**Chapter 9. Interaction redundancy: Retries and other control loops**

*This chapter covers*

* Retries: repeating access attempts on timeouts
* Retry storms
* Safe and idempotent services
* Fallbacks
* Control loops

While surfing the web, what do you do when a web page you’re trying to access fails to load? You hit the refresh button, right? I’ve talked a lot about redundant service instances, but now want to turn to another place where redundancy is used in cloud-native software: when making requests. Just as depending on a single instance of an app to always be up is untenable, so too is depending on each and every request to never experience any trouble. Instead, your software will repeat requests, just as you do. Well, maybe not *just as*. Let’s explore this a bit.

The case that I started with is the simplest: you’re loading a page to read it. For example, you might be looking at the Hacker News homepage (<https://news.ycombinator.com/>), the headline “Monks Who Play Punk (2007)” catches your fancy, and you click the link to read the full article. The article doesn’t load, or it only partially loads, so you hit the refresh button and all is fine.

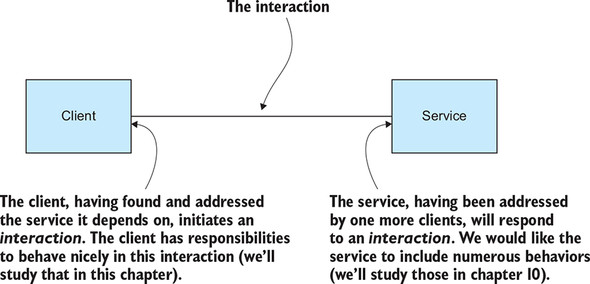
But if you experience a failure to load right after clicking the Place Your Order button on your favorite e-commerce site, you’re unlikely to just click that button again. You’ll first check to see whether the purchase went through, perhaps by checking the shopping cart, accessing your list of open orders, or maybe even checking to see whether you got an order confirmation email. If you find evidence that the order was placed, you’re good to go—no need to repeat the partially failed request. If, on the other hand, you’re reasonably confident that the order wasn’t placed, you’ll go back and repeat the request for purchase.

Let me also draw your attention to another feature you’ve likely run across in some of the websites you visit: the I Am Not a Robot check box or captcha. This type of a widget is generally used to put a governor on certain website interactions, to keep bots from creating (large numbers of) accounts, or to keep them from hacking passwords, for example. This feature is fundamentally about retries and highlights another aspect of request redundancy that I’ll cover here: when you move from human users to machine clients, you need to be cognizant of the orders-of-magnitude increases in both volume and frequency of requests.

These familiar scenarios are a good place to begin our exploration of redundant interactions, yet it’s important to understand that both the actors and the context are different. In our cloud-native software architectures, the client and the service of the interaction are programs. The software-based client has to make decisions similar to what you do as a human—how long should it wait before giving up on a request, for example. It also needs to understand when retries shouldn’t be made, and it needs to be aware of its own power.

You might have noticed that I’m using the word “interaction,” and hopefully this takes you back to the mental model for cloud-native software that I established in the first chapter. The interaction is one of the primary entities I introduced there. And although [chapter 8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08) approached the topic, it was primarily about what is needed before an interaction is established—how the client finds a dependent service, for example. Now I begin to address the interaction in earnest (see [figure 9.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig01)), first with this chapter’s focus on the client side of that interaction. [Chapter 10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10) covers the service end of the interaction.

**Figure 9.1. Certain design patterns applied on both sides of an interaction will yield far more robust and reliable systems. We’ll cover client-side patterns in this chapter, and service-side patterns in the next.**



I start the chapter by adding a naïve retry implementation to our sample application, and we’ll run some experiments to demonstrate the value of this pattern. Then I take those experiments to an extreme, so you can see what happens when large volumes of retries have negative consequences that ripple through the entire system. We’ll then explore techniques to guard against these retry storms. Finally, request retries are but one example of repeat actions, and I close the chapter by talking about the general pattern of control loops and their valuable role in cloud-native software.

**9.1. Request retries**

Cloud-native software is, almost by definition, a distributed system. In the past, invocation of a functionality from another part of your code was just a method call, and everything was running within the same process. Today, your implementations are filled with requests that go over the network—a network that isn’t always reliable. And even when the network is fine, there are no guarantees that while your process is up and running, the service you’re calling is equally healthy. It’s these attributes of distributed systems that drive the need for addressing request resilience.

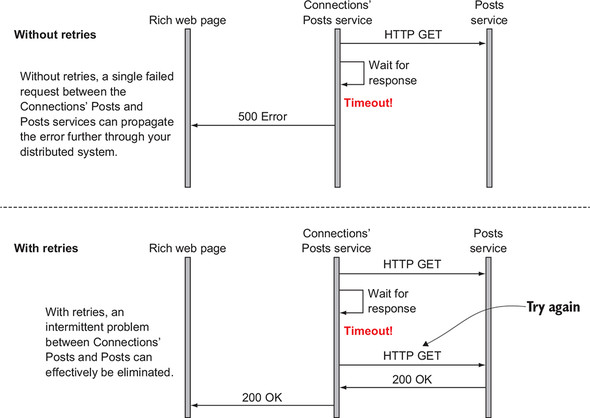
Let me clarify one thing right at the onset. There are a number of ways we might define or realize *request resilience*. A more traditional approach might focus on *request durability*—coming up with ways to ensure that requests are never lost. But this is analogous to the traditional approach of hardening servers and storage devices: make them stronger and stronger so they don’t fail. Instead, the more modern paradigm, which is the undercurrent of everything in this book, accepts that components *will* fail and that we achieve resilience by adapting to that inevitable disruption. That’s what this chapter does with requests. It explores achieving resilience with a redundancy of requests instead of treating each request as something that simply can’t be lost. All of that said, upcoming chapters present approaches that store requests, but with a twist; but I’ll leave all of that for later.

**9.1.1. The basic request retry**

The basic pattern is simple: your app is going to make a request to a remote service, and if it doesn’t hear back within a reasonable time, it’s going to try again. By now you’re familiar with our running example—the blog aggregator—in which the Connections’ Posts service makes calls to both the Connections service and the Posts service and then returns an aggregated result.

The demonstration that you’ll be looking at in a moment focuses on the Connections’ Posts service as a client that makes an HTTP request to the Posts service (it’s still making requests to the Connections service, but for the purposes of this exercise, you’ll focus on the interaction between Connections’ Posts and Posts). In this example, you’ve implemented a retry around this request ([figure 9.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig02)), so that now, when the Connections’ Posts service makes a call to the Posts service and doesn’t receive a response, it will simply try the request again.

**Figure 9.2. Retries can insulate parts of your distributed system from errors in other parts.**



This simple retry will make your overall system more tolerant to failures, returning results in cases where, otherwise, the Connections’ Posts service might have failed to produce the aggregated set of blog posts.

**9.1.2. Let’s see this in action: Simple retries**

In this chapter and the next, I’m going to have you run a series of experiments that explore the impact of applying various cloud-native patterns to the interaction between services. This first example lays the foundation, with each subsequent implementation building on the last. You’ll start by implementing a simple retry.

**Setting up**

At this point, I refer you to the setup instructions for running the samples in earlier chapters. There are no new requirements for running the sample in this chapter.

You’ll be accessing files in the cloudnative-requestresilience directory, so in your terminal window change into that directory.

As I’ve described in previous chapters, I’ve already prebuilt Docker images and made them available in Docker Hub. If you want to build the Java source and Docker images and push them to your own image repository, I refer you to earlier chapters (the most detailed instructions are in [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05)).

**Running the apps**

As you progress through the chapter, you’ll use different versions of the retry pattern, so to start, you’ll need to check out the right tag on the GitHub repo:

git checkout requestretries/0.0.1

You’ll need a Kubernetes cluster, and for this initial example you may use Minikube. See [section 5.2.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05lev2sec2) in [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05) for instructions on how get Minikube up and running. To start with a clean slate, delete any deployments that might be left over from your previous work. I’ve provided you a script to do that: deleteDeploymentComplete.sh. This simple bash script allows you to keep the MySQL, Redis, and Spring Cloud services running. Calling it with no options deletes only the three microservice deployments; calling the script with all as an argument deletes MySQL, Redis, and SCSS services as well. Verify that your environment is clean with the following command:

$ kubectl get all

NAME READY STATUS RESTARTS AGE

pod/mysql-6585c56bff-hfwn5 1/1 Running 0 2m

pod/redis-846b8c56fb-wr6zx 1/1 Running 0 2m

pod/sccs-84cc988f57-d2mgm 1/1 Running 0 2m

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/connectionsposts-svc 10.101.76.173 <none> 80:31224/TCP 44s

service/connections-svc 10.105.144.139 <none> 80:32290/TCP 44s

service/kubernetes 10.96.0.1 <none> 443/TCP 4m

service/mysql-svc 10.109.9.155 <none> 3306:32260/TCP 2m

service/posts-svc 10.98.202.179 <none> 80:32746/TCP 45s

service/redis-svc 10.109.19.150 <none> 6379:30270/TCP 2m

service/sccs-svc 10.98.94.67 <none> 8888:32640/TCP 2m

NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE

deployment.apps/mysql 1 1 1 1 2m

deployment.apps/redis 1 1 1 1 2m

deployment.apps/sccs 1 1 1 1 2m

NAME DESIRED CURRENT READY AGE

replicaset.apps/mysql-6585c56bff 1 1 1 2m

replicaset.apps/redis-846b8c56fb 1 1 1 2m

replicaset.apps/sccs-84cc988f57 1 1 1 2m

Note that you’ve left mysql, redis, and sccs running, as well as the services for your three microservices. If you’ve cleared out redis, mysql, and sccs, deploy each by running the deployServices.sh bash script. If you’ve created the MySQL service anew, don’t forget to create the cookbook database with the following commands:

$ mysql -h $(minikube service mysql-svc --format "{{.IP}}") \

-P $(minikube service mysql-svc --format "{{.Port}}") -u root -p

mysql> create database cookbook;

You can now deploy the three microservices by running three kubectl apply commands pointing to the Connections’ Posts, Connections, and Posts YAML files. I’ve created a script that encapsulates all three, so you can simply run this:

./deployApps.sh

As you’ve done in the past, invoke the Connections’ Posts microservice, by first logging in and then accessing the list of posts for your connections:

curl -i -X POST -c cookie \

$(minikube service --url connectionsposts-svc)/login?username=cdavisafc

curl -i -b cookie \

$(minikube service --url connectionsposts-svc)/connectionsposts

At this point, you should be able to execute that last command repeatedly with consistent results.

