4 Service Discovery

This is the fourth chapter in this ebook, which is about building applications with microservices. Chapter 1 introduces the Microservices Architecture pattern and discussed the benefits and drawbacks of using microservices. Chapter 2 and Chapter 3 describe different aspects of communication between microservices. In this chapter, we explore the closely related problem of service discovery.

Why Use Service Discovery?

Let's imagine that you are writing some code that invokes a service that has a REST API or Thrift API. In order to make a request, your code needs to know the network location (IP address and port) of a service instance. In a traditional application running on physical hardware, the network locations of service instances are relatively static. For example, your code can read the network locations from a configuration file that is occasionally updated.

In a modern, cloud-based microservices application, however, this is a much more difficult problem to solve, as shown in Figure 4-1.

Service instances have dynamically assigned network locations. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Consequently, your client code needs to use a more elaborate service discovery mechanism.

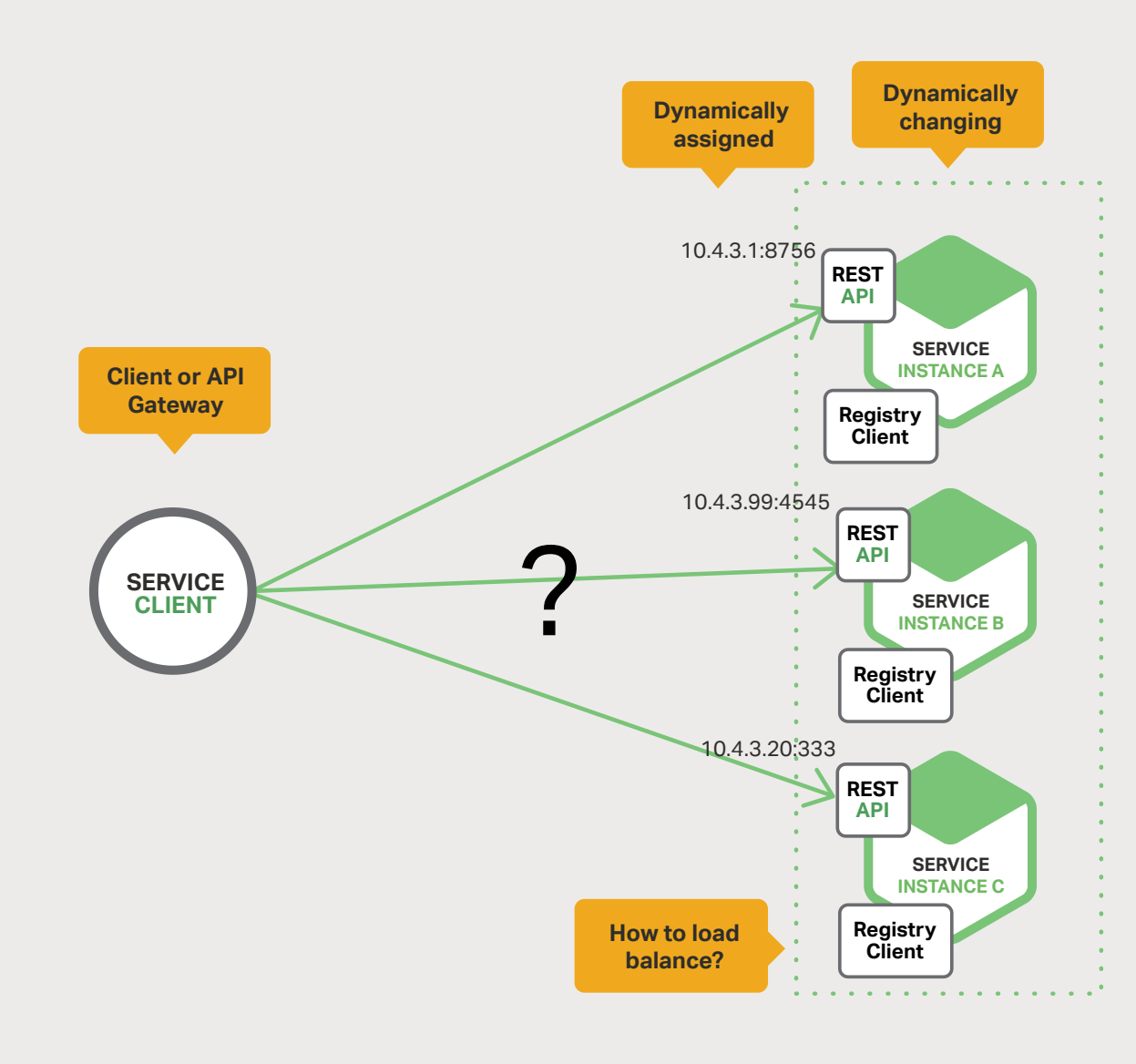

Figure 4-1. A client or API Gateway needs help finding services.

There are two main service discovery patterns: client-side discovery and server-side discovery. Let's first look at client-side discovery.

The Client-Side Discovery Pattern

When using client-side discovery pattern, the client is responsible for determining the network locations of available service instances and load balancing requests across them. The client queries a service registry, which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

Figure 4-2 shows the structure of this pattern:

