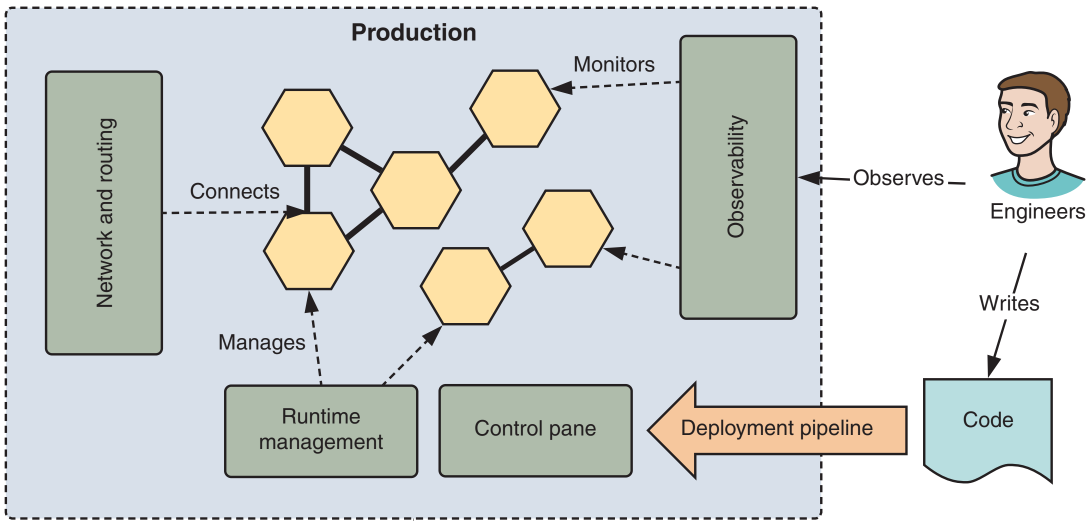
Microservices IN ACTION

Morgan Bruce
Paulo A. Pereira

Sample Chapter

MANNING

A microservice production environment



A microservice production environment has several components: a deployment target, a deployment pipeline, runtime management, networking features, and support for observability. In this book, we'll teach you about these components and how you can use them to build a stable, modern microservice application.

Microservices in Action

by Morgan Bruce
Paulo A. Pereira

Chapter 6

brief contents

6 Designing reliable services

This chapter covers

- The impact of service availability on application reliability
- Designing microservices that defend against faults in their dependencies
- Applying retries, rate limits, circuit breakers, health checks, and caching to mitigate interservice communication issues
- Applying safe communication standards across many services

No microservice is an island; each one plays a small part in a much larger system. Most services that you build will have other services that rely on them—upstream collaborators—and in turn themselves will depend on other services—downstream collaborators—to perform useful functions. For a service to reliably and consistently perform its job, it needs to be able to trust these collaborators.

This is easier said than done. Failures are inevitable in any complex system. An individual microservice might fail for a variety of reasons. Bugs can be introduced into code. Deployments can be unstable. Underlying infrastructure might let you down: resources might be saturated by load; underlying nodes might become unhealthy; even entire data centers can fail. As we discussed in chapter 5, you can't even trust that the network between your services is reliable—believing otherwise is a well-known fallacy of distributed computing.[1] Lastly, human error can lead to major failures. For example, I'm writing this chapter a week after an engineer's mistake in running a maintenance script led to a severe outage in Amazon S3, affecting thousands of well-known websites.

It's impossible to eliminate failure in microservice applications—the cost of that would be infinite! Instead, your focus needs to be on designing microservices that are tolerant of dependency failures and able to gracefully recover from them or mitigate the impact of those failures on their own responsibilities.

In this chapter, we'll introduce the concept of service availability, discuss the impact of failure in microservice applications, and explore approaches to designing reliable communication between services. We'll also discuss two different tactics—frameworks and proxies—for ensuring all microservices in an application interact safely. Using these techniques will help you maximize the reliability of your microservice application—and keep your users happy.

6.1 Defining reliability

Let's start by figuring out how to measure the reliability of a microservice. Consider a simple microservice system: a service, *holdings*, calls two dependencies, *transactions* and *market-data*. Those services in turn call further dependencies. Figure 6.1 illustrates this relationship.

For any of those services, you can assume that they spent some time performing work successfully. This is known as *uptime*. Likewise, you can safely assume—because failure is inevitable—that they spent some time failing to complete work. This is known as *downtime*. You can use uptime and downtime to calculate *availability*: the percentage of operational time during which the service was working correctly. A service's availability is a measure of how reliable you can expect it to be.

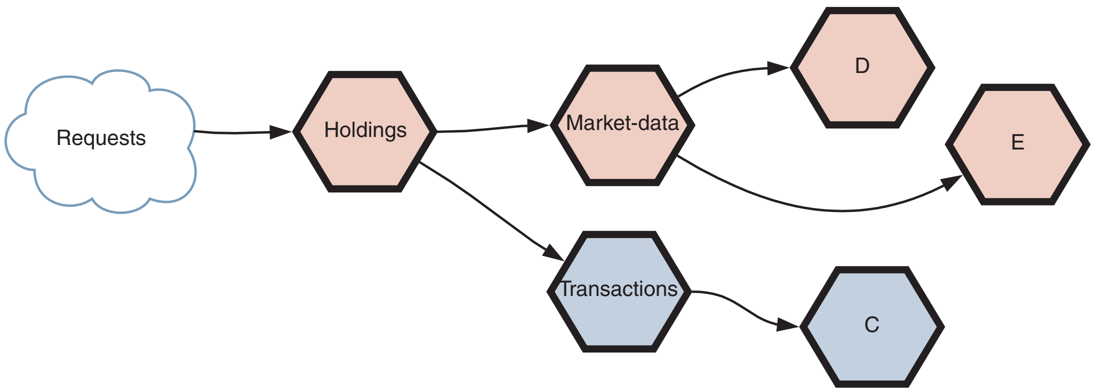


Figure 6.1 A simple microservice system, illustrating dependencies between collaborating services

[1] Peter Deutsch originally posited the eight fallacies of distributed computing in 1994. A good overview is available here: http://mng.bz/9T5F.

A typical shorthand for high availability is “nines:” for example, two nines is 99%, whereas five nines is 99.999%. It’d be highly unusual for critical production-facing services to be less reliable than this.

To illustrate how availability works, imagine that calls from holdings to market-data are successful 99.9% of the time. This might sound quite reliable, but downtime of 0.1% quickly becomes pronounced as volumes increase: only one failure per 1,000 requests, but 1,000 failures per million. These failures will directly affect your calling service unless you can design that service to mitigate the impact of dependency failure.

Microservice dependency chains can quickly become complex. If those dependencies can fail, what’s the probability of failure within your whole system? You can treat your availability figure as the probability of a request being successful—by multiplying together the availability figures for the parts of the chain, you can estimate the failure rate across your entire system.

Say you expand the previous example to specify that you have six services with the same success rate for calls. For any request to your system, you can expect one of four outcomes: all services work correctly, one service fails, multiple services fail, or all services fail.

Because calls to each microservice are successful 99.9% of the time, combined reliability of the system will be $0.999^6 = 0.994 = 99.4\%$. Although this is a simple model, you can see that the whole application will always be less reliable than its independent components; the maximum availability you can achieve is a product of the availability of a service’s dependencies.

To illustrate, imagine that service D’s availability is degraded to 95%. Although this won’t affect transactions—because it’s not part of that call hierarchy—it will reduce the reliability of both market-data and holdings. Figure 6.2 illustrates this impact.

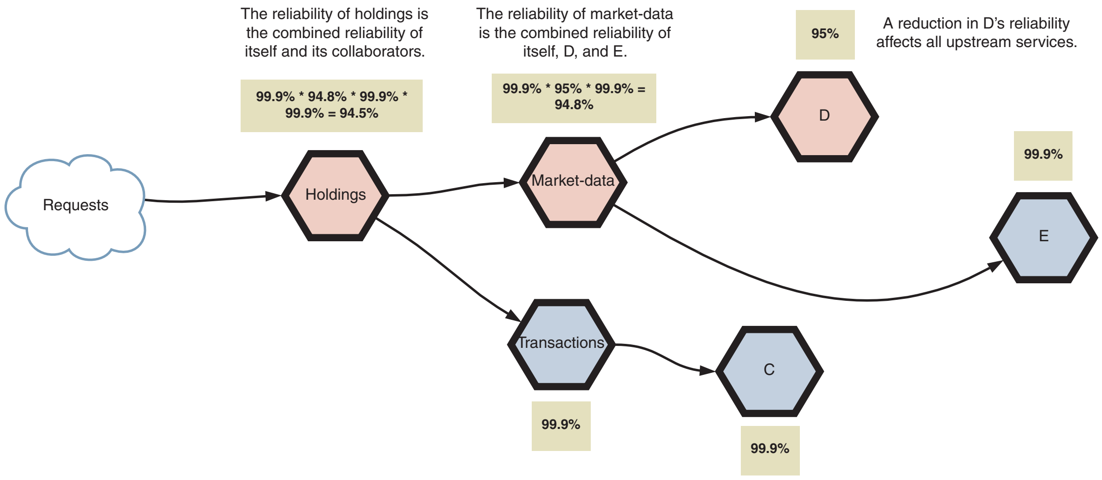

The reliability of holdings is the combined reliability of itself and its collaborators.
99.9% * 94.8% * 99.9% * 99.9% = 94.5%
The reliability of market-data is the combined reliability of itself, D, and E.
99.9% * 95% * 99.9% = 94.8%
95%
A reduction in D’s reliability affects all upstream services.
D
99.9%
Requests
Holdings
Market-data
E
Transactions
C
99.9%
99.9%

Figure 6.2 The impact of service dependency availability on reliability in a microservice application

It’s crucial to maximize service availability—or isolate the impact of unreliability—to ensure the availability of your entire application. Measuring availability won’t tell you how to make your services reliable, but it gives you a target to aim for or, more specifically, a goal to guide both the development of services and the expectations of consuming services and engineers.