Now let’s cause some trouble. Recall that in an earlier chapter you added an endpoint to the Posts service to break it. By issuing an HTTP post against the /infect endpoint, responses to subsequent requests to the service will be delayed by 400 seconds. That’s pretty broken. Notice that in the deployment manifest, I’ve removed the liveness probe that I added at the end of [chapter 8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08); for our experiments here, I want to keep these services broken. You currently have two instances of the Posts service running, so let’s break one of them by making that POST request:

curl -i -X POST $(minikube service --url posts-svc)/infect

Before invoking the Connections’ Posts service again, let’s start streaming the logs for that service by running the following command in another terminal window:

kubectl logs –f <name of your Connections' Posts pod>

Now access the Connections’ Posts service a few more times. I’d like you to notice two things. First, on each curl, you receive a response—the aggregation service is working just fine. But second, looking at the logs, you can see entries such as this one:

... : [172.17.0.10:8080] getting posts for user network cdavisafc

... : [172.17.0.10:8080] connections = 2,3

... : [172.17.0.10:8080] On (0) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Read timed out;

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg nested exception is java.net.SocketTimeoutException: **Read timed out**

... : [172.17.0.10:8080] On (1) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Read timed out;

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg nested exception is java.net.SocketTimeoutException: **Read timed out**

... : [172.17.0.10:8080] On (2) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Read timed out;

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg nested exception is java.net.SocketTimeoutException: **Read timed out**

... : [172.17.0.10:8080] Retrieved results from database

This shows that requests were made to the Posts service that timed out, but instead of that failure propagating all the way back to you, the client, the Connections’ Posts service automatically recovered by retrying the request. Even if it took a few retries (in the preceding example, it took three retries), eventually the non-infected Posts service was reached, and the result was returned.

Let’s look at the implementation from the ConnectionsPostsController.java file.

**Listing 9.1. Excerpt from ConnectionsPostsController.java**

int retryCount = 0;

**while (implementRetries** || retryCount == 0) {

try {

RestTemplate restTemp = restTemplateBuilder

.setConnectTimeout(connectTimeout)

.setReadTimeout(readTimeout)

.build();

ResponseEntity<PostResult[]> respPosts

= restTemp.getForEntity(postsUrl + ids + secretQueryParam,

PostResult[].class);

if (respPosts.getStatusCode().is5xxServerError()) {

response.setStatus(500);

return null;

} else {

logger.info(utils.ipTag() + "Retrieved results from database");

PostResult[] posts = respPosts.getBody();

for (int i = 0; i < posts.length; i++)

postSummaries.add(

new PostSummary(getUsersname(posts[i].getUserId()),

posts[i].getTitle(), posts[i].getDate()));

return postSummaries;

}

**} catch (Exception e) {**

**// Will occur when a connection times out.**

**// For this naive implementation, we will simply**

**// try again.**

**logger.info(utils.ipTag() +**

**"On (" + retryCount + ") request to unhealthy posts service " +**

**e.getMessage());**

**if (implementRetries)**

**retryCount++;**

else {

logger.info(utils.ipTag() +

"Not implementing retries - returning with a 500");

response.setStatus(500);

return null;

}

}

}

As you can see, the implementation is simple. If you’re implementing retries, which are controlled through a new application property, you’ll make a request to the Posts service. If it times out, you stay in the while loop and try again. You can see the lines that generate the very log messages you looked at when running the sample.

Although this is indeed simple, here you’re already getting to the first nuance: how long will your implementation wait before a timeout exception is thrown? (Ultimately, this is something that the application operator is likely to decide—and that could be you.) Too long, and your upstream clients (the web page invoking the Connections’ Posts service) may be left waiting for an extended period of time and may themselves time out. Too short, and Connections’ Posts may be forfeiting perfectly valid results (and causing some of the downstream ramifications covered in the next section). In the current implementation, I’ve set the connect timeout to ¼ of a second and the read timeout to ½ of a second, as you can see in these lines:

RestTemplate restTemplate = restTemplateBuilder

.setConnectTimeout(250)

.setReadTimeout(500)

.build();

I want to draw your attention to something: in everything you’ve done here, both in the case where a human refreshed a web page and in our implementation, never did you concern yourself with *why* you didn’t get a response from the downstream service. It could have been a network problem, or a bug in the application (arguably what I’ve demonstrated here), or any number of other problems. When the problem is intermittent, the reason is most often immaterial. Only when you start to see persistent problems will you care, and even then, it’s not a concern of your apps, but rather a general monitoring concern. We’ll cover troubleshooting in a later chapter.

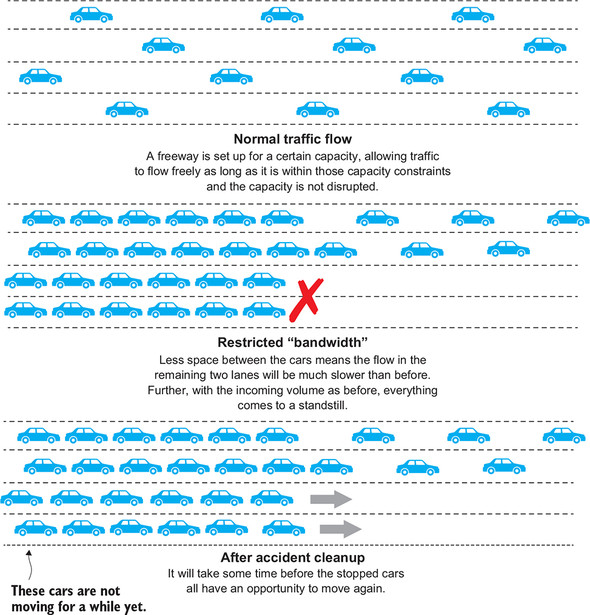
Okay, this looks pretty good; it’s easy and it seems to work. What of these downstream ramifications and persistent problems I keep alluding to? Let’s jump in.

**9.1.3. Retries: what could go wrong?**

In the preceding example, the software operates just fine with the limited load that you’re placing on the system with your command-line curls. But it can be a different story when something goes wrong in a system that is otherwise well tuned for a particular load. It’s a bit like highway traffic.

Consider a freeway that has enough lanes to allow 14,000 cars traveling at 60 mph to pass through it in an hour (by my back-of-the-envelope calculations, that’s a four-lane highway). As long as no accidents occur, all is well. But when two lanes are made impassible by a fender-bender, things change quickly. Not only will the same volume of traffic now traveling in half the lanes have to slow down to maintain safe driving distance, but the cars approaching the constrained stretch of highway, at the same volume as before, will fairly quickly generate quite a traffic jam. And, as we’ve all likely experienced, even when the accident site is cleared, it takes some time before all the queued-up traffic is flowing again. This scenario is depicted in [figure 9.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig03).

**Figure 9.3. Restricted networks act like restricted highways, backing up requests that keep coming in the same volumes as before. Even when the restrictions are lifted, it can take some time before all of the queued-up traffic is moving again.**



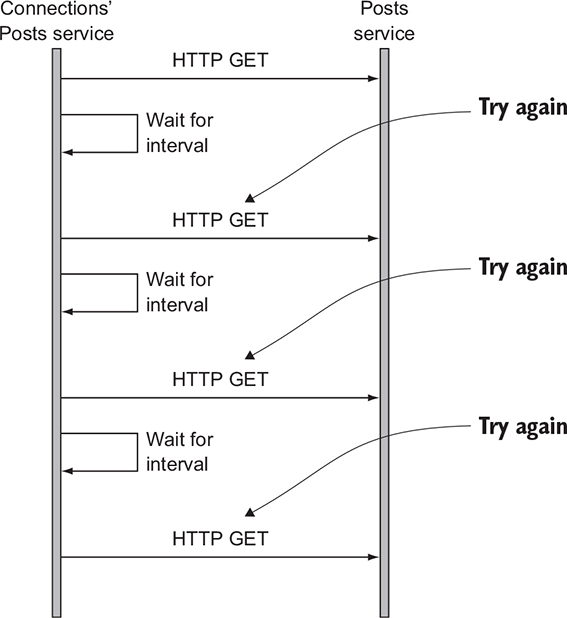
The situation is exactly the same with the request flow through your network of app instances. Even though your site reliability engineer has no doubt designed a deployment topology that leaves a bit of room for fluctuations in request volume and minor hiccups, when a sizeable portion of your load gets backed up, the effects can ripple through the system. Let’s see this in action.

**9.1.4. Creating a retry storm**

In the previous section, I presented a basic pattern: the Connections’ Posts service will retry calls to the Posts service in the event that it doesn’t initially receive a response. At a high level, this makes complete sense, and the current implementation will handle minor glitches just fine. But when something more significant occurs (the metaphorical highway accident), retries may not be as helpful and may even hurt the overall health of the system. What I want to do here is experiment with the naïve implementation that I presented in the previous section.

By way of reminder, you’re focusing on the interaction between the Connections’ Posts service (the client) and the Posts service. As you saw in the first example, you’ve implemented a retry around this request. [Figure 9.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig04) shows repeated retries, just as you saw previously and will see even more of in this coming example. Let’s get you set up so you can follow along.

**Figure 9.4. The client, the Connections’ Posts service, will retry on connection or read timeouts to the Posts service. It will continue trying until it receives a success status code from the HTTP request to the Posts service.**



**9.1.5. Let’s see this in action: Creating a retry storm**

This experiment makes no changes to the code of the previous one, but you will send a sizeable load of traffic to the system, simulate a short disruption, and observe the results (spoiler alert: they’re rather unpleasant).

**Setting up**

You’ll need everything that I listed in the first example of this chapter with one adjustment:

* Access to a larger Kubernetes cluster.
* That Kubernetes cluster must allow privileged containers to be run.

You’ll be placing your software under load and then introducing failures into the system. I want to have you explore, for example, what happens when you lose all lanes on our metaphorical highway and, just as important, what happens when those lanes are restored. To be able to place a significant load on the system, you’ll create a larger deployment of our sample app; hence, you’ll need a larger environment to run this in.

Without getting into the details of a fairly complex topic, I’ll say this: privileged containers allow for more commands to be executed within them than nonprivileged containers do, and you’ll need this when you begin to restrict network traffic. The good news is that most of the cloud providers will serve Kubernetes clusters with privileged containers enabled by default.

At the time of writing, I find that Google Kubernetes Engine (GKE) provides the easiest public cloud experience in creating Kubernetes clusters. You’ll need a cluster with approximately 25–30 GB of memory across all of the nodes. GKE also enables privileged containers by default, something you’ll need.

To run the simulations in this section, please check out the following tag from your Git repo:

git checkout requestretries/0.0.2

**Running the apps**

There are three parts to executing the experiments:

1. Deploying the application
2. Placing load against the application
3. Simulating various failure scenarios and observing the results

Unless you’ve already been running the examples on a larger Kubernetes cluster, or one that you can resize to have sufficient capacity, you’ll likely have to deploy all the components that make up our sample anew. I won’t go over the installation in detail here, instead referring you to earlier chapters, but in summary, after creating that new Kubernetes cluster and connecting to it with kubectl, you need to do the following:

1. Edit the deployment manifest for the Spring Cloud Configuration Server (SCCS), spring-cloud-config-server-deployment-kubernetes.yaml, to point to the Git repo that houses your app configs. You may, of course, keep it pointing at my repo.
2. Deploy MySQL, Redis, and SCCS. I’ve provided a script, so you can simply run the deployServices.sh bash script.
3. Create the cookbook database by connecting to MySQL with a command-line client and executing the command create database cookbook;. Notice that the MySQL deployment manifest specifies LoadBalancer for the service type that should have allocated a public IP address for your MySQL database. You can use that to connect with your mysql CLI.
4. Deploy all three microservices by executing the deployApps.sh bash script.

