We saw how to create CronJobs that can run our experiments periodically.

It's just as important to mention things that we did not do. We didn't go beyond Kubernetes. I tried to avoid that because there are too many permutations that would need to be taken into account. You might be running your cluster in AWS, GCP, Azure, VMware on-prem, or somewhere else. Each infrastructure vendor is different, and each could fit into a book of its own. So we stayed away from experiments for specific hosting and infrastructure vendors. On the bright side, you probably saw in Chaos Toolkit documentation that there are infrastructure-specific plugins that we can use. If that's not enough, you can always go beyond what plugins do by running processes (commands, scripts). Even better, you can contribute to the project by extending existing plugins or creating new ones.

What you should not do, after all this, is assume that the examples that we explored should be used as they are. They shouldn't. The exercises were aimed at teaching you how to think and how to go beyond the obvious cases. Ultimately, you should define experiments in your own way, and adapt the lessons learned to your specific needs. Your system is not the same as mine or anyone else's. While we all do some of the things the same, many are particular to an organization. So, please don't run the experiments as they are. Tailor them to your own needs.

You should always start small. Don't do everything you can think of right away. Start with basics, gain confidence in what you're doing, and then increase the scope. Don't go crazy from day one. Don't use CronJobs right away. Run the experiments as one-shot executions first. Make sure that they work, and then create CronJobs that will do the same things repeatedly. Confidence is the key. Make sure that everybody in your organization is aware of what you do. Make sure that your colleagues understand the outcomes of those experiments. The goal of chaos experiments is not for you to have fun. They are meant to help learn something and to improve our systems. To improve the system, you need to propagate the findings of the experiments throughout the whole organization.

That's about it. Thank you so much. I hope you found the book useful.

Reach out to me if you need anything. Contact me on Slack, on Twitter, on email, or sent courier pigeons. It does not matter which method you use. I'll do my best to make myself available for any questions, doubts, or issues. I will do my best to help you out.

And with that, I bid you farewell. It was exhilarating for us (Darin and me) to do this book. See you on a different course or a book. You might see me at a conference or a workshop. Most of the things I do are public. My job is to help people improve, and I hope that I accomplished that with this book.

Shameless Plug

Here's the list of some of the other work we do, as well as my contact details.

- DevOps Paradox Podcast[90]
- The DevOps Toolkit Series[91]
- DevOps Paradox[92]
- The DevOps Toolkit Series On YouTube[93]
- TechnologyConversations.com[94]
- Twitter[95]
- DevOps20 Slack Workspace[96]

[90] https://www.devopsparadox.com/
[91] https://www.devopstoolkitseries.com/
[92] https://amzn.to/2myrYYA
[93] https://www.youtube.com/channel/UCfz8x0lVzJpb_dgWm9kPVrw/
[94] https://technologyconversations.com
[95] https://twitter.com/vfarcic
[96] http://slack.devops20toolkit.com