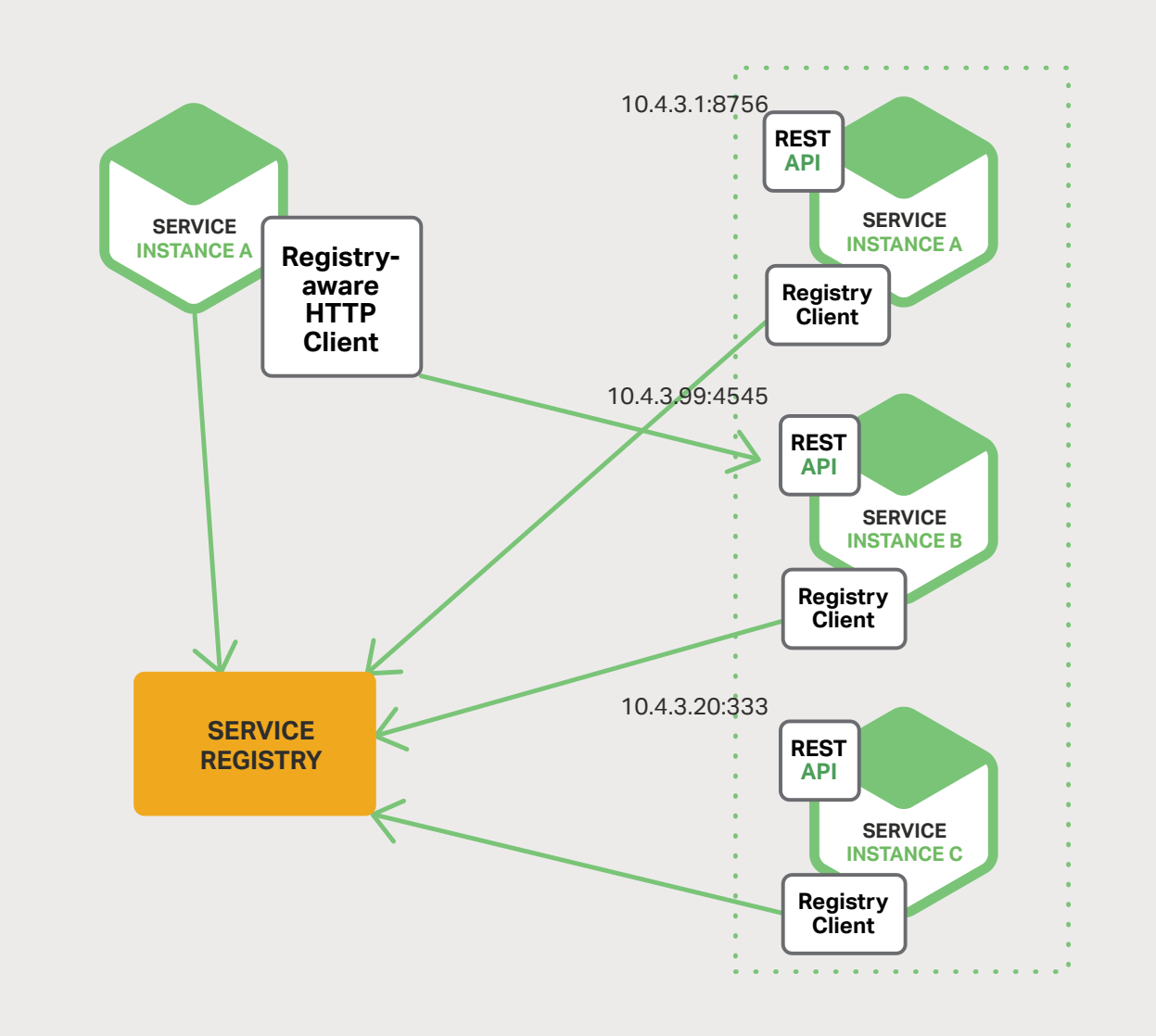

Figure 4-2. Clients can take on the task of discovering services.

The network location of a service instance is registered with the service registry when it starts up. It is removed from the service registry when the instance terminates. The service instance's registration is typically refreshed periodically using a heartbeat mechanism.

Netflix OSS provides a great example of the client-side discovery pattern. Netflix Eureka is a service registry. It provides a REST API for managing service-instance registration and for querying available instances. Netflix Ribbon is an IPC client that works with Eureka to load balance requests across the available service instances. We will discuss Eureka in more depth later in this chapter.

The client-side discovery pattern has a variety of benefits and drawbacks. This pattern is relatively straightforward and, except for the service registry, there are no other moving parts. Also, since the client knows about the available services instances, it can make intelligent, application-specific load-balancing decisions such as using hashing consistently. One significant drawback of this pattern is that it couples the client with the service registry. You must implement client-side service discovery logic for each programming language and framework used by your service clients.

Now that we have looked at client-side discovery, let's take a look at server-side discovery.

The Server-Side Discovery Pattern

The other approach to service discovery is the server-side discovery pattern. Figure 4-3 shows the structure of this pattern:

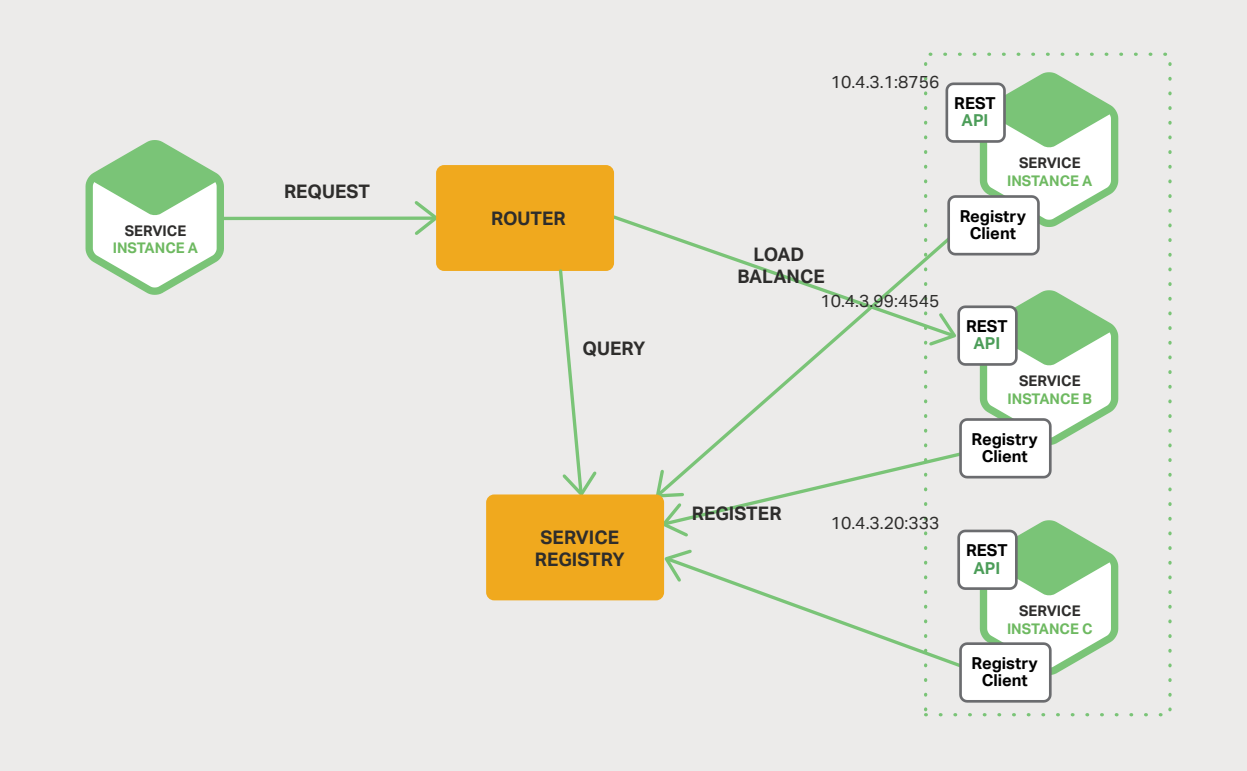

Figure 4-3. Service discovery can also be handled among servers.

The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

The AWS Elastic Load Balancer (ELB) is an example of a server-side discovery router. ELB is commonly used to load balance external traffic from the Internet. However, you can also use ELB to load balance traffic that is internal to a virtual private cloud (VPC).

A client makes requests (HTTP or TCP) via the ELB using its DNS name. The ELB load balances the traffic among a set of registered Elastic Compute Cloud (EC2) instances or EC2 Container Service (ECS) containers. There isn't a separately visible service registry. Instead, EC2 instances and ECS containers are registered with the ELB itself.

HTTP servers and load balancers such as NGINX Plus and NGINX can also be used as a server-side discovery load balancer. For example, this blog post describes using Consul Template to dynamically reconfigure NGINX reverse proxying. Consul Template is a tool that periodically regenerates arbitrary configuration files from configuration data stored in the Consul service registry. It runs an arbitrary shell command whenever the files change. In the example described in the blog post, Consul Template generates an **nginx.conf** file, which configures the reverse proxying, and then runs a command that tells NGINX to reload the configuration. A more sophisticated implementation could dynamically reconfigure NGINX Plus using either its HTTP API or DNS.

Some deployment environments such as Kubernetes and Marathon run a proxy on each host in the cluster. The proxy plays the role of a server-side discovery load balancer. In order to make a request to a service, a client routes the request via the proxy using the host's IP address and the service's assigned port. The proxy then transparently forwards the request to an available service instance running somewhere in the cluster.