This should result in a deployment that looks something like this:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

connection-posts-685c669f7b-4qvx7 1/1 Running 0 6d

connection-posts-685c669f7b-6lgmf 1/1 Running 0 6d

connection-posts-685c669f7b-6pt9p 1/1 Running 0 6d

connection-posts-685c669f7b-d8q8h 1/1 Running 0 6d

connection-posts-685c669f7b-z7gsw 1/1 Running 0 6d

connections-7cf9b5ccf9-cjnhs 1/1 Running 0 6d

connections-7cf9b5ccf9-cw4s9 1/1 Running 0 6d

connections-7cf9b5ccf9-kskqm 1/1 Running 0 6d

connections-7cf9b5ccf9-mfj8b 1/1 Running 0 6d

connections-7cf9b5ccf9-nd4nw 1/1 Running 0 6d

connections-7cf9b5ccf9-nnl8r 1/1 Running 0 6d

connections-7cf9b5ccf9-xjq8j 1/1 Running 0 6d

mysql-64bd6d89d8-96vb6 1/1 Running 0 27d

posts-7785bcf45-9tfj4 1/1 Running 0 6d

posts-7785bcf45-bsn8g 1/1 Running 0 6d

posts-7785bcf45-w5xzs 1/1 Running 0 6d

posts-7785bcf45-wtbv8 1/1 Running 0 6d

redis-846b8c56fb-bm5z9 1/1 Running 0 27d

sccs-84cc988f57-hp2z2 1/1 Running 0 27d

To place load against the application, you’ll be using Apache JMeter. I’ve created both a Kubernetes deployment of JMeter, as well as the config file containing the specifications for our load test. The first step in getting this running is to upload the config file, which you’ll do with the creation of a Kubernetes config map. Execute the following command:

kubectl create configmap jmeter-config \

--from-file=jmeter\_run.jmx=loadTesting/ConnectionsPostsLoad.jmx

When you want to run the load tests, you can now simply create the JMeter deployment; to stop the load test, you’ll delete the deployment. Let’s try that now. Execute the following command:

kubectl create -f loadTesting/jmeter-deployment.yaml

To see the JMeter output, stream the logs for the JMeter pod with a command such as the following (inserting the name of your JMeter pod):

kubectl logs -f <name of your jmeter pod>

You’ll see log output such as the following:

$ kubectl logs -f jmeter-deployment-7d747c985-kjxct

START Running Jmeter on Mon Feb 18 19:42:17 UTC 2019

JVM\_ARGS=-Xmn506m -Xms2024m -Xmx2024m

jmeter args=-n -t /etc/jmeter/jmeter\_run.jmx

Feb 18, 2019 7:42:19 PM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

Creating summariser <summary>

Created the tree successfully using /etc/jmeter/jmeter\_run.jmx

Starting the test @ Mon Feb 18 19:42:19 UTC 2019 (1550518939413)

Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445

summary + 530 in 00:00:30 = 17.7/s Err: 0 (0.00%) Active: 328

summary = 612 in 00:00:40 = 15.3/s Err: 0 (0.00%)

summary + 1027 in 00:00:30 = 34.3/s Err: 0 (0.00%) Active: 576

summary = 1639 in 00:01:10 = 23.4/s Err: 0 (0.00%)

summary + 1521 in 00:00:30 = 50.7/s Err: 0 (0.00%) Active: 823

summary = 3160 in 00:01:40 = 31.6/s Err: 0 (0.00%)

summary + 2014 in 00:00:30 = 66.6/s Err: 0 (0.00%) Active: 1073

summary = 5174 in 00:02:10 = 39.7/s Err: 0 (0.00%)

summary + 2512 in 00:00:30 = 84.4/s Err: 0 (0.00%) Active: 1319

summary = 7686 in 00:02:40 = 48.0/s Err: 0 (0.00%)

summary + 2939 in 00:00:30 = 98.0/s Err: 0 (0.00%) Active: 1500

summary = 10625 in 00:03:10 = 55.9/s Err: 0 (0.00%)

This shows that the Connections’ Posts app was serving close to 100 requests per second after the load reached full capacity (I’ve set up the tests to ramp slowly) with 0.0% error (you’ll be watching that error number as you progress). To stop the load test, execute the following command:

kubectl delete deploy jmeter-deployment

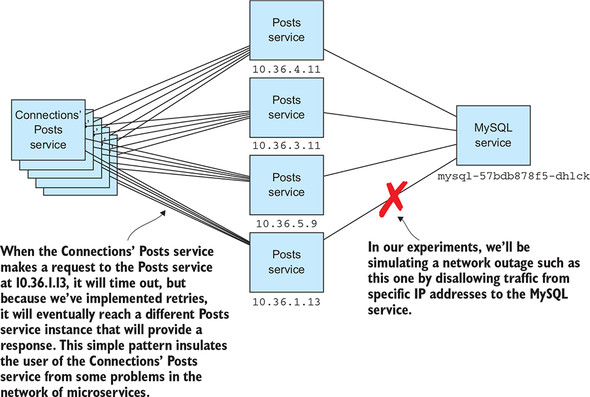
Now that you’ve established your deployment and verified that load testing is functioning properly, let’s begin our experiments.

You’ll be simulating network outages between the Posts service and the MySQL service. In the real world, such network outages can be caused by hardware failures (the loss of a physical switch, for example) or by a configuration error (the incorrect modification of a firewall rule, for example). To create the same effect here, you’ll be changing the routing rules in the MySQL service to either disallow or, later when you fix the network, allow for requests from specific instances of the Posts service.

[Figure 9.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig05) shows the five instances of the Connections’ Posts service, the four instances of the Posts service, and the single instance of the MySQL service that you’ve deployed. The lines between each of the instances represent the different ways that these services can communicate. Notice that each Posts service is annotated with an IP address and that the MySQL service is annotated with a pod name. To disallow traffic to flow on one of those connections, you’ll create a routing rule in the MySQL instance that rejects traffic from a specific IP address. To do this, you’ll run a route command in the MySQL container; and this is done with a kubectl exec command. To break the network connection between the Posts service running at IP address 10.36.1.13 and the Posts instance running in the pod named mysql-57bdb878f5-dhlck, you’ll execute the following command:

kubectl exec mysql-57bdb878f5-dhlck -- route add -host 10.36.1.13 reject

**Figure 9.5. A deployment containing five instances of the Connections’ Posts service and four of the Posts service results in 20 ways that an instance of the former may connect to an instance of the latter. There are four connections between instances of the Posts service and the single instance of the MySQL service. Retries are an effective way of finding a healthy path through the network of microservices.**



This is indicated in [figure 9.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig05) with the X through the connectivity line. When the Posts service running at IP address 10.36.1.13 tries to connect to the MySQL service, it will time out. That timeout will propagate up to the Connections’ Posts service, and the result will be a retry. If you’re lucky, that retry will reach a different Posts instance and will be able to access the database, and the Connections’ Posts request will succeed.

Reestablishing the connection is done with a similar kubectl exec command that removes the reject-routing rule from the container running the MySQL service:

kubectl exec mysql-57bdb878f5-dhlck -- route delete -host 10.36.1.13 reject

With those mechanics established, let’s now run two experiments:

1. Completely disconnect, via the route commands shown previously, all instances of Posts from the MySQL service, with the *retry logic enabled*. This is done by setting the CONNECTIONPOSTCONTROLLER\_IMPLEMENTRETRIES env variable in the deployment manifest for the Connections’ Posts app to true.
2. Completely disconnect, via the route commands shown previously, all instances of Posts from the MySQL service, with the *retry logic disabled*. This is done by setting the CONNECTIONPOSTCONTROLLER\_IMPLEMENTRETRIES env variable in the deployment manifest for the Connections’ Posts app to false. When the attempted connections to the Posts service time out, the Connections’ Posts service will return with an error status and no result.

To save you the trouble of executing the preceding kubectl exec command four times by hand, I’ve provided you a script, alternetwork-db.sh. You’ll need to edit that script, however, to reflect your MySQL pod name and the IP addresses of your Posts instances. You can get the name of your MySQL service with the usual kubectl command:

kubectl get pods

And to get the IP addresses for the Posts instances, use this:

kubectl get pods -l app=posts -o wide

Now you can deny connections from all Posts instances to the MySQL instance by running this command:

./alternetwork-db.sh add

The following is log output from one of the instances of the Connections’ Posts service, showing that it went from serving traffic to only performing retries:

2019-02-18 04:05:55.986 ... connections = 2,3

2019-02-18 04:05:55.989 ... getting posts for user network cdavisafc

2019-02-18 04:05:55.995 ... connections = 2,3

2019-02-18 04:05:56.055 ... getting posts for user network cdavisafc

2019-02-18 04:05:56.056 ... getting posts for user network cdavisafc

2019-02-18 04:05:56.059 ... On (0) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Connect to posts-

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg exception is org.apache.http.conn.ConnectTimeoutException: Connect to

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out

2019-02-18 04:05:56.060 ... connections = 2,3

2019-02-18 04:05:56.060 ... connections = 2,3

2019-02-18 04:05:56.070 ... getting posts for user network cdavisafc

2019-02-18 04:05:56.074 ... connections = 2,3

2019-02-18 04:05:56.092 ... On (1) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Connect to posts-

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg exception is org.apache.http.conn.ConnectTimeoutException: Connect to

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out

2019-02-18 04:05:56.093 ... On (2) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Connect to posts-

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg exception is org.apache.http.conn.ConnectTimeoutException: Connect to

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out

2019-02-18 04:05:56.229 ... On (0) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Connect to posts-

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg exception is org.apache.http.conn.ConnectTimeoutException: Connect to

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out

2019-02-18 04:05:56.232 ... On (0) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Connect to posts-

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg exception is org.apache.http.conn.ConnectTimeoutException: Connect to

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out

2019-02-18 04:05:56.310 ... On (0) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Connect to posts-

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg exception is org.apache.http.conn.ConnectTimeoutException: Connect to

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out

2019-02-18 04:05:56.343 ... On (6) request to unhealthy posts service I/O

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg error on GET request for "http://posts-svc/posts": Connect to posts-

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg svc:80 [posts-svc/10.19.252.1] failed: connect timed out; nested

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg exception is org.apache.http.conn.ConnectTimeoutException: Connect to

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg posts-svc:80 [posts-svc/10.19.252.1] failed: connect timed out

The following shows the JMeter output from this experiment, annotated with three points in time. When the trial begins, the Connections’ Posts app is returning results with 0.0% error. Then at time marker 1, when you run the ./alternetwork-db.sh add command, you can see that quickly, the error rate goes to 100%. The Connections’ Posts service never returns, JMeter times out on the request (and counts the attempt as an error), yet the Connections’ Posts apps continue retrying the Posts service indefinitely.