NOTE How do you monitor availability? We’ll explore approaches to monitoring service availability in a microservice application in part 4 of this book.

If you can’t trust the network, your hardware, other services, or even your own services to be 100% reliable, how can you maximize availability? You need to design defensively to meet three goals:

- Reduce the incidence of avoidable failures
- Limit the cascading and system-wide impact of unpredictable failures
- Recover quickly—and ideally automatically—when failures do occur

Achieving these goals will ultimately maximize the uptime and availability of your services.

6.2 *What could go wrong?*

As we’ve stated, failure is inevitable in a complex system. Over the lifetime of an application, it’s incredibly likely that any catastrophe that could happen, will happen. Consequently, you need to understand the different types of failures that your application might be susceptible to. Understanding the nature of these risks and their likelihood is fundamental to both architecting appropriate mitigation strategies and reacting rapidly when incidents do occur.

Balancing risk and cost

It’s important to be pragmatic: you can neither anticipate nor eliminate every possible cause of failure. When you’re designing for resilience, you need to balance the risk of a failure against what you can reasonably defend against given time and cost constraints:

- The cost to design, build, deploy, and operate a defensive solution
- The nature of your business and expectations of your customers

To put that in perspective, consider the S3 outage I mentioned earlier. You could defend against that error by replicating data across multiple regions in AWS or across multiple clouds. But given that S3 failures of that magnitude are exceptionally rare, that solution wouldn’t make economic sense for many organizations because it would significantly increase operational costs and complexity.

As a responsible service designer, you need to identify possible types of failure within your microservice application, rank them by anticipated frequency and impact, and decide how you’ll mitigate their impact. In this section, we’ll walk you through some common failure scenarios in microservice applications and how they arise. We'll also explore cascading failures—a common catastrophic scenario in a distributed system.

6.2.1 Sources of failure

Let's examine a microservice to understand where failure might arise, using one of SimpleBank's services as an example. You can assume a few things about the market-data service:

- The service will run on hardware—likely a virtualized host—that ultimately depends on a physical data center.
- Other upstream services depend on the capabilities of this service.
- This service stores data in some store—for example, a SQL database.
- It retrieves data from third-party data sources through APIs and file uploads.
- It may call other downstream SimpleBank microservices.

Figure 6.3 illustrates the service and its relationship to other components.

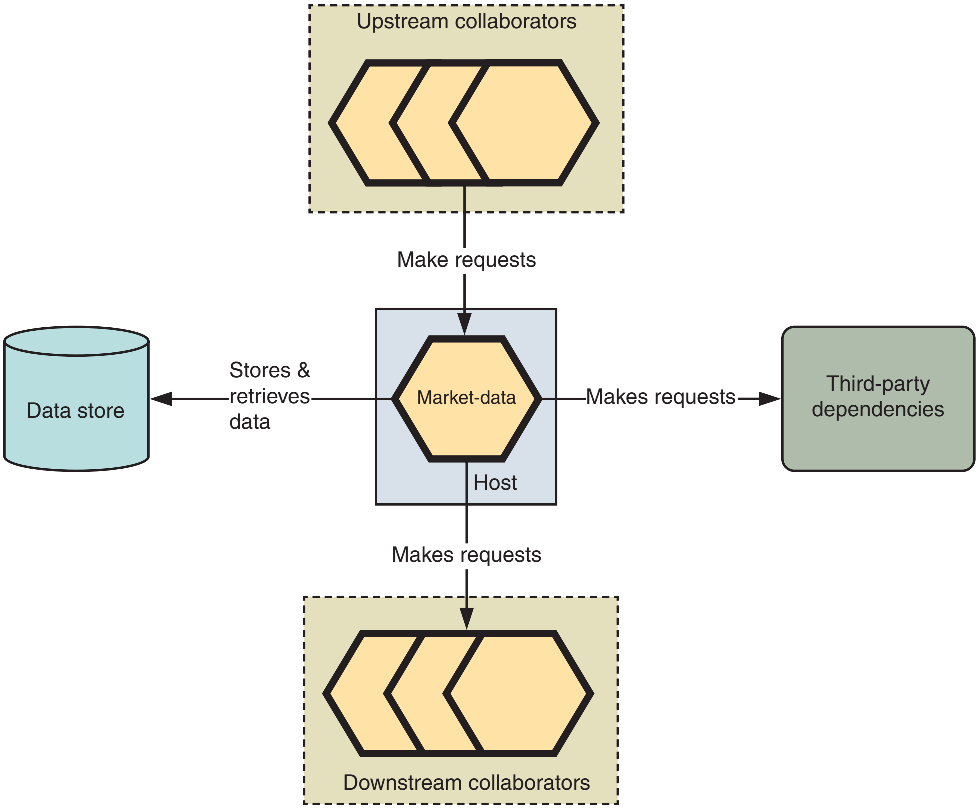


Figure 6.3 Relationships between the market-data microservice and other components of the application

Every point of interaction between your service and another component indicates a possible area of failure. Failures could occur in four major areas:

- *Hardware*—The underlying physical and virtual infrastructure on which a service operates
- *Communication*—Collaboration between different services and/or third parties
- *Dependencies*—Failure within dependencies of a service
- *Internal*—Errors within the code of the service itself, such as defects introduced by engineers

Let's explore each category in turn.

Hardware

Regardless of whether you run your services in a public cloud, on-premise, or using a PaaS, the reliability of the services will ultimately depend on the physical and virtual infrastructure that underpins them, whether that's server racks, virtual machines, operating systems, or physical networks. Table 6.1 illustrates some of the causes of failure within the hardware layer of a microservice application.

Table 6.1 Sources of failure within the hardware layer of a microservice application

Source of failure	Frequency	Description
Host	Often	Individual hosts (physical or virtual) may fail.
Data center	Rare	Data centers or components within them may fail.
Host configuration	Occasionally	Hosts may be misconfigured— for example, through errors in provisioning tools.
Physical network	Rare	Physical networking (within or between data centers) may fail.
Operating system and resource isolation	Occasionally	The OS or the isolation system—for example, Docker—may fail to operate correctly.

The range of possible failures at this layer of your application are diverse and unfortunately, often the most catastrophic because hardware component failure may affect the operation of multiple services within an organization.

Typically, you can mitigate the impact of most hardware failures by designing appropriate levels of redundancy into a system. For example, if you're deploying an application in a public cloud, such as AWS, you'd typically spread replicas of a service across multiple zones—geographically distinct data centers within a wider region—to reduce the impact of failure within a single center.

It's important to note that hardware redundancy can incur additional operational cost. Some solutions may be complex to architect and run—or just plain expensive. Choosing the right level of redundancy for an application requires careful consideration of the frequency and impact of failure versus the cost of mitigating against potentially rare events.

Communication

Communication between services can fail: network, DNS, messaging, and firewalls are all possible sources of failure. Table 6.2 details possible communication failures.

Table 6.2 Sources of communication failure within a microservice application

Source of failure	Description
Network	Network connectivity may not be possible.
Firewall	Configuration management can set security rules inappropriately.
DNS errors	Hostnames may not be correctly propagated or resolved across an application.
Messaging	Messaging systems—for example, RPC—can fail.
Inadequate health checks	Health checks may not adequately represent instance state, causing requests to be routed to broken instances.

Communication failures can affect both internal and external network calls. For example, connectivity between the market-data service and the external APIs it relies on could degrade, leading to failure.

Network and DNS failures are reasonably common, whether caused by changes in firewall rules, IP address assignment, or DNS hostname propagation in a system. Network issues can be challenging to mitigate, but because they're often caused by human intervention (whether through service releases or configuration changes), the best way to avoid many of them is to ensure that you test configuration changes robustly, and that they're easy to roll back if issues occur.

Dependencies

Failure can occur in other services that a microservice depends on, or within that microservice's internal dependencies (such as databases). For example, the database that market-data relies on to save and retrieve data might fail because of underlying hardware failure or hitting scalability limits—it wouldn't be unheard of for a database to run out of disk space!

As we outlined earlier, such failures have a drastic effect on overall system availability. Table 6.3 outlines possible sources of failure.

Table 6.3 Sources of dependency-related failure

Source of failure	Description
Timeouts	Requests to services may time out, resulting in erroneous behavior.
Decommissioned or nonbackwards-compatible functionality	Design doesn't take service dependencies into account, unexpectedly changing or removing functionality.
Internal component failures	Problems with databases or caches prevent services from working correctly.
External dependencies	Services may have dependencies outside of the application that don't work correctly or as expected—for example, third-party APIs.

In addition to operational sources of failure, such as timeouts and service outages, dependencies are prone to errors caused by design and build failures. For example, a service may rely on an endpoint in another service that's changed in a nonbackwards-compatible way or, even worse, removed completely without appropriate decommissioning.

SERVICE PRACTICES

Lastly, inadequate or limited engineering practices when developing and deploying services may lead to failure in production. Services may be poorly designed, inadequately tested, or deployed incorrectly. You may not catch errors in testing, or a team may not adequately monitor the behavior of their service in production. A service might scale ineffectively: hitting memory, disk, or CPU limits on its provisioned hardware such that performance is degraded—or the service becomes completely unresponsive.

Because each service contributes to the effectiveness of the whole system, one poor quality service can have a detrimental effect on the availability of swathes of functionality. Hopefully the practices we outline throughout this book will help you avoid this—unfortunately common—source of failure!

6.2.2 Cascading failures

You should now understand how failure in different areas can affect a single microservice. But the impact of failure doesn't stop there. Because your applications are composed of multiple microservices that continually interact with each other, failure in one service can spread across an entire system.