The server-side discovery pattern has several benefits and drawbacks. One great benefit of this pattern is that details of discovery are abstracted away from the client. Clients simply make requests to the load balancer. This eliminates the need to implement discovery logic for each programming language and framework used by your service clients. Also, as mentioned above, some deployment environments provide this functionality for free. This pattern also has some drawbacks, however. Unless the load balancer is provided by the deployment environment, it is yet another highly available system component that you need to set up and manage.

The Service Registry

The service registry is a key part of service discovery. It is a database containing the network locations of service instances. A service registry needs to be highly available and up to date. Clients can cache network locations obtained from the service registry. However, that information eventually becomes out of date and clients become unable to discover service instances. Consequently, a service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

As mentioned earlier, Netflix Eureka is good example of a service registry. It provides a REST API for registering and querying service instances. A service instance registers its network location using a `POST` request. Every 30 seconds it must refresh its registration using a `PUT` request. A registration is removed by either using an HTTP `DELETE` request or by the instance registration timing out. As you might expect, a client can retrieve the registered service instances by using an HTTP `GET` request.

Netflix achieves high availability by running one or more Eureka servers in each Amazon EC2 availability zone. Each Eureka server runs on an EC2 instance that has an Elastic IP address. DNS `TEXT` records are used to store the Eureka cluster configuration, which is a map from availability zones to a list of the network locations of Eureka servers. When a Eureka server starts up, it queries DNS to retrieve the Eureka cluster configuration, locates its peers, and assigns itself an unused Elastic IP address.

Eureka clients – services and service clients – query DNS to discover the network locations of Eureka servers. Clients prefer to use a Eureka server in the same availability zone. However, if none is available, the client uses a Eureka server in another availability zone.

Other examples of service registries include:

- etcd – A highly available, distributed, consistent, key-value store that is used for shared configuration and service discovery. Two notable projects that use etcd are Kubernetes and Cloud Foundry.
- Consul – A tool for discovering and configuring services. It provides an API that allows clients to register and discover services. Consul can perform health checks to determine service availability.
- Apache ZooKeeper – A widely used, high-performance coordination service for distributed applications. Apache ZooKeeper was originally a subproject of Hadoop, but is now a separate, top-level project.

Also, as noted previously, some systems such as Kubernetes, Marathon, and AWS do not have an explicit service registry. Instead, the service registry is just a built-in part of the infrastructure.

Now that we have looked at the concept of a service registry, let's look at how service instances are registered with the service registry.

Service Registration Options

As previously mentioned, service instances must be registered with and unregistered from the service registry. There are a couple of different ways to handle the registration and unregistration. One option is for service instances to register themselves, the self-registration pattern. The other option is for some other system component to manage the registration of service instances, the third-party registration pattern. Let's first look at the self-registration pattern.

The Self-Registration Pattern

When using the self-registration pattern, a service instance is responsible for registering and unregistering itself with the service registry. Also, if required, a service instance sends heartbeat requests to prevent its registration from expiring.

Figure 4-4 shows the structure of this pattern.

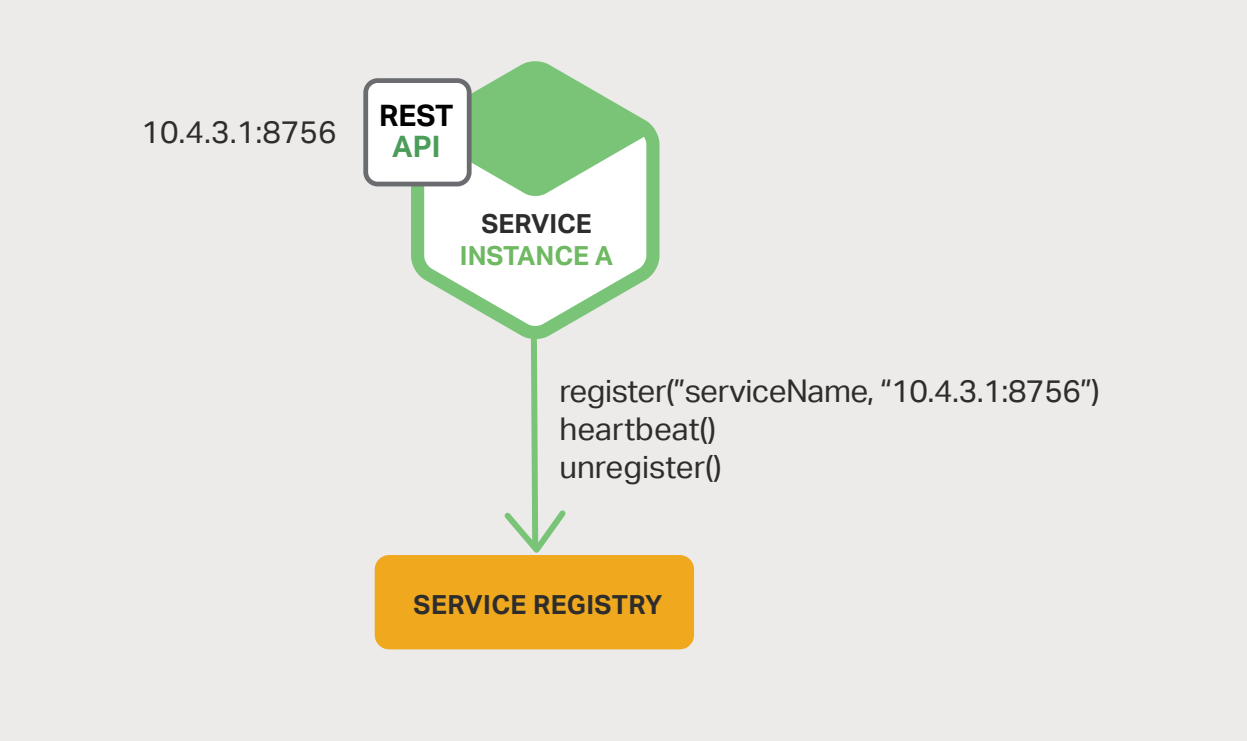

Figure 4-4. Services can handle their own registration.

A good example of this approach is the Netflix OSS Eureka client. The Eureka client handles all aspects of service instance registration and unregistration. The Spring Cloud project, which implements various patterns including service discovery, makes it easy to automatically register a service instance with Eureka. You simply annotate your Java Configuration class with an `@EnableEurekaClient` annotation.

The self-registration pattern has various benefits and drawbacks. One benefit is that it is relatively simple and doesn't require any other system components. However, a major drawback is that it couples the service instances to the service registry. You must implement the registration code in each programming language and framework used by your services.

The alternative approach, which decouples services from the service registry, is the third-party registration pattern.

The Third-Party Registration Pattern

When using the third-party registration pattern, service instances aren't responsible for registering themselves with the service registry. Instead, another system component known as the *service registrar* handles the registration. The service registrar tracks changes to the set of running instances by either polling the deployment environment or subscribing to events. When it notices a newly available service instance, it registers the instance with the service registry. The service registrar also unregisters terminated service instances.