START Running Jmeter on Mon Feb 18 20:08:18 UTC 2019

JVM\_ARGS=-Xmn402m -Xms1608m -Xmx1608m

jmeter args=-n -t /etc/jmeter/jmeter\_run.jmx

Feb 18, 2019 8:08:20 PM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

Creating summariser <summary>

Created the tree successfully using /etc/jmeter/jmeter\_run.jmx

Starting the test @ Mon Feb 18 20:08:21 UTC 2019 (1550520501121)

Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445

summary + 67 in 00:00:08 = 8.2/s Err: 0 (0.00%) Active: 67

summary + 501 in 00:00:30 = 16.7/s Err: 0 (0.00%) Active: 314

summary = 568 in 00:00:38 = 14.9/s Err: 0 (0.00%)

summary + 999 in 00:00:30 = 33.3/s Err: 0 (0.00%) Active: 562

summary = 1567 in 00:01:08 = 23.0/s Err: 0 (0.00%)

summary + 1493 in 00:00:30 = 49.8/s Err: 0 (0.00%) Active: 810

summary = 3060 in 00:01:38 = 31.2/s Err: 0 (0.00%)

summary + 1992 in 00:00:30 = 66.4/s Err: 0 (0.00%) Active: 1059

summary = 5052 in 00:02:08 = 39.4/s Err: 0 (0.00%)

summary + 2488 in 00:00:30 = 82.9/s Err: 0 (0.00%) Active: 1307

summary = 7540 in 00:02:38 = 47.7/s Err: 0 (0.00%)

summary + 2929 in 00:00:30 = 97.7/s Err: 0 (0.00%) Active: 1500

summary = 10469 in 00:03:08 = 55.7/s Err: 0 (0.00%)

summary + 2997 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 13466 in 00:03:38 = 61.7/s Err: 0 (0.00%)

<time marker 1 – I have broken the network between Posts and MySQL>

summary + 2515 in 00:00:30 = 83.8/s Err: 2239 (89.03%) Active: 1500

summary = 15981 in 00:04:08 = 64.4/s Err: 2239 (14.01%)

summary + 3000 in 00:00:30 = 100.0/s Err: 3000 (100.00%) Active: 1500

summary = 18981 in 00:04:38 = 68.2/s Err: 5239 (27.60%)

summary + 2961 in 00:00:30 = 98.7/s Err: 2961 (100.00%) Active: 1500

summary = 21942 in 00:05:08 = 71.2/s Err: 8200 (37.37%)

summary + 2970 in 00:00:30 = 99.0/s Err: 2970 (100.00%) Active: 1500

summary = 24912 in 00:05:38 = 73.7/s Err: 11170 (44.84%)

summary + 3007 in 00:00:30 = 100.1/s Err: 3007 (100.00%) Active: 1500

summary = 27919 in 00:06:08 = 75.8/s Err: 14177 (50.78%)

summary + 2968 in 00:00:30 = 99.0/s Err: 2968 (100.00%) Active: 1500

summary = 30887 in 00:06:38 = 77.6/s Err: 17145 (55.51%)

<time marker 2 – I have repaired the network between Posts and MySQL>

summary + 3007 in 00:00:30 = 100.2/s Err: 3007 (100.00%) Active: 1500

summary = 33894 in 00:07:08 = 79.2/s Err: 20152 (59.46%)

summary + 2995 in 00:00:30 = 99.8/s Err: 2995 (100.00%) Active: 1500

summary = 36889 in 00:07:38 = 80.5/s Err: 23147 (62.75%)

summary + 2997 in 00:00:30 = 99.9/s Err: 2997 (100.00%) Active: 1500

summary = 39886 in 00:08:08 = 81.7/s Err: 26144 (65.55%)

summary + 3000 in 00:00:30 = 99.9/s Err: 3000 (100.00%) Active: 1500

summary = 42886 in 00:08:38 = 82.8/s Err: 29144 (67.96%)

<another 6 minutes of 100% error!!>

summary + 3011 in 00:00:30 = 100.4/s Err: 3011 (100.00%) Active: 1500

summary = 78913 in 00:14:38 = 89.9/s Err: 65171 (82.59%)

summary + 2982 in 00:00:30 = 99.4/s Err: 2982 (100.00%) Active: 1500

summary = 81895 in 00:15:08 = 90.2/s Err: 68153 (83.22%)

summary + 3057 in 00:00:30 = 101.9/s Err: 2999 (98.10%) Active: 1500

summary = 84952 in 00:15:38 = 90.6/s Err: 71152 (83.76%)

summary + 3054 in 00:00:30 = 101.8/s Err: 2390 (78.26%) Active: 1500

summary = 88006 in 00:16:08 = 90.9/s Err: 73542 (83.56%)

summary + 2982 in 00:00:30 = 99.3/s Err: 2442 (81.89%) Active: 1500

summary = 90988 in 00:16:38 = 91.2/s Err: 75984 (83.51%)

summary + 3025 in 00:00:30 = 101.0/s Err: 2418 (79.93%) Active: 1500

summary = 94013 in 00:17:08 = 91.4/s Err: 78402 (83.39%)

summary + 2991 in 00:00:30 = 99.7/s Err: 2374 (79.37%) Active: 1500

summary = 97004 in 00:17:38 = 91.7/s Err: 80776 (83.27%)

summary + 3106 in 00:00:30 = 103.5/s Err: 2253 (72.54%) Active: 1500

summary = 100110 in 00:18:08 = 92.0/s Err: 83029 (82.94%)

summary + 3017 in 00:00:30 = 100.6/s Err: 1825 (60.49%) Active: 1500

summary = 103127 in 00:18:38 = 92.2/s Err: 84854 (82.28%)

summary + 2997 in 00:00:30 = 99.9/s Err: 1839 (61.36%) Active: 1500

summary = 106124 in 00:19:08 = 92.4/s Err: 86693 (81.69%)

summary + 2987 in 00:00:30 = 99.5/s Err: 1787 (59.83%) Active: 1500

summary = 109111 in 00:19:38 = 92.6/s Err: 88480 (81.09%)

summary + 3036 in 00:00:30 = 101.3/s Err: 1793 (59.06%) Active: 1500

summary = 112147 in 00:20:08 = 92.8/s Err: 90273 (80.50%)

summary + 2985 in 00:00:30 = 99.5/s Err: 1795 (60.13%) Active: 1500

summary = 115132 in 00:20:38 = 93.0/s Err: 92068 (79.97%)

summary + 2988 in 00:00:30 = 99.6/s Err: 1786 (59.77%) Active: 1500

summary = 118120 in 00:21:08 = 93.1/s Err: 93854 (79.46%)

summary + 3009 in 00:00:30 = 100.1/s Err: 1859 (61.78%) Active: 1500

summary = 121129 in 00:21:38 = 93.3/s Err: 95713 (79.02%)

summary + 3021 in 00:00:30 = 100.9/s Err: 1829 (60.54%) Active: 1500

summary = 124150 in 00:22:08 = 93.5/s Err: 97542 (78.57%)

summary + 3001 in 00:00:30 = 100.1/s Err: 1802 (60.05%) Active: 1500

summary = 127151 in 00:22:38 = 93.6/s Err: 99344 (78.13%)

summary + 3121 in 00:00:30 = 104.0/s Err: 1308 (41.91%) Active: 1500

summary = 130272 in 00:23:08 = 93.8/s Err: 100652 (77.26%)

summary + 3096 in 00:00:30 = 103.1/s Err: 1036 (33.46%) Active: 1500

summary = 133368 in 00:23:38 = 94.0/s Err: 101688 (76.25%)

summary + 2976 in 00:00:30 = 99.3/s Err: 596 (20.03%) Active: 1500

summary = 136344 in 00:24:08 = 94.2/s Err: 102284 (75.02%)

summary + 3005 in 00:00:30 = 100.1/s Err: 583 (19.40%) Active: 1500

summary = 139349 in 00:24:38 = 94.3/s Err: 102867 (73.82%)

summary + 3002 in 00:00:30 = 100.1/s Err: 634 (21.12%) Active: 1500

summary = 142351 in 00:25:08 = 94.4/s Err: 103501 (72.71%)

summary + 2999 in 00:00:30 = 100.0/s Err: 596 (19.87%) Active: 1500

summary = 145350 in 00:25:38 = 94.5/s Err: 104097 (71.62%)

summary + 3013 in 00:00:30 = 100.4/s Err: 580 (19.25%) Active: 1500

summary = 148363 in 00:26:08 = 94.6/s Err: 104677 (70.55%)

summary + 3016 in 00:00:30 = 100.5/s Err: 579 (19.20%) Active: 1500

summary = 151379 in 00:26:38 = 94.7/s Err: 105256 (69.53%)

summary + 2999 in 00:00:30 = 100.0/s Err: 600 (20.01%) Active: 1500

summary = 154378 in 00:27:08 = 94.8/s Err: 105856 (68.57%)

summary + 2999 in 00:00:30 = 100.0/s Err: 571 (19.04%) Active: 1500

summary = 157377 in 00:27:38 = 94.9/s Err: 106427 (67.63%)

summary + 2988 in 00:00:30 = 99.6/s Err: 600 (20.08%) Active: 1500

summary = 160365 in 00:28:08 = 95.0/s Err: 107027 (66.74%)

summary + 3107 in 00:00:30 = 103.6/s Err: 58 (1.87%) Active: 1500

summary = 163472 in 00:28:38 = 95.1/s Err: 107085 (65.51%)

summary + 2995 in 00:00:30 = 99.8/s Err: 0 (0.00%) Active: 1500

summary = 166467 in 00:29:08 = 95.2/s Err: 107085 (64.33%)

summary + 3007 in 00:00:30 = 100.2/s Err: 0 (0.00%) Active: 1500

summary = 169474 in 00:29:38 = 95.3/s Err: 107085 (63.19%)

At time marker 2 in the preceding output, after 3 minutes of having the MySQL service disconnected, you repair the network by running the following command:

./alternetwork-db.sh delete

What you’re looking for now is how long it will take before the system returns to a steady state—one where the Connections’ Posts service is experiencing 0.0% error.

As you can see, the output is rather lengthy. Approximately 9 minutes after the network was restored, you see the first signs of recovery. Then it takes another 12–13 minutes before the system is fully recovered. *This is a retry storm.* The system was so overwhelmed from the queued retries that it took well over a quarter of an hour to recover. Imagine Amazon unable to complete sales transactions for that period of time. That would be an expensive outage!