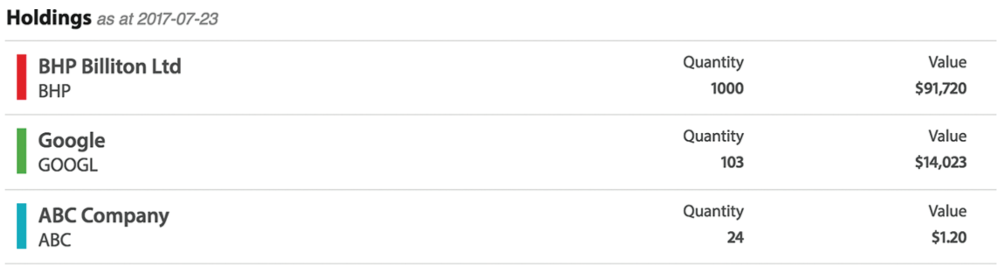
Cascading failures are a common mode of failure in distributed applications. A cascading failure is an example of *positive feedback*: an event disturbs a system, leading to some effect, which in turn increases the magnitude of the initial disturbance. In this case, positive means that the effect increases—not that the outcome is beneficial.

You can observe this phenomenon in several real-world domains, such as financial markets, biological processes, or nuclear power stations. Consider a stampede in a herd of animals: panic causes an animal to run, which in turn spreads panic to other animals, which causes them to run, and so on. In a microservice application, overload can cause a domino effect: failure in one service increases failure in upstream services, and in turn their upstream services. At worst, the result is widespread unavailability.

Let's work through an example to illustrate how overload can result in a cascading failure. Imagine that SimpleBank built a UI to show a user their current financial holdings (or positions) in an account. That might look something like figure 6.4.

Each financial position is the sum of the transactions—purchases and sales of a stock—made to date, multiplied by the current price. Retrieving these values relies on collaboration between three services:

- *Market-data*—A service responsible for retrieving and processing price and market information for financial instruments, such as stocks
- *Transactions*—A service responsible for representing transactions occurring within an account
- *Holdings*—A service responsible for aggregating transactions and market-data to report financial positions



Holdings *as at 2017-07-23*

🟥 **BHP Billiton Ltd** BHP	Quantity 1000	Value $91,720
🟩 **Google** GOOGL	Quantity 103	Value $14,023
🟦 **ABC Company** ABC	Quantity 24	Value $1.20

Figure 6.4 A user interface that reports financial holdings in an account

Figure 6.5 outlines the production configuration of these services. For each service, load is balanced across multiple replicas.

Suppose that holdings are being retrieved 1,000 times per second (QPS). If you have two replicas of your holdings service, each replica will receive 500 QPS (figure 6.6).

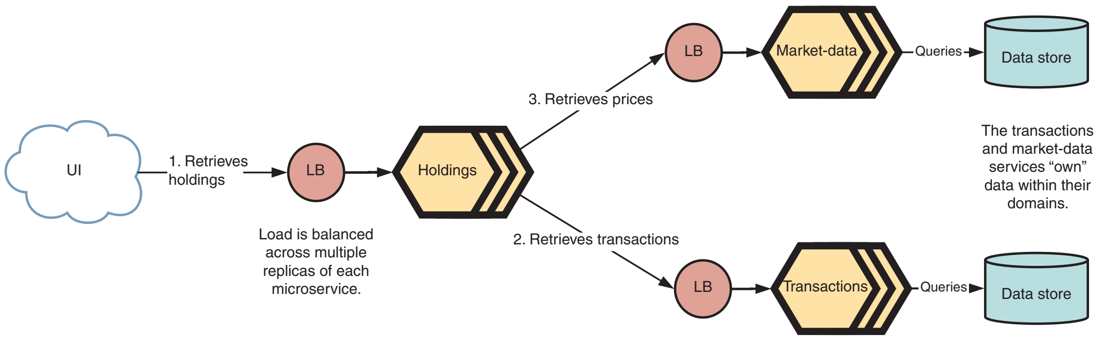

UI
1. Retrieves holdings
LB
Holdings
Load is balanced across multiple replicas of each microservice.
3. Retrieves prices
LB
Market-data
Queries
Data store
The transactions and market-data services "own" data within their domains.
2. Retrieves transactions
LB
Transactions
Queries
Data store

Figure 6.5 Production configuration and collaboration between services to populate the "current financial holdings" user interface

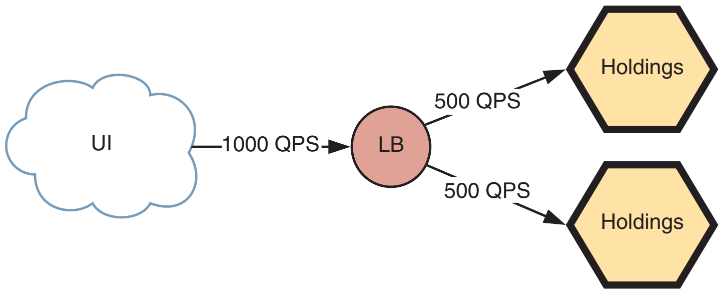


Figure 6.6 Queries made to a service are split across multiple replicas.

Your holdings service subsequently queries transactions and market-data to construct the response. Each call to holdings will generate two calls: one to transactions and one to market-data.

Now, let's say a failure occurs that takes down one of your transactions replicas. Your load balancer reroutes that load to the remaining replica, which now needs to service 1,000 QPS (figure 6.7).

But that reduced capacity is unable to handle the level of demand to your service. Depending on how you've deployed your service—the characteristics of your web server—the change in load might first lead to increased latency as requests are queued. In turn, increased latency might start exceeding the maximum wait time that the holdings service expects for that query. Alternatively, the transactions service may begin dropping requests.

It's not unreasonable for a service to retry a request to a collaborator when it fails. Now, imagine that the holdings service will retry any request to transactions that times out or fails. This will further increase the load on your remaining transactions resource, which now needs to handle both the regular request volume and the heightened retry volume (figure 6.8). In turn, the holdings service takes longer to respond while it waits on its collaborator.

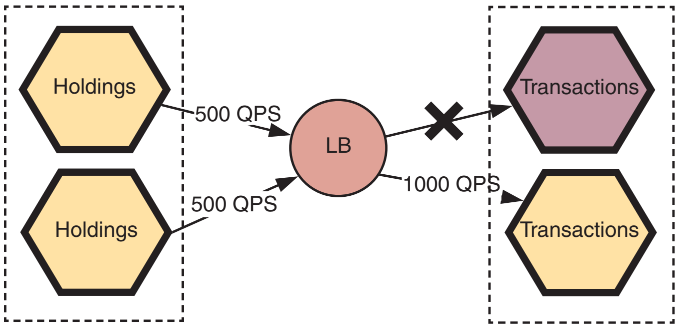


Figure 6.7 One replica of a collaborating service fails, sending all load to the remaining instance.

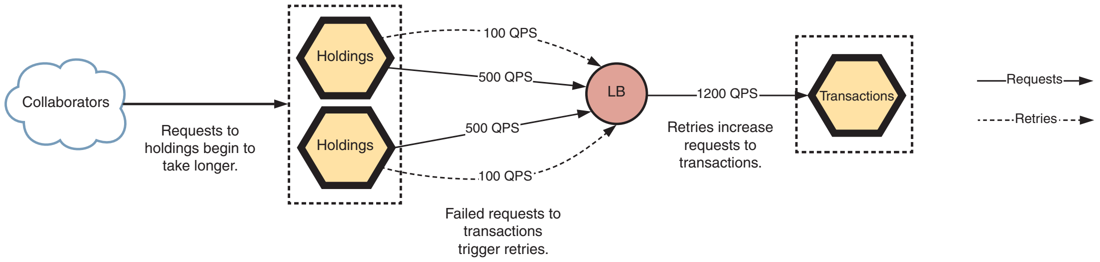

Collaborators
Requests to holdings begin to take longer.
Holdings
Holdings
100 QPS
500 QPS
500 QPS
100 QPS
LB
1200 QPS
Transactions
Retries increase requests to transactions.
Failed requests to transactions trigger retries.
Requests
Retries

Figure 6.8 Overload on transactions causes some requests to fail, in turn causing holdings to retry those requests, which starts to degrade holdings' response time.

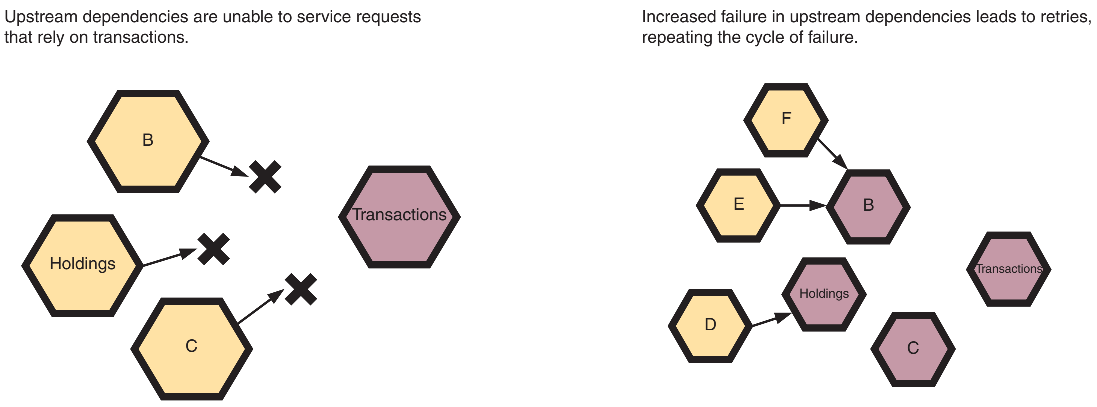

Upstream dependencies are unable to service requests that rely on transactions.
B
Holdings
C
Transactions
Increased failure in upstream dependencies leads to retries, repeating the cycle of failure.
F
E
B
D
Holdings
C
Transactions

Figure 6.9 Overload in a service leads to complete failure. Unhealthy retry behavior is repeated across dependency chains as service performance progressively degrades, leading to further overloads.

This feedback loop—failed requests lead to a higher volume of subsequent requests, leading to a higher rate of failure—continues to escalate. Your whole system is unable to complete work, as other services that rely on transactions or holdings begin to fail. Your initial failure in a single service causes a domino effect, worsening response times and availability across several services. At worst, the cumulative impact on the transactions service causes it to fail completely. Figure 6.9 illustrates this final stage of a cascading failure.

Cascading failures aren't only caused by overload—although this is one of the most common root causes. In general, increased error rates or slower response times can lead to unhealthy service behavior, increasing the chance of failure across multiple services that depend on each other.