Figure 4-5 shows the structure of this pattern:

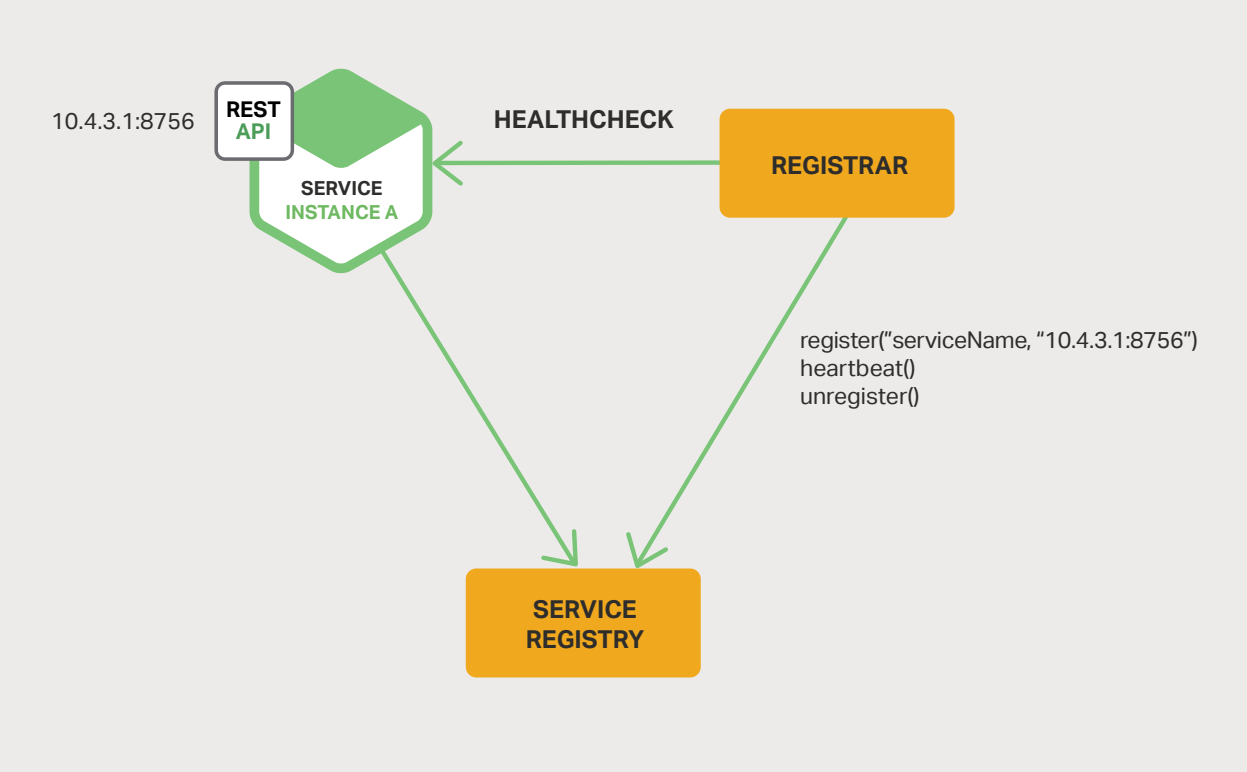

Figure 4-5. A separate registrar service can be responsible for registering others.

One example of a service registrar is the open source Registrator project. It automatically registers and unregisters service instances that are deployed as Docker containers. Registrator supports several service registries, including etcd and Consul.

Another example of a service registrar is NetflixOSS Prana. Primarily intended for services written in non-JVM languages, it is a sidecar application that runs side by side with a service instance. Prana registers and unregisters the service instance with Netflix Eureka.

The service registrar is a built-in component in some deployment environments. The EC2 instances created by an Autoscaling Group can be automatically registered with an ELB. Kubernetes services are automatically registered and made available for discovery.

The third-party registration pattern has various benefits and drawbacks. A major benefit is that services are decoupled from the service registry. You don't need to implement service-registration logic for each programming language and framework used by your developers. Instead, service instance registration is handled in a centralized manner within a dedicated service.

One drawback of this pattern is that unless it's built into the deployment environment, it is yet another highly available system component that you need to set up and manage.

Summary

In a microservices application, the set of running service instances changes dynamically. Instances have dynamically assigned network locations. Consequently, in order for a client to make a request to a service it must use a service-discovery mechanism.

A key part of service discovery is the service registry. The service registry is a database of available service instances. The service registry provides a management API and a query API. Service instances are registered with and unregistered from the service registry using the management API. The query API is used by system components to discover available service instances.

There are two main service-discovery patterns: client-side discovery and service-side discovery. In systems that use client-side service discovery, clients query the service registry, select an available instance, and make a request. In systems that use server-side discovery, clients make requests via a router, which queries the service registry and forwards the request to an available instance.

There are two main ways that service instances are registered with and unregistered from the service registry. One option is for service instances to register themselves with the service registry, the self-registration pattern. The other option is for some other system component to handle the registration and unregistration on behalf of the service, the third-party registration pattern.

In some deployment environments you need to set up your own service-discovery infrastructure using a service registry such as Netflix Eureka, etcd, or Apache ZooKeeper. In other deployment environments, service discovery is built in. For example, Kubernetes and Marathon handle service instance registration and unregistration. They also run a proxy on each cluster host that plays the role of server-side discovery router.

An HTTP reverse proxy and load balancer such as NGINX can also be used as a server-side discovery load balancer. The service registry can push the routing information to NGINX and invoke a graceful configuration update; for example, you can use Consul Template. NGINX Plus supports additional dynamic reconfiguration mechanisms – it can pull information about service instances from the registry using DNS, and it provides an API for remote reconfiguration.

Microservices in Action: NGINX Flexibility

by Floyd Smith

In a microservices environment, your backend infrastructure is likely to be constantly changing as services are created, deployed, and scaled up and down as a result of autoscaling, failures, and upgrades. As described in this chapter, a service discovery mechanism is required in environments where service locations are dynamically reassigned.

Part of the benefit of using NGINX for microservices is that you can easily configure it to automatically react to changes in backend infrastructure. NGINX configuration is not only easy and flexible, it's also compatible with the use of templates, as used in Amazon Web Services, making it easier to manage changes for a specific service and to manage changing sets of services subject to load balancing.

NGINX Plus features an on-the-fly reconfiguration API, eliminating the need to restart NGINX Plus or manually reload its configuration to get it to recognize changes to the set of services being load balanced. In NGINX Plus Release 8 and later, the changes you make with the API can be configured to persist across restarts and configuration reloads. (Reloads do not require a restart and do not drop connections.) And NGINX Plus Release 9 and later have support for service discovery using DNS `SRV` records, enabling tighter integration with existing server discovery platforms, such as Consul and etcd.

We here at NGINX have created a model for managing service discovery:

1. Run separate Docker containers for each of several apps, including a service discovery app such as etcd, a service registration tool, one or more backend servers, and NGINX Plus itself to load balance the other containers.
2. The registration tool monitors Docker for new containers and registers new services with the service discovery tool, also removing containers that disappear.
3. Containers and the services they run are automatically added to or removed from the group of load-balanced upstream servers.

Demo apps for this process are available for several service-discovery apps: Consul APIs, DNS `SRV` records from Consul, etcd, and ZooKeeper.

5 Event-Driven Data Management for Microservices

This is the fifth chapter of this ebook about building applications with microservices. The first chapter introduces the Microservices Architecture pattern and discusses the benefits and drawbacks of using microservices. The second and third describe different aspects of communication within a microservices architecture. The fourth chapter explores the closely related problem of service discovery. In this chapter, we change gears and look at the distributed data management problems that arise in a microservices architecture.