And as bad as that seems, what I’ve demonstrated here remains a small example. In a system with hundreds of connected service instances, a short network blip can result in hours-long outages that can go so far as to even crash application instances. Remember the story that started this book? That Amazon outage was ultimately caused by a retry storm that occurred after a short network outage.

|  |
| --- |
|  |

**Warning**

A retry storm can have catastrophic effects on a complex, distributed system.

|  |
| --- |
|  |

Before moving on to covering mitigations of retry storms, I’d like you to run the same test but with retries turned off. This time, when the attempt to access the Posts service times out, the Connections’ Posts service will return an error with no result, but it will return. To turn off retries, change the value of the env variable CONNECTIONPOSTCONTROLLER\_IMPLEMENTRETRIES in the cookbook-deployment-kubernetes-connectionposts.yaml file to false and update the deployment with the following command:

kubectl apply -f cookbook-deployment-kubernetes-connectionposts.yaml

You can then create the JMeter pod as you’ve previously done with the kubectl create command (if you hadn’t already deleted the prior deployment, please do so first with the kubectl delete deploy command). The following is the output of JMeter, with two time markers inserted:

START Running Jmeter on Mon Feb 18 20:58:54 UTC 2019

JVM\_ARGS=-Xmn528m -Xms2112m -Xmx2112m

jmeter args=-n -t /etc/jmeter/jmeter\_run.jmx

Feb 18, 2019 8:58:56 PM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

Creating summariser <summary>

Created the tree successfully using /etc/jmeter/jmeter\_run.jmx

Starting the test @ Mon Feb 18 20:58:56 UTC 2019 (1550523536966)

Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445

summary + 18 in 00:00:02 = 7.9/s Err: 0 (0.00%) Active: 18

summary + 401 in 00:00:30 = 13.4/s Err: 0 (0.00%) Active: 263

summary = 419 in 00:00:32 = 13.0/s Err: 0 (0.00%)

summary + 890 in 00:00:30 = 29.7/s Err: 0 (0.00%) Active: 506

summary = 1309 in 00:01:02 = 21.0/s Err: 0 (0.00%)

summary + 1378 in 00:00:30 = 46.0/s Err: 0 (0.00%) Active: 752

summary = 2687 in 00:01:32 = 29.1/s Err: 0 (0.00%)

summary + 1877 in 00:00:30 = 62.6/s Err: 0 (0.00%) Active: 1000

summary = 4564 in 00:02:02 = 37.3/s Err: 0 (0.00%)

summary + 2369 in 00:00:30 = 79.0/s Err: 0 (0.00%) Active: 1249

summary = 6933 in 00:02:32 = 45.5/s Err: 0 (0.00%)

summary + 2869 in 00:00:30 = 95.6/s Err: 0 (0.00%) Active: 1498

summary = 9802 in 00:03:02 = 53.8/s Err: 0 (0.00%)

summary + 3004 in 00:00:30 = 100.2/s Err: 0 (0.00%) Active: 1500

summary = 12806 in 00:03:32 = 60.3/s Err: 0 (0.00%)

summary + 2998 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 15804 in 00:04:02 = 65.2/s Err: 0 (0.00%)

summary + 3001 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 18805 in 00:04:32 = 69.1/s Err: 0 (0.00%)

<time marker 1 – I have broken the network between Posts and MySQL>

summary + 2951 in 00:00:30 = 98.4/s Err: 2662 (90.21%) Active: 1500

summary = 21756 in 00:05:02 = 72.0/s Err: 2662 (12.24%)

summary + 2999 in 00:00:30 = 100.0/s Err: 2999 (100.00%) Active: 1500

summary = 24755 in 00:05:32 = 74.5/s Err: 5661 (22.87%)

summary + 3001 in 00:00:30 = 100.0/s Err: 3001 (100.00%) Active: 1500

summary = 27756 in 00:06:02 = 76.6/s Err: 8662 (31.21%)

summary + 3000 in 00:00:30 = 100.0/s Err: 3000 (100.00%) Active: 1500

summary = 30756 in 00:06:32 = 78.4/s Err: 11662 (37.92%)

summary + 3001 in 00:00:30 = 100.0/s Err: 3001 (100.00%) Active: 1500

summary = 33757 in 00:07:02 = 80.0/s Err: 14663 (43.44%)

summary + 3000 in 00:00:30 = 100.0/s Err: 3000 (100.00%) Active: 1500

summary = 36757 in 00:07:32 = 81.3/s Err: 17663 (48.05%)

summary + 2999 in 00:00:30 = 100.0/s Err: 2999 (100.00%) Active: 1500

summary = 39756 in 00:08:02 = 82.4/s Err: 20662 (51.97%)

<time marker 2 – I have repaired the network between Posts and MySQL>

summary + 3051 in 00:00:30 = 101.7/s Err: 1473 (48.28%) Active: 1500

summary = 42807 in 00:08:32 = 83.6/s Err: 22135 (51.71%)

summary + 2999 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 45806 in 00:09:02 = 84.5/s Err: 22135 (48.32%)

As you can see, while the network is disrupted, the Connections’ Posts service reports 100% error. But most important, as soon as the network is reestablished, at time marker 2, the system immediately returns to a stable state with 0.0% error. There are no queued retries overwhelming the system.

|  |
| --- |
|  |

**Note**

When employing retries, the system took 15 minutes to recover from a 3-minute network disruption. When *not* employing retries, the recovery from a 3-minute network disruption was immediate.

|  |
| --- |
|  |

So you’re faced with a paradox. Retries can cause catastrophic effects, yet they can also provide great benefits, particularly when failed calls are only intermittent. Is there a way to take advantage of the benefits of retries without the risk of a retry storm wreaking havoc on the system? Indeed there is. There are several. In this chapter, I’ll talk about being smarter about the way we do retries—about being a kinder client. The next chapter covers putting up protections in front of a service to keep less-kind clients from causing problems in the system.

**9.1.6. Avoiding retry storms: Kind clients**

Despite the dramatically negative consequences that you saw with retries in the previous section, their value remains obvious. For intermittent connectivity issues in particular, retries will often work, thereby snuffing an error that could otherwise have propagated widely through the distributed system that makes up our cloud-native software. The trick then is to balance the tension between the potential negative effects and the positive ones.

The first observation we can make is that for the issues that present themselves only sporadically and for limited durations, it rarely takes more than one or two repeat attempts to have a successful exchange. Therefore, the first control you could put in place on our retry loop is to limit the total number of such retry attempts. So, for example, instead of having a while loop that runs indefinitely, you could implement a counter and stop the retries when you’ve hit a threshold.

But then, what happens when a connection is only momentarily unavailable, but all retry attempts have been exhausted before connectivity is reestablished? You’ve just lost out on the benefits of retries because you were overzealous on repeating your requests. Introducing a delay between retry attempts lends a bit of balance here.

**9.1.7. Let’s see this in action: being a kinder client**

Let’s apply two controls, limiting the number of retry attempts and slowing the rate of retries, to our implementation and see how this changes the behavior of our software, particularly while under load.

I won’t repeat all of the setup and build instructions again; my presentation here is just an extension to that of the preceding section. To access the new implementation, check out the following tag from the Git repo:

git checkout requestretries/0.0.3

In the next listing, you’ll see that in the place where you formerly had the naïve retry implementation, you now have the following code.

**Listing 9.2. Excerpt from ConnectionsPostsController.java**

try {

postSummaries = postsServiceClient.getPosts(ids, restTemplate);

response.setStatus(200);

return postSummaries;

} catch (HttpServerErrorException e) {

logger.info(utils.ipTag() + "Call to Posts service returned 500");

response.setStatus(500);

return null;

} catch (ResourceAccessException e) {

logger.info(utils.ipTag() + "Call to Posts service timed out");

response.setStatus(500);

return null;

} catch (Exception e) {

logger.info(utils.ipTag() + "Unexpected Exception: Exception Class "

+ e.getClass() + e.getMessage());

response.setStatus(500);

return null;

}

Notice that the only difference in the various catch blocks is the message that’s logged, so logically, the implementation is now as follows:

try {

postSummaries = postsServiceClient.getPosts(ids, restTemplate);

response.setStatus(200);

return postSummaries;

} catch (Exception e) {

logger.info(utils.ipTag() + e.getMessage());

response.setStatus(500);

return null;

}

You’ll also notice that calling the Posts service is now facilitated through a new class, PostsServiceClient, which is a client for the Posts service. Creating this class provides a surface area against which the Spring Retry annotations can be applied.

With the preceding code, if the call to the Posts service is successful, you return the set of posts obtained through the postsServiceClient.getPosts invocation. Otherwise, set HTTP status to 500 (an error) and return nothing. Let’s have a look at the implementation of this Posts service client.

**Listing 9.3. Method from PostsServiceClient.java**

@Retryable( value = ResourceAccessException.class,

maxAttempts = 3,

backoff = @Backoff(delay = 500))

public ArrayList<PostSummary> getPosts(String ids,

RestTemplate restTemplate) throws Exception {

ArrayList<PostSummary> postSummaries = new ArrayList<PostSummary>();

String secretQueryParam = "&secret=" + utils.getPostsSecret();

logger.info("Trying getPosts: " + postsUrl + ids + secretQueryParam);

ResponseEntity<ConnectionsPostsController.PostResult[]> respPosts

= restTemplate.getForEntity(postsUrl + ids + secretQueryParam,

ConnectionsPostsController.PostResult[].class);

if (respPosts.getStatusCode().is5xxServerError()) {

throw new HttpServerErrorException(respPosts.getStatusCode(),

"Exception thrown in obtaining Posts");

} else {

ConnectionsPostsController.PostResult[] posts

= respPosts.getBody();

for (int i = 0; i < posts.length; i++)

postSummaries.add(

new PostSummary(

getUsersname(posts[i].getUserId(),restTemplate),

posts[i].getTitle(), posts[i].getDate()));

return postSummaries;

}

}

This code uses a project that’s part of the Spring Framework, Spring Retries (<https://github.com/spring-projects/spring-retry>). I find it interesting that the retry patterns now encapsulated in this project were originally embedded within the Spring Batch project. Having been extracted into its own project allows it to be used in many scenarios; as a case in point, the first line of the README for the Spring Retry project says, “It is used in Spring Batch, Spring Integration, Spring for Apache Hadoop (amongst others).” Retries are so ubiquitous in cloud-native software that it makes sense to have a library so that you can easily use them in many use cases.

I want to draw your attention to two parts of this code. First, notice that the @Retryable annotation includes attributes that reflect exactly the controls I talked about previously: limiting the number of retries, and giving some time between retry attempts (you’ll wait half a second between attempts). Notice also that you can specify that retries should be attempted for only certain exceptions—in this case, an access (connect or read timeout) exception.

The other thing that you’ll notice in studying the code is that you’re no longer responsible for the looping logic. This code simply implements the happy path. It makes the HTTP request to Posts, and if that returns an error HTTP status code, it passes that error up. Otherwise, it processes the response body and returns those values. There’s no try/catch, no loop. If, however, this code were to generate a ResourceAccessException, which can be thrown by the restTemplate, the Spring Retry implementation would catch it, and based on the annotation values, perhaps execute the method again. On a side note, Spring Retry achieves this via aspects; hence, the inclusion of the AOP (Aspect-Oriented Programming) dependency along with the Spring Retry one.