You can use several approaches to limit the occurrence of cascading failures in microservice applications: circuit breakers; fallbacks; load testing and capacity planning; back-off and retries; and appropriate timeouts. We'll explore these approaches in the next section.

6.3 Designing reliable communication

Earlier, we emphasized the importance of collaboration in a microservice application. Dependency chains of multiple microservices will achieve most useful capabilities in an application. When one microservice fails, how does that impact its collaborators and ultimately, the application's end customers?

If failure is inevitable, you need to design and build your services to maximize availability, correct operation, and rapid recovery when failure does occur. This is fundamental to achieving resiliency. In this section, we'll explore several techniques for ensuring that services behave appropriately—maximizing correct operation—when their collaborators are unavailable:

- Retries
- Fallbacks, caching, and graceful degradation
- Timeouts and deadlines

- Circuit breakers
- Communication brokers

Before we start, let's get a simple service running that we can use to illustrate the concepts in this section. You can find these examples in the book's Github repository (http://mng.bz/7eN9). Clone the repository to your computer and open the chapter-6 directory. This directory contains some basic services—holdings and market-data—which you'll run inside Docker containers (figure 6.10). The holdings service exposes a `GET /holdings` endpoint, which makes a JSON API request to retrieve price information from market-data.

To run these, you'll need `docker-compose` installed (directions online: https://docs.docker.com/compose/install/). If you're ready to go, type the following at the command line:

```
$ docker-compose up
```

This will build Docker images for each service and start them as isolated containers on your machine. Now let's dive in!

6.3.1 Retries

In this section, we'll explore how to use retries when failed requests occur. To understand these techniques, let's start by examining communication from the perspective of your upstream service, holdings.

Imagine that a call from the holdings service to retrieve prices fails, returning an error. From the perspective of the calling service, it's not clear yet whether this failure is isolated—repeating that call is likely to succeed, or systemic—the next call has a high likelihood of failing. You expect calls to retrieve data to be *idempotent*—to have no effect on the state of the target system and therefore be repeatable.[2]

As a result, your first instinct might be to retry the request. In Python, you can use an open source library—`tenacity`—to decorate the appropriate method of your API client (the `MarketDataClient` class in holdings/clients.py) and perform retries if the method throws an exception. The following listing shows the class with retry code added.

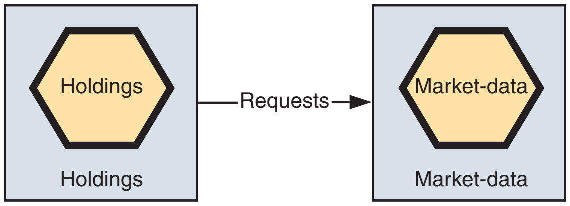


Figure 6.10 Docker containers for working with microservice requests

[2] Requests that effect some system change aren't typically idempotent. One strategy for guaranteeing "exactly once" semantics is to implement idempotency keys. See Brandur Leach, "Designing robust and predictable APIs with idempotency," February 22, 2017, https://stripe.com/blog/idempotency.

Listing 6.1 Adding a retry to a service call

```
import requests
import logging
from tenacity import retry, stop, before

class MarketDataClient(object):

    logger = logging.getLogger(__name__)
    base_url = 'http://market-data:8000'

    def _make_request(self, url):
        response = requests.get(f"{self.base_url}/{url}",
                                headers={'content-type': 'application/json'})
        return response.json()

    @retry(stop=stop_after_attempt(3),
           before=before_log(logger, logging.DEBUG))
    def all_prices(self):
        return self._make_request("prices")
```

Imports relevant functions from the library

Retries the query up to three times

Logs each retry before execution

Let's call the holdings service to see how it behaves. In another terminal window, make the following request:

```
curl -I http://{DOCKER_HOST}/holdings
```

This will return a 500 error, but if you follow the logs from the market-data service, you can see a request being made to `GET /prices` three times, before the holdings service gives up.

If you read the previous section, you should be wary at this point. Failure might be isolated or persistent, but the holdings service can't know which one is the case based on one call.

If the failure is isolated and transient, then a retry is a reasonable option. This helps to minimize direct impact to end users—and explicit intervention from operational staff—when abnormal behavior occurs. It's important to consider your budget for retries: if each retry takes a certain number of milliseconds, then the consuming service can only absorb so many retries before it surpasses a reasonable response time.

But if the failure is persistent—for example, if the capacity of market-data is reduced—then subsequent calls may worsen the issue and further destabilize the system. Suppose you retry each failed request to market-data five times. Every failed request you make to this service potentially results in another five requests; the volume of retries continues to grow. The entire service is doing less useful work as it attempts to service a high volume of retries. At worst, retries suffocate your market-data service, magnifying your original failure. Figure 6.11 illustrates this growth of requests.

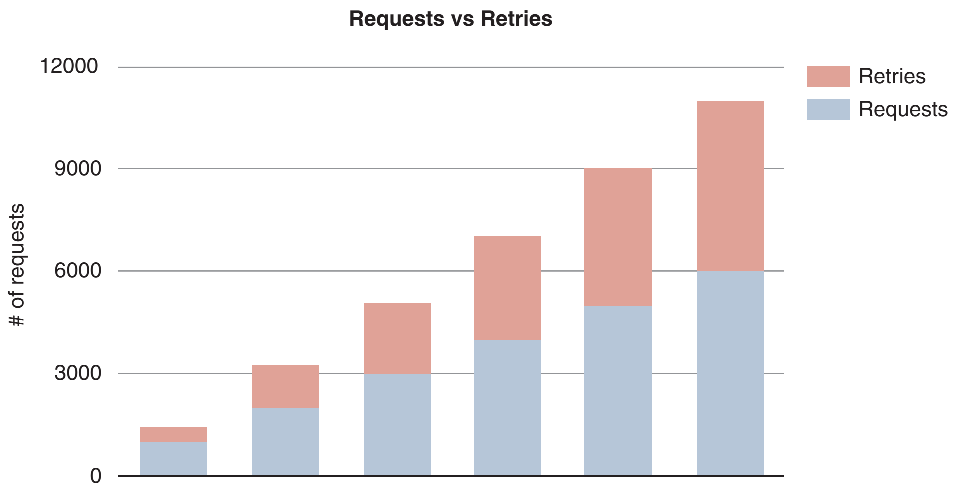


Figure 6.11 Growth of load on your unstable market-data service resulting from failed requests being retried

How can you use retries to improve your resiliency in the face of intermittent failures without contributing to wider system failure if persistent failures occur? First, you could use a variable time between successive retries to try to spread them out evenly and reduce the frequency of retry-based load. This is known as an *exponential back-off* strategy and is intended to give a system under load time to recover. You can change the retry strategy you used earlier, as shown in the following listing. Afterwards, by curling the `/holdings` endpoint, you can observe the retry behavior of the service.

Listing 6.2 Changing your retry strategy to exponential back-off

```
@retry(wait=wait_exponential(multiplier=1, max=5),
       stop=stop_after_delay(5))
def all_prices(self):
    return self._make_request("prices")
```

Waits 2 ^ x * 1 second between each retry

Stops after five seconds

Unfortunately, exponential back-off can lead to another instance of curious emergent behavior. Imagine that a momentary failure interrupts several calls to market-data, leading to retries. Exponential back-off can cause the service to schedule those retries together so they further amplify themselves, like the ripples from throwing a stone in a pond.

Instead, back-off should include a random element—jitter—to spread out retries to a more constant rate and avoid thundering herds of synchronized retries.[3] The following listing shows how to adjust your strategy again.

[3] A great article by Marc Brooker about exponential back-off and the importance of jitter is available on the AWS Architecture Blog, March 4, 2015, http://mng.bz/TRk5.

Listing 6.3 Adding jitter to an exponential back-off

```
@retry(wait=wait_exponential(multiplier=1, max=5) + wait_random(0, 1),
       stop=stop_after_delay(5))
def all_prices(self):
    return self._make_request("prices")
```

Exponentially backs off, adding a random wait between zero and one second

Stops after five seconds

This strategy will ensure that retries are less likely to happen in synchronization across multiple waiting clients.

Retries are an effective strategy for tolerating intermittent dependency faults, but you need to use them carefully to avoid exacerbating the underlying issue or consuming unnecessary resources:

- Always limit the total number of retries.
- Use exponential back-off with jitter to smoothly distribute retry requests and avoid compounding load.
- Consider which error conditions should trigger a retry and, therefore, which retries are unlikely to, or will never, succeed.

When your service meets retry limits or can't retry a request, you can either accept failure or find an alternative way to serve the request. In the next section, we'll explore fallbacks.

6.3.2 Fallbacks

If a service's dependencies fail, you can explore four fallback options:

- Graceful degradation
- Caching
- Functional redundancy
- Stubbed data

Graceful Degradation

Let's return to the problem with the holdings service: if market-data fails, the application may not be able to provide valuations to end customers. To resolve this issue, you might be able to design an acceptable degradation of service. For example, you could show holding quantities without valuations. This limits the richness of your UI but is better than showing nothing—or an error. You can see techniques like this in other domains. For example, an e-commerce site could still allow purchases to be made, even if the site's order dispatch isn't functioning correctly.

Caching

Alternatively, you could cache the results of past queries for prices, reducing the need to query the market-data service at all. Say a price is valid for five minutes. If so, the holdings service could cache pricing data for up to five minutes, either locally or in a dedicated cache (for example, Memcached or Redis). This solution would both improve performance and provide contingency in the event of a temporary outage.

Let's try out this technique. You'll use a library called `cachetools`, which provides an implementation of a time-to-live cache. As you did earlier with retries, you'll decorate your client method, as shown in the following listing.