Microservices and the Problem of Distributed Data Management

A monolithic application typically has a single relational database. A key benefit of using a relational database is that your application can use ACID transactions, which provide some important guarantees:

- Atomicity – Changes are made atomically
- Consistency – The state of the database is always consistent
- Isolation – Even though transactions are executed concurrently, it appears they are executed serially
- Durable – Once a transaction has committed, it is not undone

As a result, your application can simply begin a transaction, change (insert, update, and delete) multiple rows, and commit the transaction.

Another great benefit of using a relational database is that it provides SQL, which is a rich, declarative, and standardized query language. You can easily write a query that combines data from multiple tables. The RDBMS query planner then determines the optimal way to execute the query. You don't have to worry about low-level details such as how to access the database. And, because all of your application's data is in one database, it is easy to query.

Unfortunately, data access becomes much more complex when we move to a microservices architecture. That is because the data owned by each microservice is private to that microservice and can only be accessed via its API. Encapsulating the data ensures that the microservices are loosely coupled and can evolve independently of one another. If multiple services access the same data, schema updates require time-consuming, coordinated updates to all of the services.

To make matters worse, different microservices often use different kinds of databases. Modern applications store and process diverse kinds of data, and a relational database is not always the best choice. For some use cases, a particular NoSQL database might have a more convenient data model and offer much better performance and scalability. For example, it makes sense for a service that stores and queries text to use a text search engine such as Elasticsearch. Similarly, a service that stores social graph data should probably use a graph database, such as Neo4j. Consequently, microservices-based applications often use a mixture of SQL and NoSQL databases, the so-called polyglot persistence approach.

A partitioned, polyglot-persistent architecture for data storage has many benefits, including loosely coupled services and better performance and scalability. However, it does introduce some distributed data management challenges.

The first challenge is how to implement business transactions that maintain consistency across multiple services. To see why this is a problem, let's take a look at an example of an online B2B store. The Customer Service maintains information about customers, including their credit lines. The Order Service manages orders and must verify that a new order doesn't violate the customer's credit limit. In the monolithic version of this application, the Order Service can simply use an ACID transaction to check the available credit and create the order.

In contrast, in a microservices architecture the ORDER and CUSTOMER tables are private to their respective services, as shown in Figure 5-1:

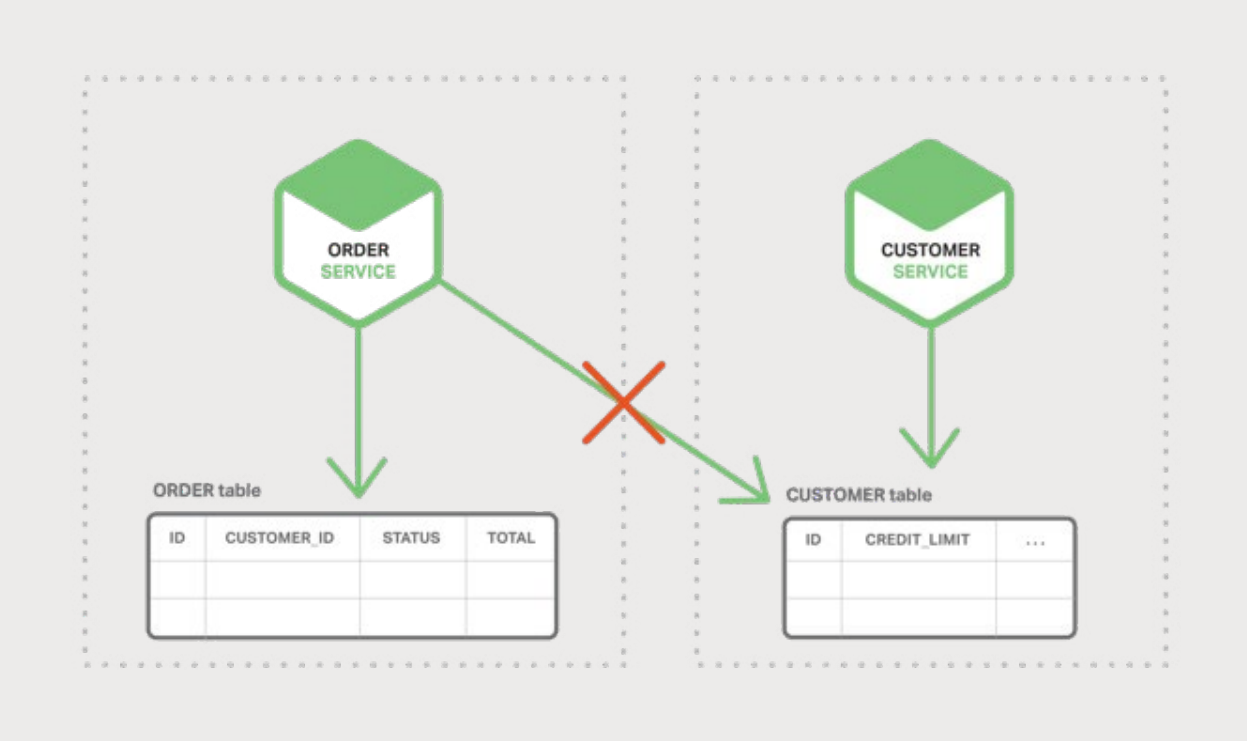

Figure 5-1. Microservices each have their own data.

The Order Service cannot access the CUSTOMER table directly. It can only use the API provided by the Customer Service. The Order Service could potentially use distributed transactions, also known as two-phase commit (2PC). However, 2PC is usually not a viable option in modern applications. The CAP theorem requires you to choose between availability and ACID-style consistency, and availability is usually the better choice. Moreover, many modern technologies, such as most NoSQL databases, do not support 2PC. Maintaining data consistency across services and databases is essential, so we need another solution.

The second challenge is how to implement queries that retrieve data from multiple services. For example, let's imagine that the application needs to display a customer and his recent orders. If the Order Service provides an API for retrieving a customer's orders then you can retrieve this data using an application-side join. The application retrieves the customer from the Customer Service and the customer's orders from the Order Service. Suppose, however, that the Order Service only supports the lookup of orders by their primary key (perhaps it uses a NoSQL database that only supports primary key-based retrievals). In this situation, there is no obvious way to retrieve the needed data.

Event-Driven Architecture

For many applications, the solution is to use an event-driven architecture. In this architecture, a microservice publishes an event when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event it can update its own business entities, which might lead to more events being published.

You can use events to implement business transactions that span multiple services. A transaction consists of a series of steps. Each step consists of a microservice updating a business entity and publishing an event that triggers the next step. The following sequence of diagrams shows how you can use an event-driven approach to checking for available credit when creating an order.

The microservices exchange events via a Message Broker:

- The Order Service creates an Order with status NEW and publishes an Order Created event.