**Listing 9.4. Excerpt from pom.xml for the Connections’ Posts service**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.retry</groupId>

<artifactId>spring-retry</artifactId>

<version>1.2.2.RELEASE</version>

</dependency>

Let’s see what this implementation does in our load scenario from the previous section. If you want to follow along, you’ll, of course, have to redeploy the software. If you ran the examples in the previous section, you can run the deployApps.sh bash script. You’ll then place exactly the same load against this deployment as you did for the previous. The following is the output (as can be seen in the logs for the JMeter pod), once again with two time markers inserted:

START Running Jmeter on Mon Feb 18 21:58:55 UTC 2019

JVM\_ARGS=-Xmn502m -Xms2008m -Xmx2008m

jmeter args=-n -t /etc/jmeter/jmeter\_run.jmx -l resultsconnectionsposts

Feb 18, 2019 9:58:57 PM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

Creating summariser <summary>

Created the tree successfully using /etc/jmeter/jmeter\_run.jmx

Starting the test @ Mon Feb 18 21:58:57 UTC 2019 (1550527137576)

Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445

summary + 14 in 00:00:02 = 8.1/s Err: 0 (0.00%) Active: 14

summary + 394 in 00:00:30 = 13.2/s Err: 0 (0.00%) Active: 259

summary = 408 in 00:00:32 = 12.9/s Err: 0 (0.00%)

summary + 887 in 00:00:30 = 29.6/s Err: 0 (0.00%) Active: 508

summary = 1295 in 00:01:02 = 21.0/s Err: 0 (0.00%)

summary + 1388 in 00:00:30 = 46.3/s Err: 0 (0.00%) Active: 756

summary = 2683 in 00:01:32 = 29.3/s Err: 0 (0.00%)

summary + 1887 in 00:00:30 = 62.9/s Err: 0 (0.00%) Active: 1005

summary = 4570 in 00:02:02 = 37.6/s Err: 0 (0.00%)

summary + 2377 in 00:00:30 = 79.3/s Err: 0 (0.00%) Active: 1253

summary = 6947 in 00:02:32 = 45.8/s Err: 0 (0.00%)

summary + 2878 in 00:00:30 = 95.9/s Err: 0 (0.00%) Active: 1500

summary = 9825 in 00:03:02 = 54.1/s Err: 0 (0.00%)

summary + 2993 in 00:00:30 = 99.7/s Err: 0 (0.00%) Active: 1500

summary = 12818 in 00:03:32 = 60.6/s Err: 0 (0.00%)

summary + 3006 in 00:00:30 = 100.2/s Err: 0 (0.00%) Active: 1500

summary = 15824 in 00:04:02 = 65.5/s Err: 0 (0.00%)

<time marker 1 – I have broken the network between Posts and MySQL>

summary + 2645 in 00:00:30 = 88.2/s Err: 2354 (89.00%) Active: 1500

summary = 18469 in 00:04:32 = 68.0/s Err: 2354 (12.75%)

summary + 3002 in 00:00:30 = 100.0/s Err: 3002 (100.00%) Active: 1500

summary = 21471 in 00:05:02 = 71.2/s Err: 5356 (24.95%)

summary + 3000 in 00:00:30 = 100.0/s Err: 3000 (100.00%) Active: 1500

summary = 24471 in 00:05:32 = 73.8/s Err: 8356 (34.15%)

summary + 3006 in 00:00:30 = 100.2/s Err: 3006 (100.00%) Active: 1500

summary = 27477 in 00:06:02 = 76.0/s Err: 11362 (41.35%)

summary + 3015 in 00:00:30 = 100.5/s Err: 3015 (100.00%) Active: 1500

summary = 30492 in 00:06:32 = 77.9/s Err: 14377 (47.15%)

summary + 3051 in 00:00:30 = 101.7/s Err: 3051 (100.00%) Active: 1500

summary = 33543 in 00:07:02 = 79.6/s Err: 17428 (51.96%)

<time marker 2 – I have repaired the network between Posts and MySQL>

summary + 3002 in 00:00:30 = 100.0/s Err: 3002 (100.00%) Active: 1500

summary = 36545 in 00:07:32 = 80.9/s Err: 20430 (55.90%)

summary + 2942 in 00:00:30 = 98.1/s Err: 2942 (100.00%) Active: 1500

summary = 39487 in 00:08:02 = 82.0/s Err: 23372 (59.19%)

summary + 3323 in 00:00:30 = 110.8/s Err: 378 (11.38%) Active: 1500

summary = 42810 in 00:08:32 = 83.7/s Err: 23750 (55.48%)

summary + 3021 in 00:00:30 = 100.6/s Err: 2 (0.07%) Active: 1500

summary = 45831 in 00:09:02 = 84.6/s Err: 23752 (51.83%)

summary + 2998 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 48829 in 00:09:32 = 85.4/s Err: 23752 (48.64%)

summary + 3001 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 51830 in 00:10:02 = 86.1/s Err: 23752 (45.83%)

When you begin the test, you’re seeing 0.0% errors coming from Connections’ Posts. The calls to the Posts service, and all other processing, are all completing successfully. At time marker 1, you disconnect the Posts and MySQL services by using the same route commands in the MySQL containers via the following command:

./alternetwork-db.sh add

As you can see, the error quickly reaches 100% because the Connections’ Posts service will return a server error if it doesn’t receive a result from the Posts service, even while retries are implemented. But now look at what happens after time marker 2, when you reestablish the network by running this:

./alternetwork-db.sh delete

You see the first signs of recovery in only 1 minute and are fully recovered in less than 3. You avoided the retry storm, even in the most extreme conditions where the network was lost for minutes.

This might leave you wondering whether this implementation provides value in the more intermittent error scenarios. Let’s simulate this by disconnecting only one of the Posts services from the network. You can do this by executing a single one of the kubectl commands from the alternetwork-db.sh script. For example:

kubectl exec mysql-57bdb878f5-dhlck -- route $1 -host 10.36.4.11 reject

What you’ve done is broken only a single connection from one instance of a Posts service to the MySQL service, exactly as shown in [figure 9.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig05).

Looking at the JMeter log output, you can see that although the Posts service is having trouble because of the lack of connectivity to MySQL (starting at time marker 1), many of the resultant failed attempts from Connections’ Posts are being washed completely away by the retries. You’ve disconnected only a single instance of the Posts service from MySQL. On average, 25% of the requests from Connections’ Posts to Posts will fail. But as you can see in the following output, the overall error is far smaller—less than 1%. And when connectivity is restored at time marker 2, the error rate immediately returns to 0.0%:

START Running Jmeter on Mon Feb 18 22:16:50 UTC 2019

JVM\_ARGS=-Xmn524m -Xms2096m -Xmx2096m

jmeter args=-n -t /etc/jmeter/jmeter\_run.jmx -l resultsconnectionsposts

Feb 18, 2019 10:16:52 PM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

Creating summariser <summary>

Created the tree successfully using /etc/jmeter/jmeter\_run.jmx

Starting the test @ Mon Feb 18 22:16:52 UTC 2019 (1550528212234)

Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445

summary + 58 in 00:00:07 = 8.2/s Err: 0 (0.00%) Active: 58

summary + 483 in 00:00:30 = 16.1/s Err: 0 (0.00%) Active: 304

summary = 541 in 00:00:37 = 14.6/s Err: 0 (0.00%)

summary + 982 in 00:00:30 = 32.7/s Err: 0 (0.00%) Active: 553

summary = 1523 in 00:01:07 = 22.7/s Err: 0 (0.00%)

summary + 1477 in 00:00:30 = 49.3/s Err: 0 (0.00%) Active: 802

summary = 3000 in 00:01:37 = 30.9/s Err: 0 (0.00%)

summary + 1974 in 00:00:30 = 65.8/s Err: 0 (0.00%) Active: 1049

summary = 4974 in 00:02:07 = 39.2/s Err: 0 (0.00%)

summary + 2473 in 00:00:30 = 82.4/s Err: 0 (0.00%) Active: 1298

summary = 7447 in 00:02:37 = 47.4/s Err: 0 (0.00%)

summary + 2920 in 00:00:30 = 97.4/s Err: 0 (0.00%) Active: 1500

summary = 10367 in 00:03:07 = 55.4/s Err: 0 (0.00%)

<time marker 1 – I have broken a **single** connection between Posts and MySQL>

summary + 2998 in 00:00:30 = 99.9/s Err: 3 (0.10%) Active: 1500

summary = 13365 in 00:03:37 = 61.6/s Err: 3 (0.02%)

summary + 2999 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 16364 in 00:04:07 = 66.3/s Err: 3 (0.02%)

summary + 2993 in 00:00:30 = 99.8/s Err: 1 (0.03%) Active: 1500

summary = 19357 in 00:04:37 = 69.9/s Err: 4 (0.02%)

summary + 3001 in 00:00:30 = 100.1/s Err: 1 (0.03%) Active: 1500

summary = 22358 in 00:05:07 = 72.8/s Err: 5 (0.02%)

summary + 2994 in 00:00:30 = 99.8/s Err: 1 (0.03%) Active: 1500

summary = 25352 in 00:05:37 = 75.2/s Err: 6 (0.02%)

summary + 3005 in 00:00:30 = 100.1/s Err: 2 (0.07%) Active: 1500

summary = 28357 in 00:06:07 = 77.3/s Err: 8 (0.03%)

summary + 3001 in 00:00:30 = 100.1/s Err: 1 (0.03%) Active: 1500

summary = 31358 in 00:06:37 = 79.0/s Err: 9 (0.03%)

<time marker 2 – I have repaired the connection between Posts and MySQL>

summary + 2999 in 00:00:30 = 100.0/s Err: 1 (0.03%) Active: 1500

summary = 34357 in 00:07:07 = 80.5/s Err: 10 (0.03%)

summary + 3000 in 00:00:30 = 100.0/s Err: 1 (0.03%) Active: 1500

summary = 37357 in 00:07:37 = 81.7/s Err: 11 (0.03%)

summary + 3009 in 00:00:30 = 100.3/s Err: 0 (0.00%) Active: 1500

summary = 40366 in 00:08:07 = 82.9/s Err: 11 (0.03%)

What you see here is that with only a few simple controls, limiting the number of retry attempts and taking some time between them, you can realize the benefits of retries while avoiding worsening conditions in an already degraded system.

**9.1.8. When not to retry**

You’ve just clearly seen the benefits of retries, and you should use them liberally in your software designs. Except when you shouldn’t. There will be many nuanced reasons that you may want to avoid retries (using caching as an alternative may boost performance, for example), and I won’t cover those here. But I’d like to spend a moment circling back to a topic I introduced at the start of this chapter: times that it’s unsafe to perform retries (when you don’t receive a response after clicking the Purchase button, for example).

I chose the word “safe” here quite intentionally because there’s a formal definition of *safety* in HTTP that gets at precisely the point I want to make. Here are two definitions from the HTTP spec ([www.w3.org/Protocols/rfc2616/rfc2616-sec9.html](http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html)):

* A *safe* method is one that may be invoked *zero* or more times with the same effect. The method must not have any side effects.
* An *idempotent* method is one where invoking the method *one* or more times will have the same effect. Side effects are permitted in this case, but all repeated invocations must have the same side effect as the first.

What you’re doing with retries is addressing the “or more” part of those statements. But which one of these statements applies to our pattern? In short, it’s the former—*only safe methods should be retried*. When you make a request over the network, there’s no guarantee that any of your requests will reach their intended recipient, so you may end up in a situation where *zero* of your attempts are successful. Therefore, as a general rule, only safe methods should be retried. If you wish to implement any failure handling around nonsafe methods, you must implement compensating behaviors such as Sagas.