Listing 6.4 Adding in-process caching to a client call

```
import requests
import logging
from cachetools import cached, TTLCache

class MarketDataClient(object):

    logger = logging.getLogger(__name__)
    cache = TTLCache(maxsize=10, ttl=5*60)
    base_url = 'http://market-data:8000'

    def _make_request(self, url):
        response = requests.get(f"{self.base_url}/{url}",
                                headers={'content-type': 'application/json'})
        return response.json()

    @cached(cache)
    def all_prices(self):
        logger.debug("Making request to get all_prices")
        return self._make_request("prices")
```

Instantiates a cache

Decorates your method to store results using your cache

Subsequent calls made to `GET /holdings` should retrieve price information from the cache, rather than by making calls to market-data. If you used an external cache instead, multiple instances could use the cache, further reducing load on market-data and providing greater resiliency for all holdings replicas, albeit at the cost of maintaining an additional infrastructural component.

FUNCTIONAL REDUNDANCY

Similarly, you might be able to fall back to other services to achieve the same functionality. Imagine that you could purchase market data from multiple sources, each covering a different set of instruments at a different cost. If source A failed, you could instead make requests to source B (figure 6.12).

Functional redundancy within a system has many drivers: external integrations; algorithms for producing similar results with varying performance characteristics; and even older features that remain operational but have been superseded. In a globally distributed deployment, you could even fall back on services hosted in another region.[4]

Only some failure scenarios would allow the use of an alternative service. If the cause of failure was a code defect or resource overload in your original service, then rerouting to another service would make sense. But a general network failure could affect multiple services, including ones you might try rerouting to.

[4] At the ultimate end of this scale, Netflix can serve a given customer from any of their global data centers, conveying an impressive degree of resilience.

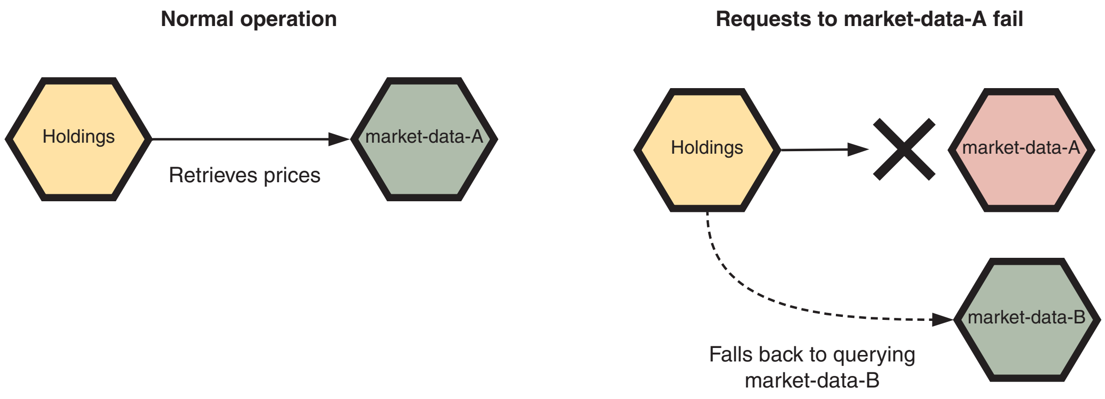


Figure 6.12 If service failure occurs, you may be able to serve the same capability with other services.

Stubbed data

Lastly, although it wouldn't be appropriate in this specific scenario, you could use stubbed data for fallbacks. Picture the "recommended to you" section on Amazon: if the backend was unable for some reason to retrieve those personalized recommendations, it'd be more graceful to fall back to a nonpersonalized data set than to show a blank section on the UI.

6.3.3 *Timeouts*

When the holdings service sends a request to market-data, that service consumes resources waiting for a reply. Setting an appropriate deadline for that interaction limits the time those resources are consumed.

You can set a timeout within your HTTP request function. For HTTP calls, you want to timeout if you haven't received any response, but not if the response itself is slow to download. Try the following listing to add a timeout.

Listing 6.5 Adding a timeout to an HTTP call

```
def _make_request(self, url):
    response = requests.get(f"{self.base_url}/{url}",
                            headers={'content-type': 'application/json'},
                            timeout=5)
    return response.json()
```

Sets a timeout of five seconds before receiving data from market-data

In a computational sense, network communication is slow, so the speed of failures is important. In a distributed system, some errors might happen almost instantly. For example, a dependent service may rapidly fail in the event of an internal bug. But many failures are slow. For example, a service that's overloaded by requests may respond sluggishly, in turn consuming the resources of the calling service while it waits for a response that may never come.

Slow failures illustrate the importance of setting appropriate deadlines—timing out in a reasonable timeframe—for communication between microservices. If you don't set upper bounds, it's easy for unresponsiveness to spread through entire microservice dependency chains. In fact, lack of deadlines can extend the impact of issues because a server consumes resources while it waits forever for an issue to be resolved.

Picking a deadline can be difficult. If they're too long, they can consume unnecessary resources for a calling service if a service is unresponsive. If they're too short, they can cause higher levels of failure for expensive requests. Figure 6.13 illustrates these constraints.

For many microservice applications, you set deadlines at the level of individual interactions; for example, a call from holdings to market-data may always have a deadline of 10 seconds. A more elegant approach is to apply an absolute deadline across an entire operation and propagate the remaining time across collaborators.

Without propagating deadlines, it can be difficult to make them consistent across a request. For example, holdings could waste resources waiting for market-data far beyond the overall deadline imposed by a higher level of the stack, such as an API gateway.

Imagine a chain of dependencies between multiple services. Each service takes a certain amount of time to do its work and expects its collaborators to take some time. If any of those times vary, static expectations may no longer be correct (figure 6.14).

If your service interactions are over HTTP, you could propagate deadlines using a custom HTTP header, such as `X-Deadline: 1000`, passing that value to set read timeout values on subsequent HTTP client calls. Many RPC frameworks, such as gRPC, explicitly implement mechanisms for propagating deadlines within a request context.

6.3.4 Circuit breakers

You can combine some of the techniques we've discussed so far. You can consider an interaction between holdings and market-data as analogous to an electrical circuit. In electrical wiring, circuit breakers perform a protective role—preventing spikes in current from damaging a wider system. Similarly, a circuit breaker is a pattern for pausing requests made to a failing service to prevent cascading failures.

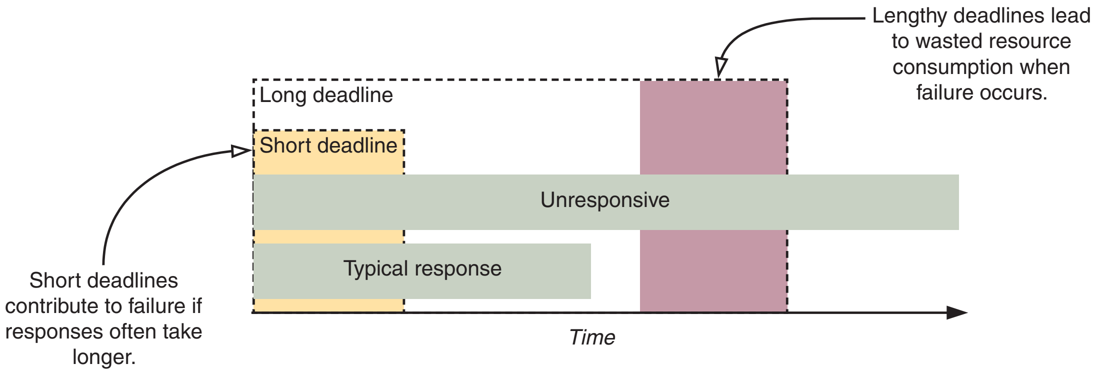

Lengthy deadlines lead to wasted resource consumption when failure occurs.
Long deadline
Short deadline
Unresponsive
Typical response
Short deadlines contribute to failure if responses often take longer.
Time

Figure 6.13 Choosing the right deadline requires balancing time constraints to maximize the window of successful requests.

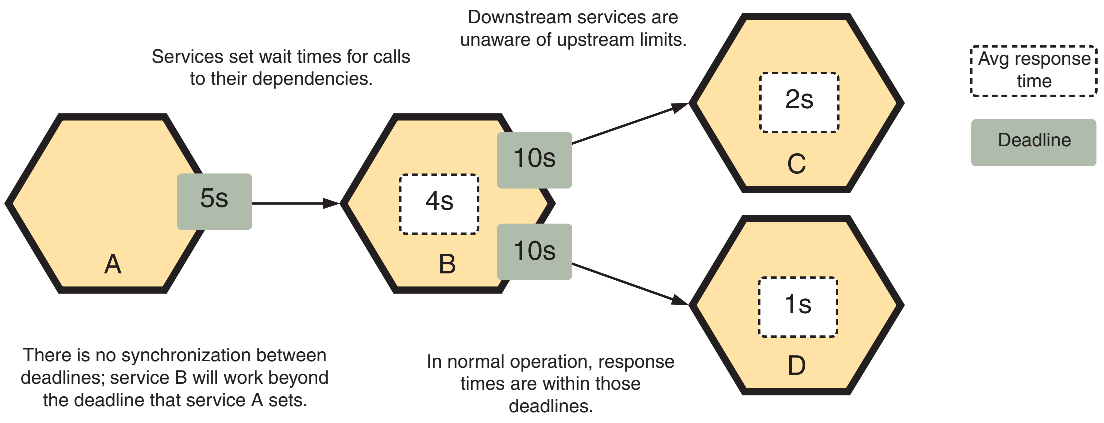

Services set wait times for calls to their dependencies.
Downstream services are unaware of upstream limits.
Avg response time
Deadline
A
5s
4s
B
10s
10s
2s
C
1s
D
There is no synchronization between deadlines; service B will work beyond the deadline that service A sets.
In normal operation, response times are within those deadlines.

Figure 6.14 Services may set expectations about how long they expect calls to collaborators to take; varying widely because of failure or latency can exacerbate the impact of those failures.

How does it work? Two principles, both of which we touched on in the previous section, inform the design of a circuit breaker:

1. Remote communication should fail quickly in the event of an issue, rather than wasting resources waiting for a response that might never come.
2. If a dependency is failing consistently, it's better to stop making further requests until that dependency recovers.

When making a request to a service, you can track the number of times that request succeeds or fails. You might track this number within each running instance of a service or share that state (using an external cache) across multiple services. In this normal operation, we consider the circuit to be closed.