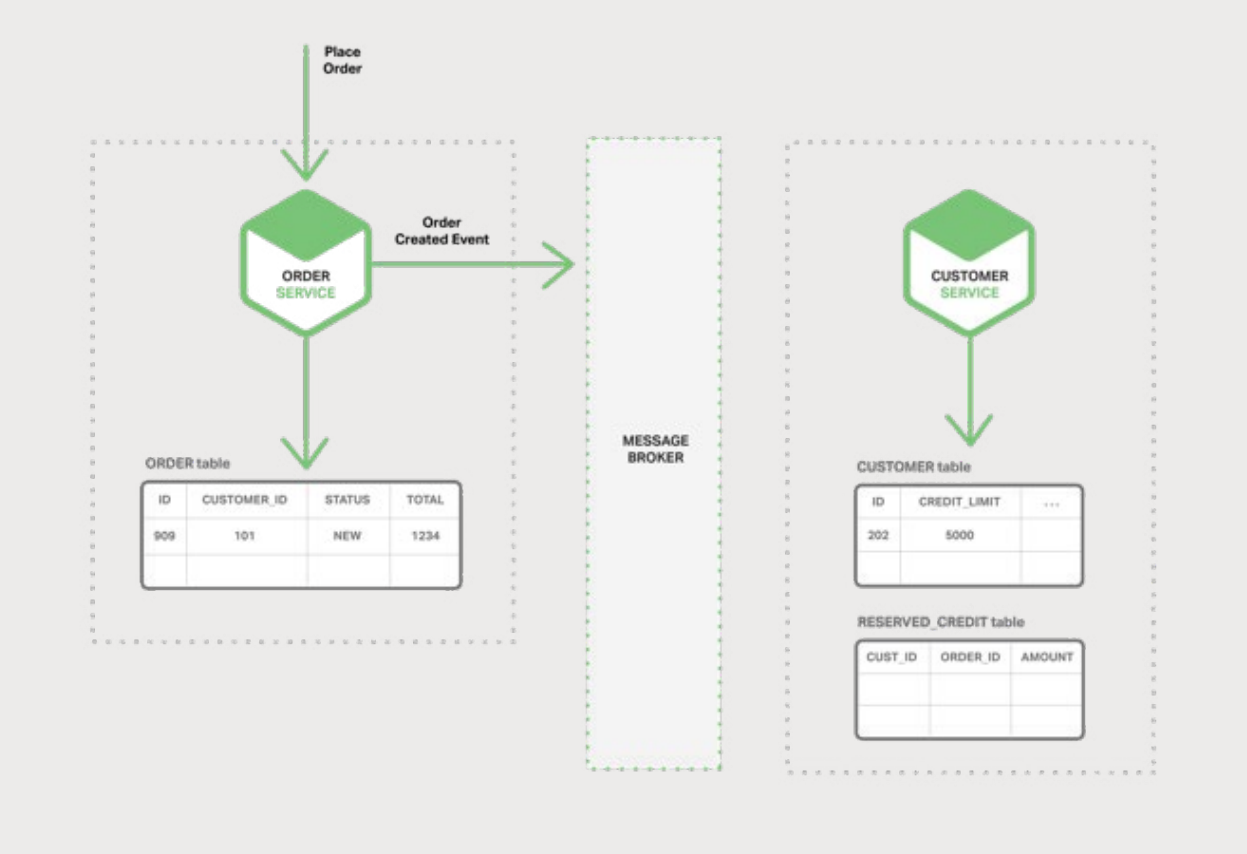

Figure 5-2. The Order Service publishes an event.

- The Customer Service consumes the Order Created event, reserves credit for the order, and publishes a Credit Reserved event.

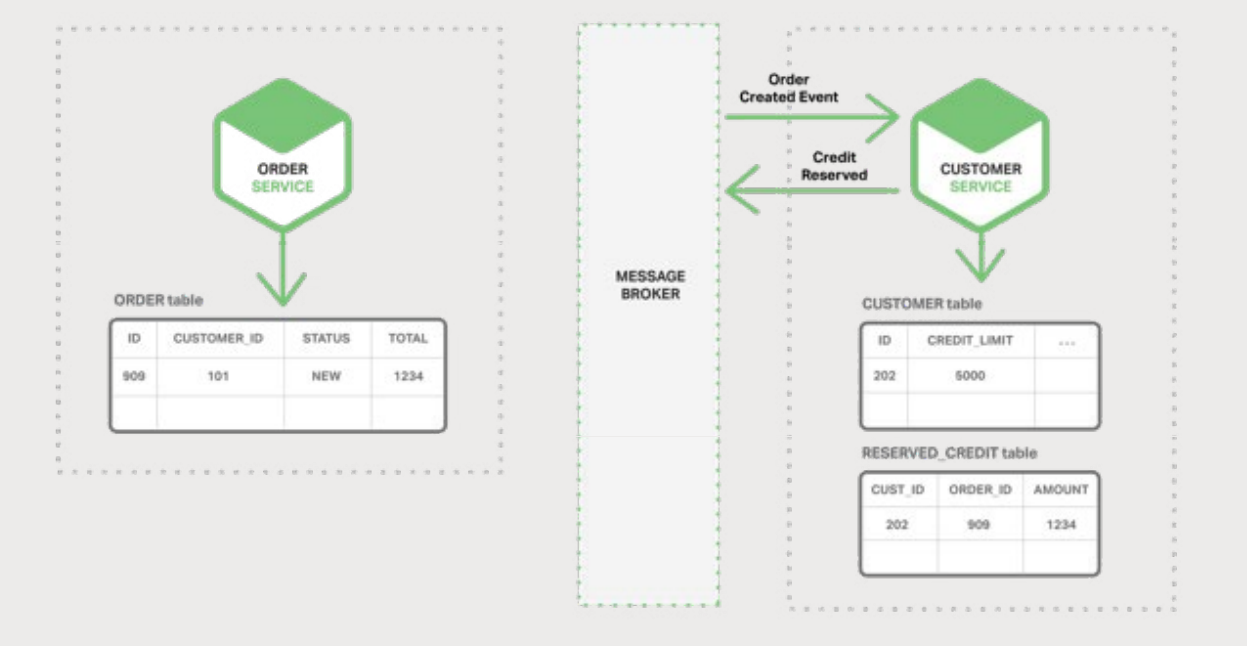

Figure 5-3. The Customer Service responds.

- The Order Service consumes the Credit Reserved event and changes the status of the order to OPEN.