Here’s an important point: it’s up to you, the developer, to know whether the invocations you’re making are safe or not. Referring back to the HTTP specification, you see that the HTTP requests that are safe are GET, HEAD, OPTIONS, and TRACE. But Spring Retry doesn’t have visibility into any of the HTTP requests you make from within @Retryable methods, so it’s up to you to add that annotation only to methods that are safe. If you had a method that encapsulated a POST request that deducted $100 from your bank account, you’d be disappointed with retries. Apply retries only when it’s safe to do so.

**9.2. Fallback logic**

*Design for failure.* This is a mantra for cloud-native software that I hope you’ve already been learning throughout reading this book. As a case in point, retrying a request when the first fails is a good design. But what do you do when your attempts at recovery are also unsuccessful? What happens when you retry several times and still get no response? What you’ve done in the preceding examples of this chapter is return an error, but you can do better.

One of the most fundamental patterns in designing for failure is to implement fallback methods—code that’s executed when the main logic fails. Sure, sometimes when the software can’t complete its task, the right thing is to return an error. But in this world of highly distributed, constantly changing software deployments with an abundance of failure scenarios, you need to build new muscle. You need to establish a habit of thinking about alternative results, even if less than ideal.

The example that has been running through this chapter provides an excellent opportunity to exercise that muscle, and extending the retry logic is the perfect place to do it. When designing fallback behaviors (and any of the resilience patterns described in this book), you need to think through the real-world scenario your software is addressing. In the part of the implementation that you’ll extend, you’re trying to get the list of blog posts for a set of users. Although some of those users may be rather prolific, perhaps posting several times a week or even more than once a day, the creation of new blog posts is something that still happens rather infrequently. If a user were to access their aggregated feed at a time when the MySQL database storing the posts were unavailable, it might be better to return a set of posts that may be missing only the newest entries instead of returning nothing at all. I can tell you personally, when I’m accessing an aggregated set of recipes to decide what to make for dinner this eve, I can still produce something pretty tasty even if I don’t have the absolute latest recipe that Food52 has posted.

**9.2.1. Let’s see this in action: Implementing fallback logic**

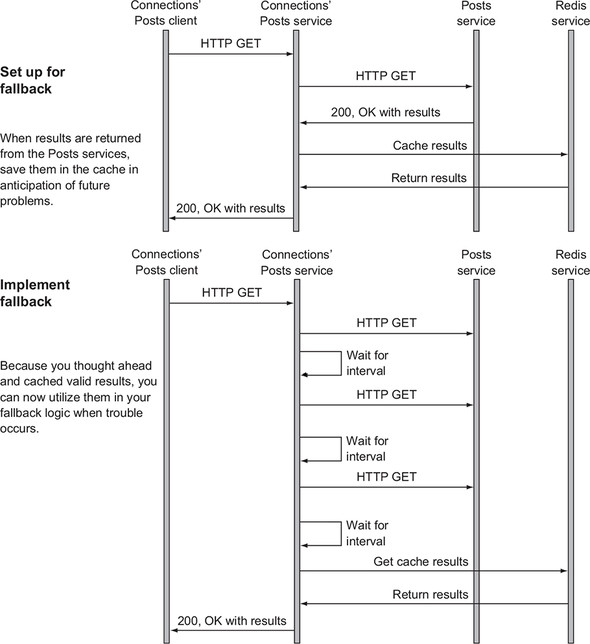
Let’s look at this in practice. Check out the following tag from the repository:

git checkout requestretries/0.0.4

I won’t repeat the optional build instructions here. Please see the earlier examples in this chapter (and book) if you want to change the code and do deployments yourself. As always, I’ve prebuilt everything and have made Docker images available in Docker Hub.

Before testing it, let’s have a look at the implementation. For fallback in the case when the Posts service isn’t providing a valid result, your Connections’ Posts service will simply return the latest Posts that it has previously seen. To do this, you’ve added simple caching to the implementation. Recall that our Connections’ Posts implementation already binds to a Redis key/value store, a database ideally suited for caching. So now when a call to Posts yields a result, the logic in Connections’ Posts will store that result in Redis before returning it. That store sets you up to be able to then implement the fallback behavior when the Posts service is in a bad way. The upper part of [figure 9.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig06) shows the flow that caches results when the Posts service is reachable and delivering results. The lower part of [figure 9.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig06) shows the flow that reads results from the cache when Posts is experiencing trouble.

**Figure 9.6. Thinking ahead, you cache results when they’re available, so that when you later experience trouble, you have those cached values to use as a part of your fallback logic.**



Adding the fallback implementation then is simple. With Spring Retry, you add a method to your service with the @Recover annotation, and Spring will invoke that method after all retry attempts have been exhausted. The method signature must match that of the method implementing the main logic, with the addition of an exception type as the first argument. The recover method will be called only under certain circumstances, as defined by the type of error.

**Listing 9.5. Method from PostsServiceClient.java**

@Recover

public ArrayList<PostSummary> returnCached(

ResourceAccessException e,

String ids, RestTemplate restTemplate)

throws Exception {

logger.info("Failed ... Posts service - returning cached results");

PostResults postResults = postResultsRepository.findOne(ids);

ObjectMapper objectMapper = new ObjectMapper();

ArrayList<PostSummary> postSummaries;

try {

postSummaries = objectMapper.readValue(

postResults.getSummariesJson(),

new TypeReference<ArrayList<PostSummary>>() {});

} catch (Exception ec) {

logger.info("Exception on deserialization " + ec.getClass()

+ " message = " + ec.getMessage());

return null;

}

return postSummaries;

}

Although it may seem obvious as you look at this simple example, I do want to draw your attention to the fact that in most cases your fallback behavior requires some setup. In this example, the preceding code isn’t all that’s required for fallback. The logic that caches results when they’re successfully obtained is needed setup for this. Our @Retryable method is the place where you’re thinking ahead to darker days.

**Listing 9.6. Method from PostsServiceClient.java**

@Retryable( value = ResourceAccessException.class,

maxAttempts = 3, backoff = @Backoff(delay = 500))

public ArrayList<PostSummary> getPosts(String ids,

RestTemplate restTemplate)

throws Exception {

ArrayList<PostSummary> postSummaries = new ArrayList<PostSummary>();

String secretQueryParam = "&secret=" + utils.getPostsSecret();

logger.info("Trying getPosts: " + postsUrl + ids + secretQueryParam);

ResponseEntity<ConnectionsPostsController.PostResult[]> respPosts

= restTemplate.getForEntity(

postsUrl + ids + secretQueryParam,

ConnectionsPostsController.PostResult[].class);

if (respPosts.getStatusCode().is5xxServerError()) {

throw new HttpServerErrorException(respPosts.getStatusCode(),

"Exception thrown in obtaining Posts");

} else {

ConnectionsPostsController.PostResult[] posts = respPosts.getBody();

for (int i = 0; i < posts.length; i++)

postSummaries.add(

new PostSummary(getUsersname(posts[i].getUserId(), restTemplate),

posts[i].getTitle(), posts[i].getDate()));

**// thinking ahead to darker days, cache the result**

**ObjectMapper objectMapper = new ObjectMapper();**

**String postSummariesJson =**

**objectMapper.writeValueAsString(postSummaries);**

**PostResults postResults = new PostResults(ids, postSummariesJson);**

**postResultsRepository.save(postResults);**

return postSummaries;

}

}

Let’s now have a look at the effect that adding fallback behavior has on the stability of our implementation. You’re going to run the same load test that you’ve done previously. If you wish to follow along, please update your deployment by rerunning the application deployment script:

./deployApps.sh

You now run the load test with our usual command:

kubectl create -f loadTesting/jmeter-deployment.yaml

As always, after the load test has reached full capacity, you break the network between all instances of the Posts service and the MySQL service for 3 minutes (time marker 1) and then restore the network (time marker 2). Before you look at the results of the test run, let’s look at the log output for one of the Connections’ Posts services:

(the network is currently broken...)

... : [10.36.4.11:8080] getting posts for user network cdavisafc

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret

... : **Failed to connect to or obtain results from Posts service - returning**

**cached results**

... : **Failed to connect to or obtain results from Posts service - returning**

**cached results**

... : [10.36.4.11:8080] connections = 2,3

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret

... : **Failed to connect to or obtain results from Posts service - returning**

**cached results**

(after restoring the network)

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret

... : [10.36.4.11:8080] getting posts for user network cdavisafc

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret

... : [10.36.4.11:8080] connections = 2,3

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret

... : [10.36.4.11:8080] getting posts for user network cdavisafc

... : [10.36.4.11:8080] connections = 2,3

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret

... : [10.36.4.11:8080] getting posts for user network cdavisafc

... : [10.36.4.11:8080] connections = 2,3

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret

... : Trying getPosts: http://posts-svc/posts?userIds=2,3&secret=newSecret

While the network is disrupted, Spring Retry first repeats the access attempt three times and then calls the @Recover method, returning cached results. After the network is restored, live results are once again returned.

Let’s now have a look at how this implementation fares under load. The following is the log output from the JMeter test:

START Running Jmeter on Mon Feb 18 23:10:22 UTC 2019

JVM\_ARGS=-Xmn506m -Xms2024m -Xmx2024m

jmeter args=-n -t /etc/jmeter/jmeter\_run.jmx -l resultsconnectionsposts

Feb 18, 2019 11:10:24 PM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

Creating summariser <summary>

Created the tree successfully using /etc/jmeter/jmeter\_run.jmx

Starting the test @ Mon Feb 18 23:10:24 UTC 2019 (1550531424214)

Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445

summary + 194 in 00:00:19 = 10.0/s Err: 0 (0.00%) Active: 159

summary + 687 in 00:00:30 = 22.9/s Err: 0 (0.00%) Active: 406

summary = 881 in 00:00:49 = 17.8/s Err: 0 (0.00%)

summary + 1184 in 00:00:30 = 39.5/s Err: 0 (0.00%) Active: 655

summary = 2065 in 00:01:19 = 26.0/s Err: 0 (0.00%)

summary + 1682 in 00:00:30 = 56.1/s Err: 0 (0.00%) Active: 904

summary = 3747 in 00:01:49 = 34.2/s Err: 0 (0.00%)

summary + 2176 in 00:00:30 = 72.6/s Err: 0 (0.00%) Active: 1151

summary = 5923 in 00:02:19 = 42.5/s Err: 0 (0.00%)

summary + 2676 in 00:00:30 = 89.2/s Err: 0 (0.00%) Active: 1400

summary = 8599 in 00:02:49 = 50.8/s Err: 0 (0.00%)

summary + 3000 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 11599 in 00:03:19 = 58.2/s Err: 0 (0.00%)