If the number of failures seen or the rate of failures within a certain time window passes a threshold, then the circuit is opened. Rather than attempting to send requests to your collaborating service, you should short-circuit requests and, where possible, perform an appropriate fallback—returning a stubbed message, routing to a different service, or returning a cached response. Figure 6.15 illustrates the lifecycle of a request using a circuit breaker.

Setting the time window and threshold requires careful consideration of both the expected reliability of the target service and the volume of interactions between services. If requests are relatively sparse, then a circuit breaker may not be effective, because a large time window might be required to obtain a representative sample of requests. For service interactions with contrasting busy and quiet periods, you may want to introduce a minimum throughput to ensure a circuit only reacts when load is statistically significant.

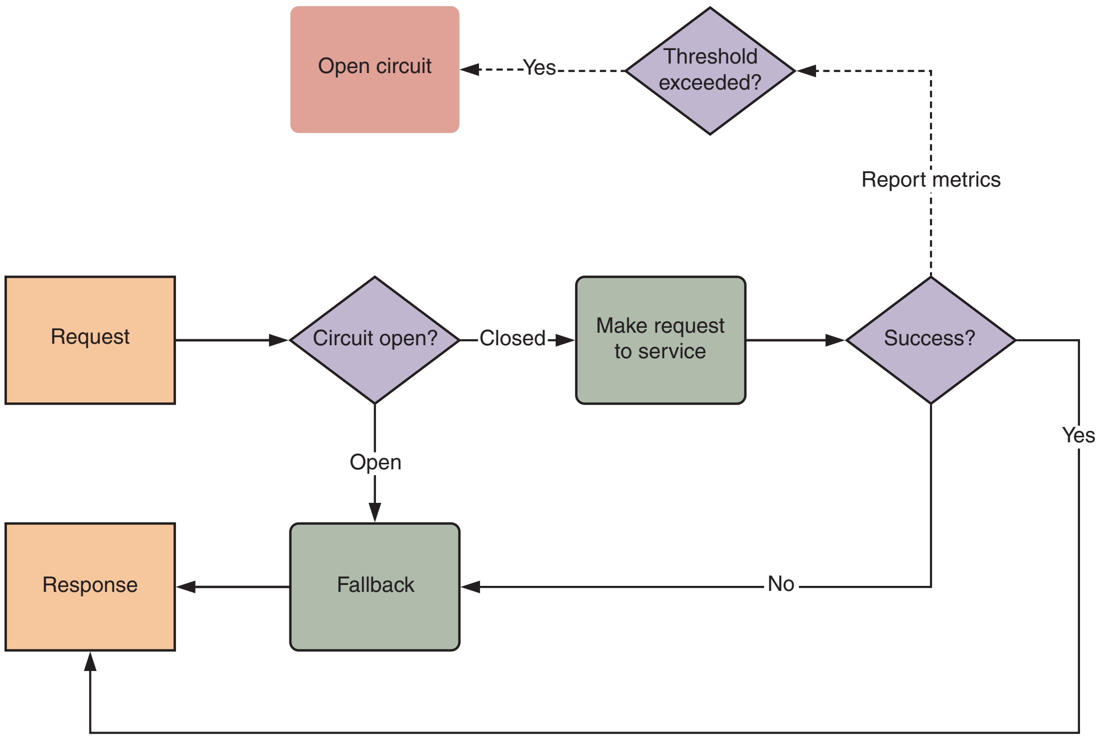


Figure 6.15 A circuit breaker controls the flow of requests between two services and opens when the number of failed requests surpasses a threshold.

NOTE You should monitor when circuits are opened and closed, as well as potentially alerting the team responsible, especially if the circuit is frequently opened. We'll discuss this further in part 4.

Once the circuit has opened, you probably don't want to leave it that way. When availability returns to normal, the circuit should be closed. The circuit breaker needs to send a trial request to determine whether the connection has returned to a healthy state. In this trial state, the circuit breaker is *half open*: if the call succeeds, the circuit will be closed; otherwise, it will remain open. As with other retries, you should schedule these attempts with an exponential back-off with jitter. Figure 6.16 shows the three distinct states of a circuit breaker.

Several libraries are available that provide implementations of the circuit breaker pattern in different languages, such as Hystrix (Java), CB2 (Ruby), or Polly (.NET).

TIP Don't forget that closed is the good state for a circuit breaker! The use of open and closed to represent, respectively, negative and positive states may seem counterintuitive but reflects the real-world behavior of an electrical circuit.

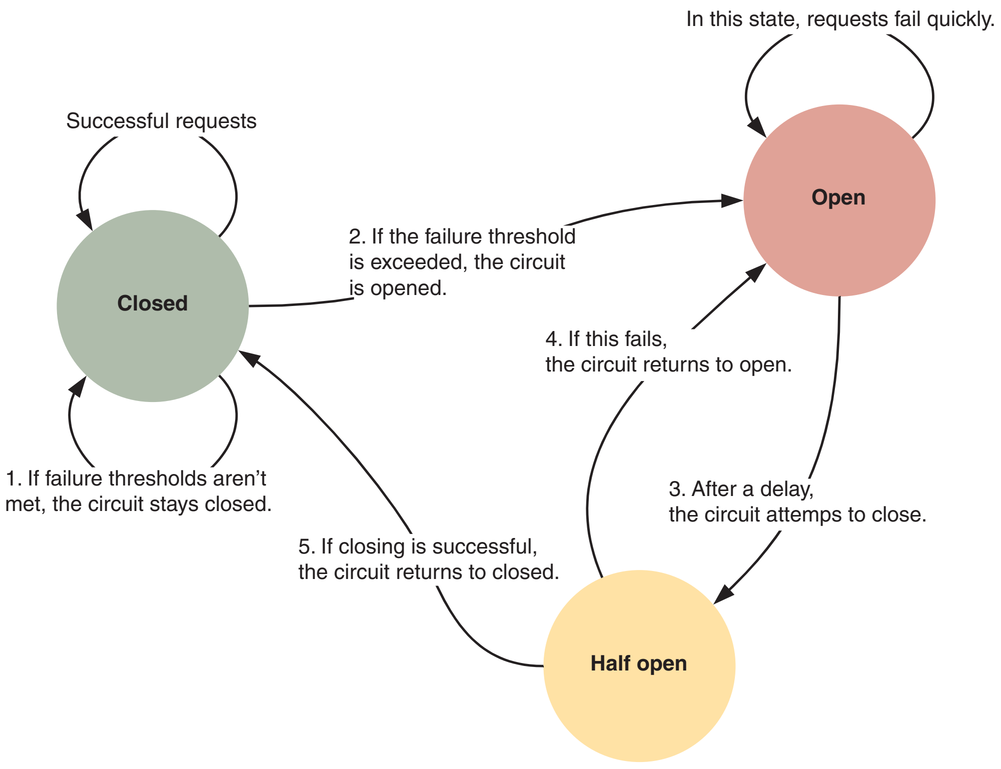

In this state, requests fail quickly.
Successful requests
Open
Closed
2. If the failure threshold is exceeded, the circuit is opened.
4. If this fails, the circuit returns to open.
1. If failure thresholds aren't met, the circuit stays closed.
3. After a delay, the circuit attemps to close.
5. If closing is successful, the circuit returns to closed.
Half open

Figure 6.16 A circuit breaker transitions between three stages: open, closed, and half open.

6.3.5 *Asynchronous communication*

So far, we've focused on failure in synchronous, point-to-point communication between services. As we outlined in the first section, the more services in a chain, the lower overall availability you can guarantee for that path.

Designing asynchronous service interactions, using a communication broker like a message queue, is another technique you can use to maximize reliability. Figure 6.17 illustrates this approach.

Where you don't need immediate, consistent responses, you can use this technique to reduce the number of direct service interactions, in turn increasing overall availability—albeit at the expense of making business logic more complex. As we mentioned elsewhere in this book, a communication broker becomes a single point of failure that will require careful attention for you to scale, monitor, and operate effectively.

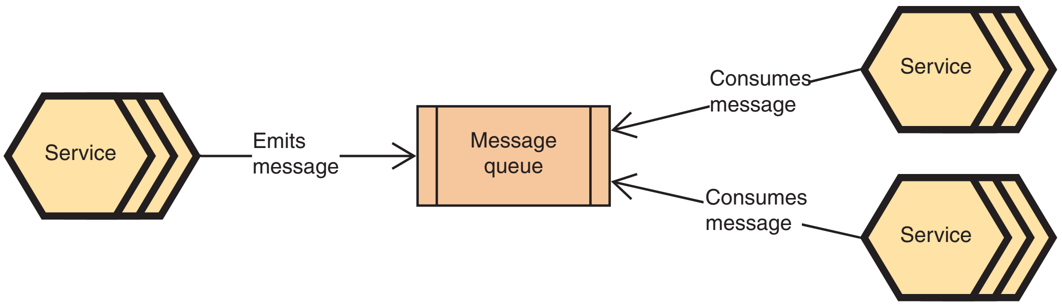


Figure 6.17 Using a message queue to decouple services from direct interaction

6.4 Maximizing service reliability

In the previous sections, we explored techniques to ensure a service can tolerate faults in interactions with its collaborators. Now, let's consider how you can maximize availability and fault tolerance within an individual service. In this section, we'll explore two techniques—health checks and rate limits—as well as methods for validating the resilience of services.

6.4.1 Load balancing and service health

In production, you deploy multiple instances of your market-data service to ensure redundancy and horizontal scalability. A load balancer will distribute requests from other services between these instances. In this scenario, the load balancer plays two roles:

1. Identifying which underlying instances are healthy and able to serve requests
2. Routing requests to different underlying instances of the service

A load balancer is responsible for executing or consuming the results of *health checks*. In the previous section, you could ascertain the health of a dependency at the point of interaction—when requests were being made. But that's not entirely adequate. You should have some way of understanding the application's readiness to serve requests at any time, rather than when it's actively being queried.

Every service you design and deploy should implement an appropriate health check. If a service instance becomes unhealthy, that instance should no longer receive traffic from other services. For synchronous RPC-facing services, a load balancer will typically query each instance's health check endpoint on an interval basis. Similarly, asynchronous services may use a heartbeat mechanism to test connectivity between the queue and consumers.