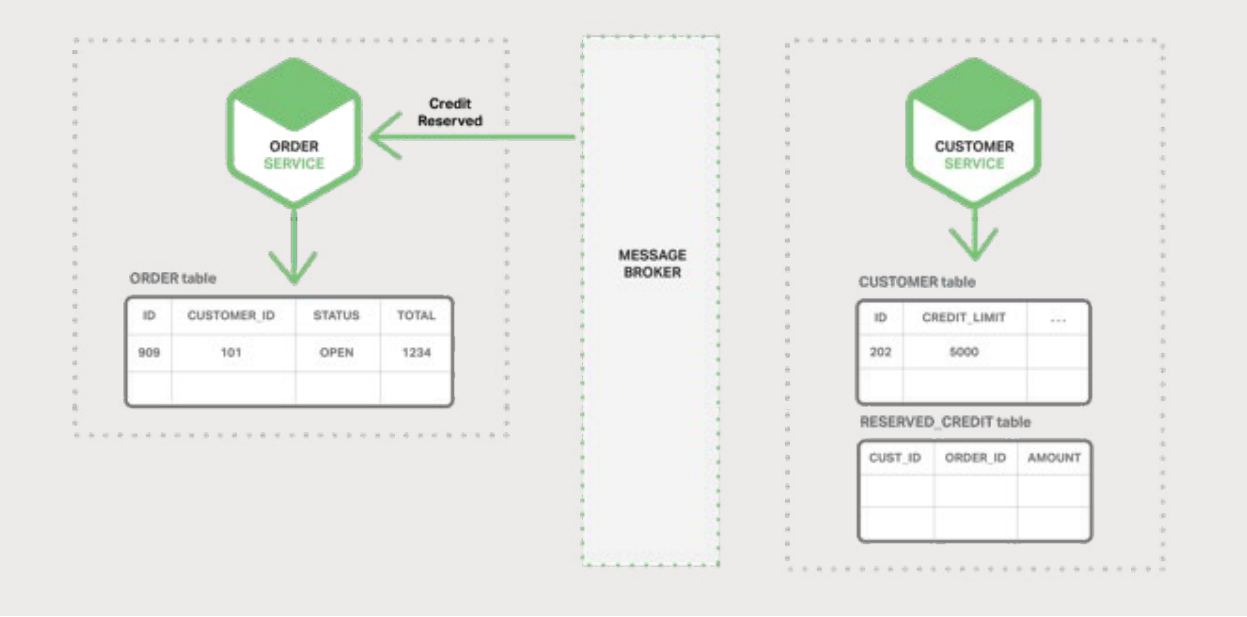

Figure 5-4. The Order Service acts on the response.

A more complex scenario could involve additional steps, such as reserving inventory at the same time as the customer's credit is checked.

Provided that (a) each service atomically updates the database and publishes an event – more on that later – and (b) the Message Broker guarantees that events are delivered at least once, then you can implement business transactions that span multiple services. It is important to note that these are not ACID transactions. They offer much weaker guarantees such as eventual consistency. This transaction model has been referred to as the BASE model.

You can also use events to maintain materialized views that pre-join data owned by multiple microservices. The service that maintains the view subscribes to the relevant events and updates the view. Figure 5-5 depicts a Customer Order View Updater Service that updates the Customer Order View based on events published by the Customer Service and Order Service.

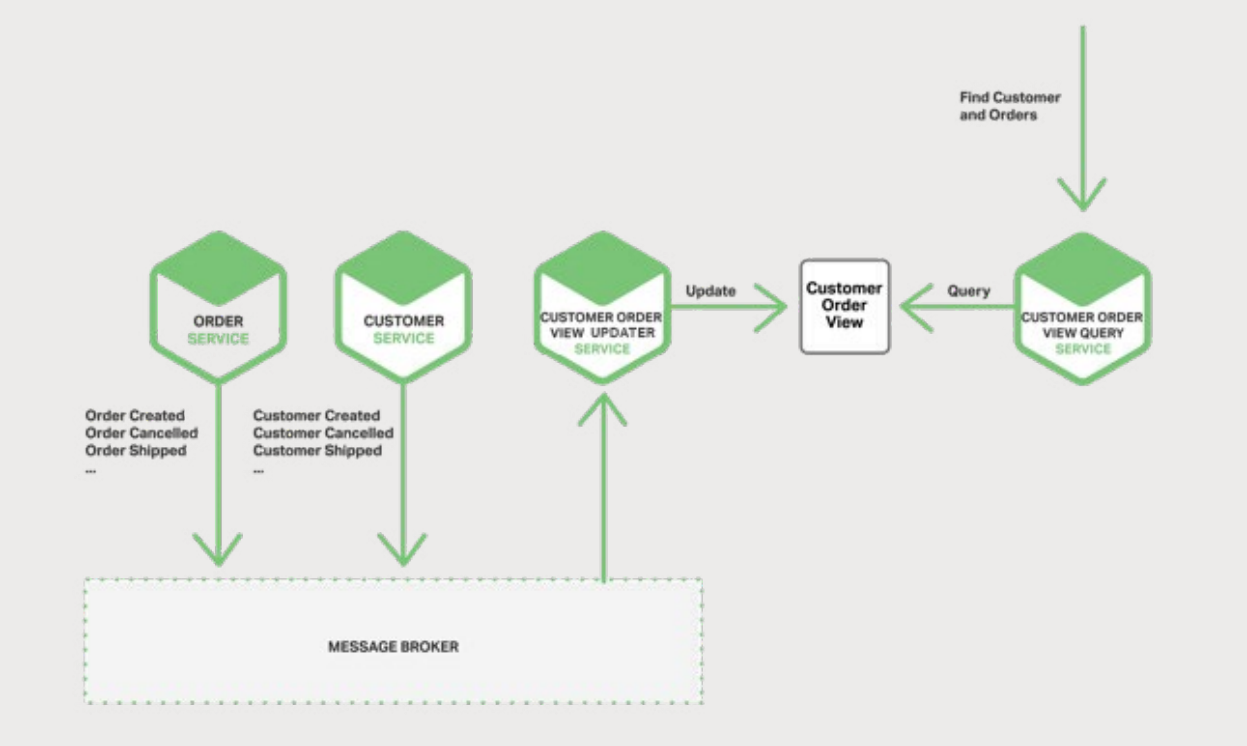

Figure 5-5. The Customer Order View is accessed by two services.

When the Customer Order View Updater Service receives a Customer or Order event, it updates the Customer Order View datastore. You could implement the Customer Order View using a document database such as MongoDB and store one document for each Customer. The Customer Order View Query Service handles requests for a customer and recent orders by querying the Customer Order View datastore.

An event-driven architecture has several benefits and drawbacks. It enables the implementation of transactions that span multiple services and provide eventual consistency. Another benefit is that it also enables an application to maintain materialized views.

One drawback is that the programming model is more complex than when using ACID transactions. Often you must implement compensating transactions to recover from application-level failures; for example, you must cancel an order if the credit check fails. Also, applications must deal with inconsistent data. That is because changes made by in-flight transactions are visible. The application can also see inconsistencies if it reads from a materialized view that is not yet updated. Another drawback is that subscribers must detect and ignore duplicate events.

Achieving Atomicity

In an event-driven architecture there is also the problem of atomically updating the database and publishing an event. For example, the Order Service must insert a row into the ORDER table and publish an Order Created event. It is essential that these two operations are done atomically. If the service crashes after updating the database but before publishing the event, the system becomes inconsistent. The standard way to ensure atomicity is to use a distributed transaction involving the database and the Message Broker. However, for the reasons described above, such as the CAP theorem, this is exactly what we do not want to do.

Publishing Events Using Local Transactions

One way to achieve atomicity is for the application to publish events using a multi-step process involving only local transactions. The trick is to have an EVENT table, which functions as a message queue, in the database that stores the state of the business entities. The application begins a (local) database transaction, updates the state of the business entities, inserts an event into the EVENT table, and commits the transaction. A separate application thread or process queries the EVENT table, publishes the events to the Message Broker, and then uses a local transaction to mark the events as published. Figure 5-6 shows the design.