<time marker 1 – I have broken the network between Posts and MySQL>

summary + 2752 in 00:00:30 = 91.7/s Err: 0 (0.00%) Active: 1500

summary = 14351 in 00:03:49 = 62.6/s Err: 0 (0.00%)

summary + 3000 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 17351 in 00:04:19 = 66.9/s Err: 0 (0.00%)

summary + 3001 in 00:00:30 = 100.1/s Err: 0 (0.00%) Active: 1500

summary = 20352 in 00:04:49 = 70.3/s Err: 0 (0.00%)

summary + 2998 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 23350 in 00:05:19 = 73.1/s Err: 0 (0.00%)

summary + 3038 in 00:00:30 = 101.3/s Err: 0 (0.00%) Active: 1500

summary = 26388 in 00:05:49 = 75.5/s Err: 0 (0.00%)

summary + 3039 in 00:00:30 = 101.3/s Err: 0 (0.00%) Active: 1500

summary = 29427 in 00:06:19 = 77.6/s Err: 0 (0.00%)

summary + 3000 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 32427 in 00:06:49 = 79.2/s Err: 0 (0.00%)

<time marker 2 – I have repaired the network between Posts and MySQL>

summary + 3089 in 00:00:30 = 102.9/s Err: 0 (0.00%) Active: 1500

summary = 35516 in 00:07:19 = 80.8/s Err: 0 (0.00%)

summary + 3080 in 00:00:30 = 102.7/s Err: 0 (0.00%) Active: 1500

summary = 38596 in 00:07:49 = 82.2/s Err: 0 (0.00%)

It’s exactly as you’d expect. Because Connections’ Posts is returning cached results when the downstream dependent Posts service isn’t producing results, the client of Connections’ Posts (JMeter) never sees an error during the outage. Yet as you saw in the preceding Connections’ Posts log, as soon as the network is restored, live values are returned.

|  |
| --- |
|  |

**Note**

This is pretty solid. The users of your software saw *no* errors, even while parts of the system were experiencing trouble. Trouble in one part of your software didn’t cascade through the distributed system.

|  |
| --- |
|  |

Reflecting back on the series of tests you’ve just run, [table 9.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09table01) summarizes the results.

**Table 9.1. Employing a simple retry with other patterns can eliminate or lessen negative impacts.**

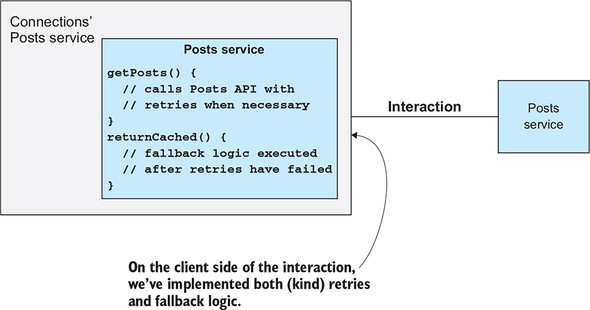
| Version of Connections’ Posts service | During network outage | Time to initial signs of recovery | Time to full recovery |
| --- | --- | --- | --- |
| Naïve retries | 100% error | 9 minutes | 12–13 minutes |
| Kind retries using Spring Retry and no fallback | 100% error | 1 minute | 3 minutes |
| Kind retries using Spring Retry with fallback method | 0.0% error | N/A—no failure during network outage | N/A |

As is clear from this summary, employing an otherwise good cloud-native pattern in an overly simplistic way can generate additional problems. But by using the simple retry with other patterns, the negative impacts can be significantly lessened or even entirely eliminated.

As you can see, the various compensating behaviors you’ve implemented have had a tremendous positive impact on the stability of the system and the user experience. Designing for failure makes a difference!

Now that you’ve implemented several patterns on the client side of the interaction, let’s look back at the opening diagram of the chapter and fill in a few details. In [figure 9.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09fig07), you can see that the client side has implemented both retries and fallback behavior. In [chapter 10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10), you’ll turn to the other end of the interaction.

**Figure 9.7. Implementation of client-side patterns such as retries and fallbacks yields a far more robust system. (You’ll turn to the services side of the interaction in the next chapter.)**



**9.3. Control loops**

Despite their seeming simplicity, I’ve just spent a great deal of time covering retries for two reasons. First, they’re an essential tool for building resilient distributed systems, and as you saw, can be tricky to get just right. But even more important, I use them as a concrete example of a more generic pattern that I want to say more about now: control loops.

**9.3.1. Understanding types of control loops**

The retries you’ve been studying here aren’t the first example of redundant actions in this text, though up until now I mentioned them only briefly. For instance, by deploying your apps into a Kubernetes environment, you’re taking advantage of at least one of the control loops built into that runtime platform: the replication controller. The Kubernetes replication controller implements a control loop that allows you to specify your app deployment declaratively, and Kubernetes will create and maintain that application topology. The control loop never expects to reach a *done* state. It’s designed to constantly be looking for the inevitable change and to respond appropriately.

This isn’t a book on Kubernetes, so I won’t cover them in detail, but the controllers are constantly (in a control loop) comparing the actual state of workloads running in the Kubernetes cluster to the desired state of those workloads, which it obtains (in the control loop) from the Kubernetes API server, the source of truth for desired cluster state. Here’s a small sampling of some of the control loops implemented by that platform:

* ***Replication controller—***This controller is in the business of managing deployments (see the YAML files for our app deployments), ensuring that the desired number of replicas remain running through failures and upgrades.
* ***Daemonset controller—***A Kubernetes daemonset defines pods; exactly one will be running on each worker node (physical or virtual machine) of the Kubernetes cluster. The daemonset controller ensures that all nodes have all desired daemonsets deployed.
* ***Endpoints controller—***As workloads are deployed and assigned IP addresses dynamically, the endpoints controller will update the Kubernetes DNS service (among other things).
* ***Namespaces controller—***Namespaces may be used as a tenancy within a Kubernetes cluster, and certain policies may be applied to namespaces upon their creation. For example, a network segment may be created and assigned to isolate network traffic for apps deployed into that namespace. The namespaces controller watches for changes in the list of Kubernetes namespaces and performs any necessary actions.

Let me home in on that last example for a moment. I’ve been talking about control *loops*. Why do you need a loop? Can’t your system simply perform the necessary actions when, for example, someone issues the kubectl create namespace command? In theory, yes. But as you saw in the example of the first part of this chapter, it’s quite possible that this code may not be accessible when the command is executed. If that happens, do you fail to create the namespace? Do you automatically retry a time or two? The controller pattern is explicitly designed to handle these types of problems, and it does this so well that its use permeates throughout a modern distributed system such as Kubernetes. So too should you apply it liberally in your cloud-native software.

**9.3.2. Controlling the control loop**

Earlier in the chapter, I talked about controlling the retry loop. The controls you applied did things like limit the number of times the loop was executed, control its cadence, and select the conditions under which an action would be initiated (the type of exception). Again, just as the retry loop generalizes to a basic control loop, so too do some of the parameters you can apply to them.

For example, the type of exception that a @Retryable method is applied to is akin to a data type. Notice that the Kubernetes controllers I listed previously each applied to different types of Kubernetes objects. Although in a general sense a controller loops indefinitely, it’s perfectly acceptable to change this tenet, as you did when limiting the total number of retries for a particular remote request. And finally, let’s look at the cadence at which actions are taken as a result of the control loop.

When talking about request retries, I showed you an example; you slowed the rate of retries (to half a second) but kept the interval consistent. You might have noticed the @Backoff annotation in the @Retryable declaration. This annotation hints at the ability to customize the backoff algorithm. You could implement any linear or nonlinear backoff policy of your choice, but Spring Retry already has some common ones built in. You already saw an example of the former, when you waited half a second between retry attempts. Let me now show you a nonlinear example in action. To start, if you haven’t already done so, please check out the following tag from your Git repo:

git checkout requestretries/0.0.5

The app deployment manifests describe a smaller deployment of our software, suitable for deployment into your Minikube cluster. You can deploy the appropriate version of the sample by executing the deployApps.sh bash script. Although you can certainly send curl commands against the Connections’ Posts service at this point, what I want to focus on here is the behavior of the Kubernetes replication controller—the control loop that’s watching and maintaining the state of your app deployments.

Have a look at the output of the kubectl get pods command. I’d like you to watch it continuously, so type the command watch kubectl get pods. You’ll see something like this:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

connection-posts-67d8db4c7b-tscf8 1/1 Running 0 10h

connections-748dc47cc6-7bzzr 1/1 Running 0 10h

mysql-64bd6d89d8-ggwss 1/1 Running 0 1d

posts-649d88dff-kmmx8 1/1 Running 0 9h

redis-846b8c56fb-8k8f7 1/1 Running 0 1d

sccs-84cc988f57-fjhzx 1/1 Running 0 1d

It’s a fresh installation with one instance each of our sample microservices. What I’d like you to do now is infect the Posts service; you’ll make it unhealthy. You can do this by issuing the following command:

curl -i -X POST $(minikube service --url posts-svc)/infect

And now just watch the get pods output. After approximately 15–30 seconds, you’ll see the Posts service getting restarted. If it goes fast, you may notice only that the counter in the RESTARTS column is incremented:

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

connection-posts-67d8db4c7b-tscf8 1/1 Running 0 10h

connections-748dc47cc6-7bzzr 1/1 Running 0 10h

mysql-64bd6d89d8-ggwss 1/1 Running 0 1d

posts-649d88dff-kmmx8 1/1 Running 1 9h

redis-846b8c56fb-8k8f7 1/1 Running 0 1d

sccs-84cc988f57-fjhzx 1/1 Running 0 1d

Now infect that instance again by issuing the same curl command. After approximately 15–30 seconds, you’ll see the app restart again. Keep re-infecting it every time it comes back. After you’ve done this four or five times, instead of the app restarting, the status will read CrashLoopBackOff. The replication controller has noticed that the app has repeatedly become unhealthy and will wait a bit longer before attempting another restart. It has implemented a nonlinear backoff policy.

$ kubectl get pods

NAME READY STATUS RESTARTS AGE

connection-posts-67d8db4c7b-tscf8 1/1 Running 0 10h

connections-748dc47cc6-7bzzr 1/1 Running 0 10h

mysql-64bd6d89d8-ggwss 1/1 Running 0 1d

posts-649d88dff-kmmx8 1/1 CrashLoopBackOff 5 9h

redis-846b8c56fb-8k8f7 1/1 Running 0 1d

sccs-84cc988f57-fjhzx 1/1 Running 0 1d

What I hope you take away from this dialogue, and the chapter as a whole, is the need to start looking for the control loops in your cloud-native software designs. Although for many of us an imperative style of programming feels natural, in a distributed system problems abound. They may not show themselves initially, but lurking beneath seemingly sound implementations are hairy edge cases and implementations that fail when unexpected changes occur. An eventually consistent, control-loop-driven software design will fare far better in the distributed systems that make up our cloud-native software.

**Summary**

* Retrying requests that time out can absorb errors that otherwise would have propagated through the system.
* If not done right, queued retry requests can overload a system even after connectivity problems are corrected.
* Properly configured retries can dramatically reduce the risk of these retry storms, while still providing significant benefits in less-dramatic outages.
* It’s your responsibility as a developer to use retries only when it’s *safe* to do so.
* You should make a habit of implementing not only the core flow of your service, but also the fallback logic for when the happy path is failing.
* Retries are but one example of a control-loop pattern.
* Control loops are an essential technique for the distributed systems that make up cloud-native software.