TIP It's often desirable for repeated or systematic instance failures, as detected by health checks, to trigger alerts to an operations team—a little human intervention can be helpful. We'll explore that further in part 4 of this book.

You can classify health checks based on two criteria: liveness and readiness. A liveness check is typically a simple check that the application has started and is running correctly. For example, an HTTP service should expose an endpoint—commonly `/health`, `/ping`, or `/heartbeat`—that returns a 200 OK response once the service is running (figure 6.18). If an instance is unresponsive, or returns an error message, the load balancer will no longer deliver requests there.

In contrast, a readiness check indicates whether a service is ready to serve traffic, because being alive may still not indicate that requests will be successful. A service might have many dependencies—databases, third-party services, configuration, caches—so you can use a readiness check to see if these constituent components are in the correct state to serve requests. Both of the example services implement a simple HTTP liveness check, as shown in the following listing.

Listing 6.6 Flask handler for an HTTP liveness check

```
@app.route('/ping', methods=["GET"])
def ping():
    return 'OK'
```

Health checks are binary: either an instance is available or it isn't. This works well with typical round-robin load balancing, where requests are distributed to each replica in turn. But in some circumstances the functioning of a service may be degraded and exhibit increased latency or error rates without a health check reflecting this status. As such, it can be beneficial to use load balancers that are aware of latency and able to route requests to instances that are performing better, or those that are under less load, to achieve more consistent service behavior. This is a typical feature of a microservice proxy, which we'll touch on later in this chapter.

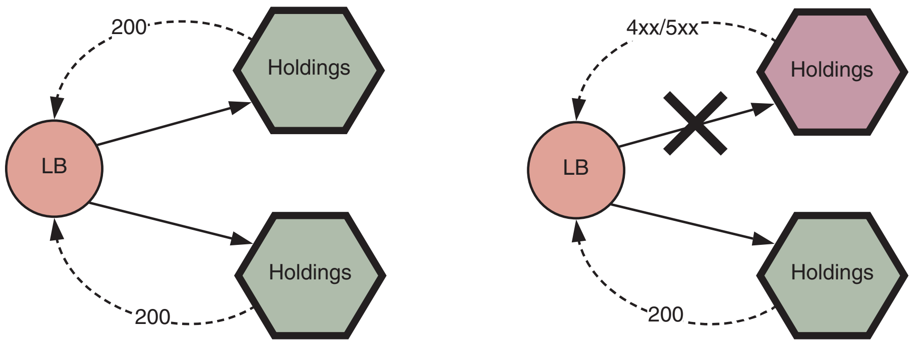


Figure 6.18 Load balancers continuously query service instances to check their health. If an instance is unhealthy, the load balancer will no longer route requests to that instance until it recovers.

6.4.2 *Rate limits*

Unhealthy service usage patterns can sometimes arise in large microservice applications. Upstream dependencies might make several calls, where a single batch call would be more appropriate, or available resources may not be distributed equitably among all callers. Similarly, a service with third-party dependencies could be limited by restrictions that those dependencies impose.

An appropriate solution is to explicitly limit the rate of requests or total requests available to collaborating services in a timeframe. This helps to ensure that a service—particularly when it has many collaborators—isn't overloaded. The limiting might be indiscriminate (drop all requests above a certain volume) or more sophisticated (drop requests from infrequent service clients, prioritize requests for critical endpoints, and drop low-priority requests). Table 6.4 outlines different rate-limiting strategies.

Table 6.4 Common rate-limiting strategies

Strategy	Description
Drop requests above volume	Drop consumer requests above a specified volume
Prioritize critical traffic	Drop requests to low-priority endpoints to prioritize resources for critical traffic
Drop uncommon clients	Prioritize frequent consumers of the service over infrequent users
Limit concurrent requests	Limit the overall number of requests an upstream service can make over a time period

Rate limits can be shared with a service's clients at design time or, better, at runtime. A service might return a header to a consumer that indicates the remaining volume of requests available. On receipt, the upstream collaborator should take this into account and adjust its rate of outbound requests. This technique is known as back pressure.

6.4.3 *Validating reliability and fault tolerance*

Applying the tactics and patterns we've covered will put you on a good path toward maximizing availability. But it's not enough to plan and design for resiliency: you need to validate that your services can tolerate faults and recover gracefully.

Thorough testing provides assurance that your chosen design is effective when both predicted and unpredictable failures occur. Testing requires the application of *load testing* and *chaos testing*. Although it's likely you're familiar with code testing—such as unit and acceptance testing to validate implementation, usually in a controlled environment—you might not know that load and chaos testing are intended to validate service limits by closely replicating the turbulence of production operation. Although testing isn't the primary focus of this book, it's useful to understand how these different testing techniques can help you build a robust microservice application.

Load testing

As a service developer, you can usually be confident that the number of requests made to your service will increase over time. When developing a service, you should

1. Model the expected growth and shape of service traffic to ensure that you understand the likely usage of your service
2. Estimate the capacity required to service that traffic
3. Validate the deployed capacity of the service by load testing against those limits
4. Use business and service metrics as appropriate to re-estimate capacity

Imagine you're considering how much capacity the market-data service requires. First, what do you know about the service's usage patterns? You know that holdings queries the service, but it may be called from elsewhere too—pricing data is used throughout SimpleBank's product.

Let's assume that queries to market-data grow roughly in line with the number of active users on the platform, but you may experience spikes (for example, when the market opens in the morning). You can plan capacity based on predictions of your business growth. Table 6.5 outlines a simple estimation of the QPS that you can expect this service to receive over a three-month period.

Table 6.5 Estimate of calls to a service per second based on growth in average active users over a three-month period

			Jun	*Jul*	*Aug*
Total Users			4000	5600	7840
Expected Growth			40%	40%	40%
Active Users	*Average*	20%	800	1120	1568
	Peak	70%	2800	3920	5488
Service Calls			*Average*		
	Per User/Minute	30	24000	33600	47040
	Per User/Second	0.5	400	560	784
			Peak		
	Per User/Minute	30	84000	117600	164640
	Per User/Second	0.5	1400	1960	2744

Identifying the qualitative factors that drive growth in service utilization is vital to good design and optimizing capacity. Once you've done that, you can determine how much capacity to deploy. For example, the table suggests you need to be able to service 400 requests per second in normal operation, growing by 40% month on month, with spikes in peak usage to 1,400 requests per second.

> **TIP** An in-depth review of capacity and scale planning techniques is outside the scope of this book, but a great overview is available in Abbott and Fisher's *The Art of Scalability* (Addison-Wesley Professional, 2015) (ISBN 978-0134032801).

Once you've established a baseline capacity for your service, you can then iteratively test that capacity against expected traffic patterns. Along with validating the traffic limits of a microservice configuration, load testing can identify potential bottlenecks or design flaws that aren't apparent at lower levels of load. Load testing can provide you with highly effective insight into the limitations of your services.

At the level of individual services, you should automate the load testing of each service as part of its delivery pipeline—something we'll explore in part 3 of this book. Along with this systematic load testing, you should perform exploratory load testing to identify limits and test your assumptions about the load that services can handle.

You also should load test services together. This can aid in identifying unusual load patterns and bottlenecks based on service interaction. For example, you could write a load test that exercises all the services in the `GET /holdings` example.

CHAOS TESTING

Many failures in a microservice application don't arise from within the microservices themselves. Networks fail, virtual machines fail, databases become unresponsive—failure is everywhere! To test for these types of failure scenarios, you need to apply chaos testing.

Chaos testing pushes your microservice application to fail in production. By introducing instability and failure, it accurately mimics real system failures, as well as training an engineering team to be able to react to those failures. This should ultimately build your confidence in the system's capability to withstand real chaos because you'll be gradually improving the resiliency of your system and reducing the possible number of events that would cause operational impact.

As explained on the "Principles of Chaos Engineering" website (https://principlesofchaos.org/), you can think of chaos testing as "the facilitation of experiments to uncover systemic weaknesses." The website lays out this approach:

1. Define a measurable steady state of normal system operation.
2. Hypothesize that behavior in an experimental and control group will remain steady; the system will be resilient to the failure introduced.
3. Introduce variables that reflect real-world failure events—for example, removing servers, severing network connections, or introducing higher levels of latency.
4. Attempt to disprove the hypothesis you defined in (2).

Recall how the holdings, transactions, and market-data services were deployed in figure 6.5. In this case, you expect steady operation to return holdings data within a reasonable response time. A chaos test could introduce several variables:

1. Killing nodes running market-data or transactions, either partially or completely
2. Reducing capacity by killing holdings instances at random
3. Severing the network connection—for example, between holdings and downstream services or between services and their data stores

Figure 6.19 illustrates these options.

Companies with mature chaos testing practices might even perform testing on both a systematic and random basis against live production environments. This might sound terrifying; real outages can be stressful enough, let alone actively working to make them happen. But without taking this approach, it's incredibly difficult to know that your system is truly resilient in the ways that you expect. In any organization, you should start small, by introducing a limited set of possible failures, or only running scheduled, rather than random, tests. Although you can also perform chaos tests in a staging environment, you'll need to carefully consider whether that environment is truly representative of or equivalent to your production configuration.

TIP Chaos Toolkit (http://chaostoolkit.org/) is a great tool to start with if you'd like to practice chaos engineering techniques.

Ultimately, by regularly and systematically validating your system against chaotic events and resolving the issues you encounter, you and your team will be able to achieve a significant level of confidence in your application's resilience to failure.

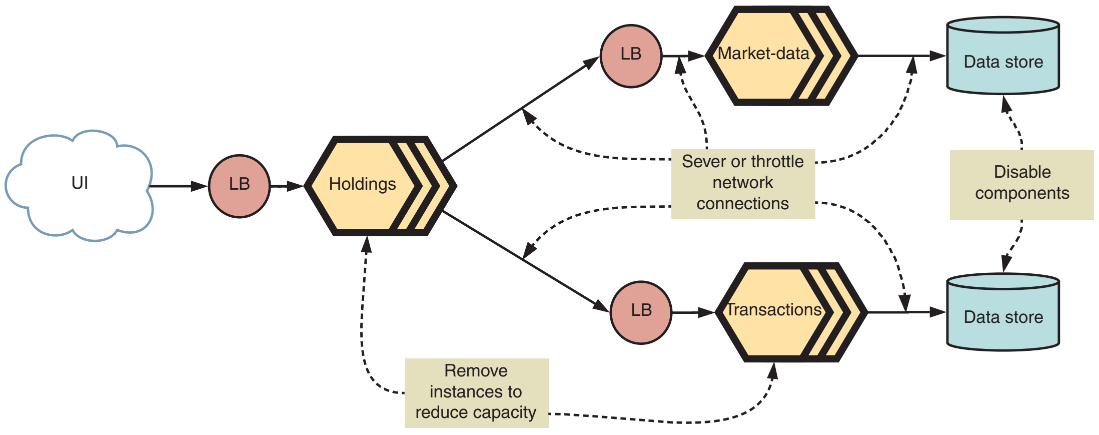


Figure 6.19 Potential variables to introduce in a chaos test to reflect real-world failure events

6.5 Safety by default

Critical paths in your microservice application will only be as resilient and available as their weakest link. Given the impact that individual services can have overall availability, it's imperative to avoid emergencies where introducing new services or changes in a service dependency chain significantly degrade that measure. Likewise, you don't want to find out that crucial functionality can't tolerate faults *when that fault happens*.

When applications are technically heterogeneous, or distinct teams deliver underlying services, it can be exceptionally difficult to maintain consistent approaches to reliable interaction. We touched on this back in chapter 2 when we discussed isolation and technical divergence. Teams are under different delivery pressures and different services have different needs—at worst, developers might forget to follow good resiliency practices.

Any change in service topology can have a negative impact. Figure 6.20 illustrates two examples: adding a new collaborator downstream from market-data might decrease market-data's availability, whereas adding a new consumer might reduce the overall capacity of the market-data service, reducing service for existing consumers.

Frameworks and proxies are two different technical approaches to applying communication standards across multiple services that make it easy for engineers to fall into doing the right thing by ensuring services communicate resiliently and safely by default.

6.5.1 Frameworks

A common approach for ensuring services always communicate appropriately is to mandate the use of specific libraries implementing common interaction patterns like circuit breakers, retries, and fallbacks. Standardizing these interactions across all services using a library has the following advantages:

1. Increases the overall reliability of your application by avoiding roll-your-own approaches to service interaction
2. Simplifies the process of rolling out improvements or optimizations to communication across any number of services
3. Clearly and consistently distinguishes network calls from local calls within code
4. Can be extended to provide supporting functionality, such as collecting metrics on service interactions

This approach tends to be more effective when a company uses one language (or few languages) for writing code; for example, Hystrix, which we mentioned earlier, was intended to provide a standardized way—across all Java-based services in Netflix's organization—of controlling interactions between distributed services.

NOTE Standardizing communication is a crucial element of building a microservice chassis, which we'll explore in the next chapter.

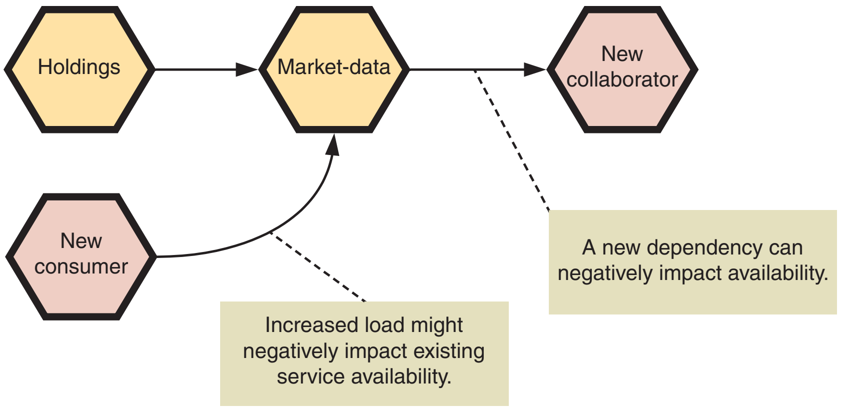

Holdings
Market-data
New collaborator
New consumer
A new dependency can negatively impact availability.
Increased load might negatively impact existing service availability.

Figure 6.20 Availability impact of new services in a dependency chain

6.5.2 Service mesh

Alternatively, you could introduce a *service mesh*, such as Linkerd (https://linkerd.io) or Envoy (www.envoyproxy.io), between your services to control retries, fallbacks, and circuit breakers, rather than making this behavior part of each individual service. A service mesh acts as a proxy. Figure 6.21 illustrates how a service mesh handles communication between services.

Instead of services communicating directly with other services, service communication passes through the service mesh application, typically deployed as a separate process on the same host as the service. You then can configure the proxy to manage that traffic appropriately—retrying requests, managing timeouts, or balancing load across different services. From the caller's perspective, the mesh doesn't exist—it makes HTTP or RPC calls to another service as normal.

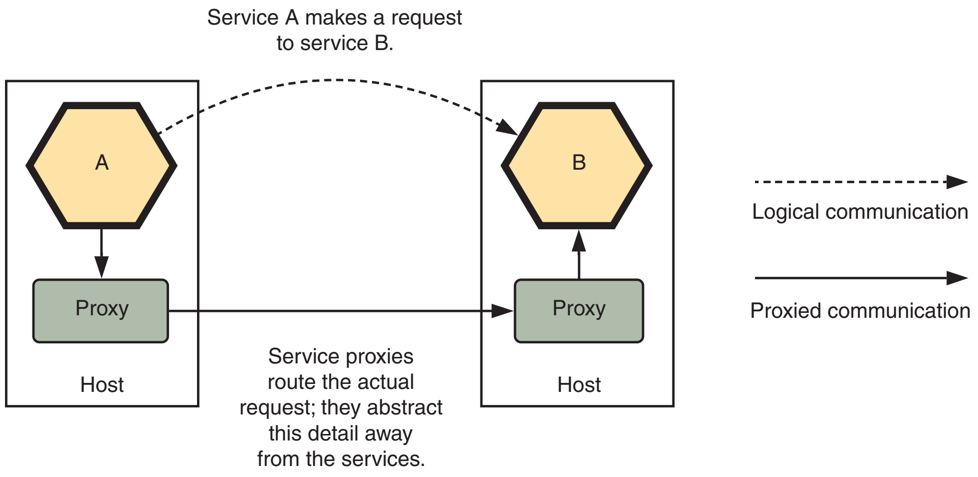

Service A makes a request to service B.
A
B
Proxy
Proxy
Host
Host
Logical communication
Proxied communication
Service proxies route the actual request; they abstract this detail away from the services.

Figure 6.21 Communication between services using a service mesh

Although this may make the treatment of service interaction less explicit to an engineer working on a service, it can simplify defensive communication in applications that are heterogeneous. Otherwise, consistent communication can require significant time investment to achieve across different languages, because ecosystems and libraries may have unequal capabilities or support for resiliency features.

Summary

- Failure is inevitable in complex distributed systems—you have to consider fault tolerance when you're designing them.
- The availability of individual services affects the availability of the wider application.
- Choosing the right level of risk mitigation for an application requires careful consideration of the frequency and impact of failure versus the cost of mitigating against potentially rare events.
- Most failures occur in one of four areas: hardware, communication, dependencies, or internally.
- Cascading failures result from positive feedback and are a common failure mode in a microservice application. They're most commonly caused by server overload.
- You can use retries and deadlines to mitigate against faults in service interactions. You need to apply retries carefully to avoid exacerbating failure in other services.
- You can use fallbacks—such as caching, alternative services, and default results—to return successful responses, even when service dependencies fail.
- You should propagate deadlines between services to ensure they're consistent across a system and to minimize wasted work.
- Circuit breakers between services protect against cascading failures by failing quickly when a high threshold of errors is encountered.
- Services can use rate limits to protect themselves from spikes in load beyond their capacity to service.
- Individual services should expose health checks for load balancers and monitoring to be able to use.
- You can effectively validate resiliency by practicing both load and chaos testing.
- You can apply standards—whether through proxies or frameworks—to help engineers "fall into the pit of success" and build services that tolerate faults by default.

SOFTWARE DEVELOPMENT

Microservices IN ACTION

Bruce • Pereira

Free eBook
See first page

Invest your time in designing great applications, improving infrastructure, and making the most out of your dev teams. Microservices are easier to write, scale, and maintain than traditional enterprise applications because they're built as a system of independent components. Master a few important new patterns and processes, and you'll be ready to develop, deploy, and run production-quality microservices.

Microservices in Action teaches you how to write and maintain microservice-based applications. Created with day-to-day development in mind, this informative guide immerses you in real-world use cases from design to deployment. You'll discover how microservices enable an efficient continuous delivery pipeline, and explore examples using Kubernetes, Docker, and Google Container Engine.

What's Inside

- An overview of microservice architecture
- Building a delivery pipeline
- Best practices for designing multi-service transactions and queries
- Deploying with containers
- Monitoring your microservices

Written for intermediate developers familiar with enterprise architecture and cloud platforms like AWS and GCP.

Morgan Bruce and **Paulo A. Pereira** are experienced engineering leaders. They work daily with microservices in a production environment, using the techniques detailed in this book.

"The one [and only] book on implementing microservices with a real-world, cover-to-cover example you can relate to."
—Christian Bach, Swiss Re

"A perfect fit for those who want to move their majestic monolith to a scalable microservice architecture."
—Akshat Paul
McKinsey & Company

"Shows not only how to write microservices, but also how to prepare your business and infrastructure for this change."
—Maciej Jurkowski, Grupa Pracuj

"A deep dive into microservice development with many real and useful examples."
—Antonio Pessolano
Consoft Sistemi

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/microservices-in-action

ISBN-13: 978-1-61729-445-7
ISBN-10: 1-61729-445-4
54999
9 781617 294457

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]