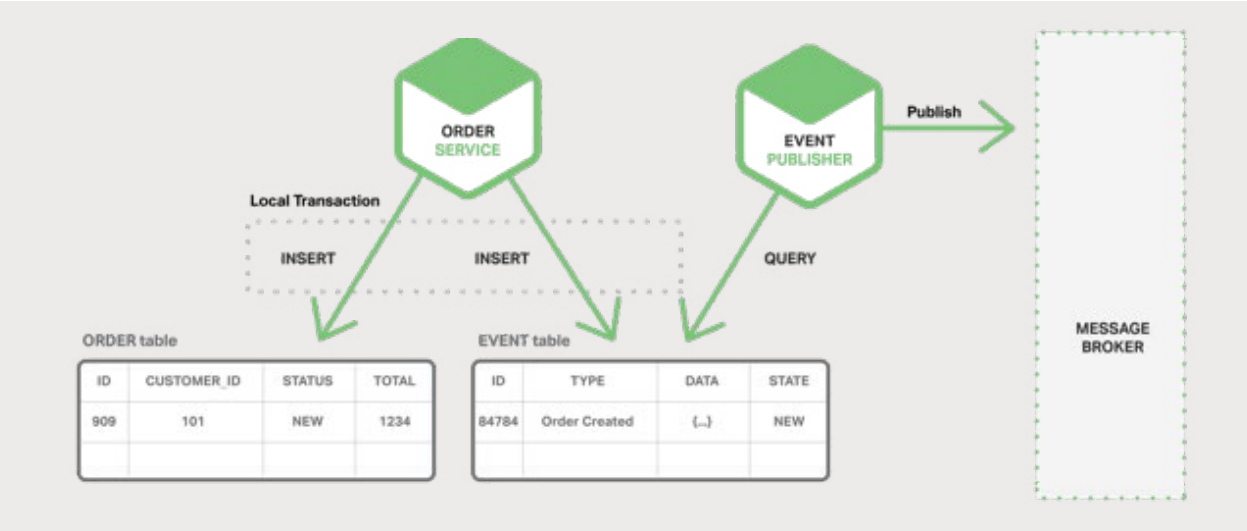

Figure 5-6. Achieving atomicity with local transactions.

The Order Service inserts a row into the ORDER table and inserts an Order Created event into the EVENT table. The Event Publisher thread or process queries the EVENT table for unpublished events, publishes the events, and then updates the EVENT table to mark the events as published.

This approach has several benefits and drawbacks. One benefit is that it guarantees an event is published for each update without relying on 2PC. Also, the application publishes business-level events, which eliminates the need to infer them. One drawback of this approach is that it is potentially error-prone since the developer must remember to publish events. A limitation of this approach is that it is challenging to implement when using some NoSQL databases because of their limited transaction and query capabilities.

This approach eliminates the need for 2PC by having the application use local transactions to update state and publish events. Let's now look at an approach that achieves atomicity by having the application simply update state.

Mining a Database Transaction Log

Another way to achieve atomicity without 2PC is for the events to be published by a thread or process that mines the database's transaction or commit log. The application updates the database, so changes are recorded in the database's transaction log. The Transaction Log Miner thread or process reads the transaction log and publishes events to the Message Broker. Figure 5-7 shows the design.

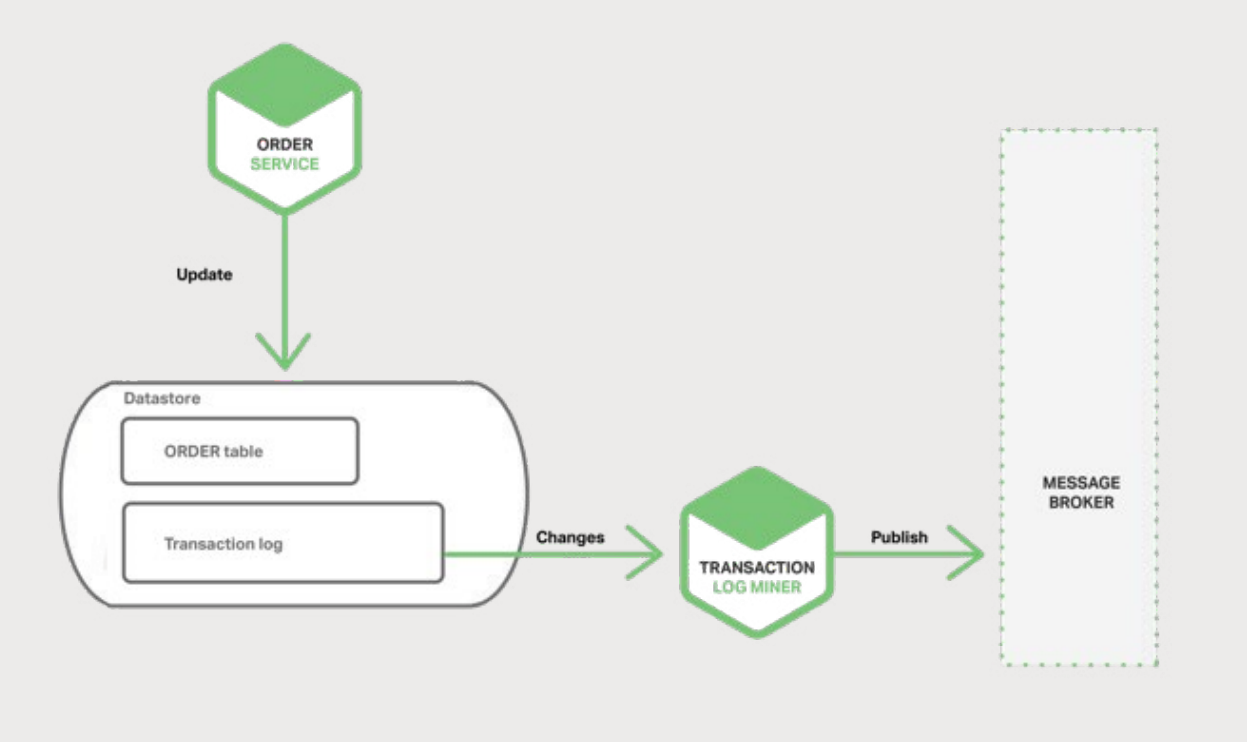

Figure 5-7. A Message Broker can arbitrate data transactions.

An example of this approach is the open source LinkedIn Databus project. Databus mines the Oracle transaction log and publishes events corresponding to the changes. LinkedIn uses Databus to keep various derived data stores consistent with the system of record.

Another example is the streams mechanism in AWS DynamoDB, which is a managed NoSQL database. A DynamoDB stream contains the time-ordered sequence of changes (create, update, and delete operations) made to the items in a DynamoDB table in the last 24 hours. An application can read those changes from the stream and, for example, publish them as events.

Transaction log mining has various benefits and drawbacks. One benefit is that it guarantees that an event is published for each update without using 2PC. Transaction log mining can also simplify the application by separating event publishing from the application's business logic. A major drawback is that the format of the transaction log is proprietary to each database and can even change between database versions. Also, it can be difficult to reverse engineer the high-level business events from the low-level updates recorded in the transaction log.

Transaction log mining eliminates the need for 2PC by having the application do one thing: update the database. Let's now look at a different approach that eliminates the updates and relies solely on events.

Using Event Sourcing

Event sourcing achieves atomicity without 2PC by using a radically different, event-centric approach to persisting business entities. Rather than store the current state of an entity, the application stores a sequence of state-changing events. The application reconstructs an entity's current state by replaying the events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic.

To see how event sourcing works, consider the Order entity as an example. In a traditional approach, each order maps to a row in an ORDER table and to rows in, for example, an ORDER_LINE_ITEM table.

But when using event sourcing, the Order Service stores an Order in the form of its state-changing events: Created, Approved, Shipped, Cancelled. Each event contains sufficient data to reconstruct the Order's state.

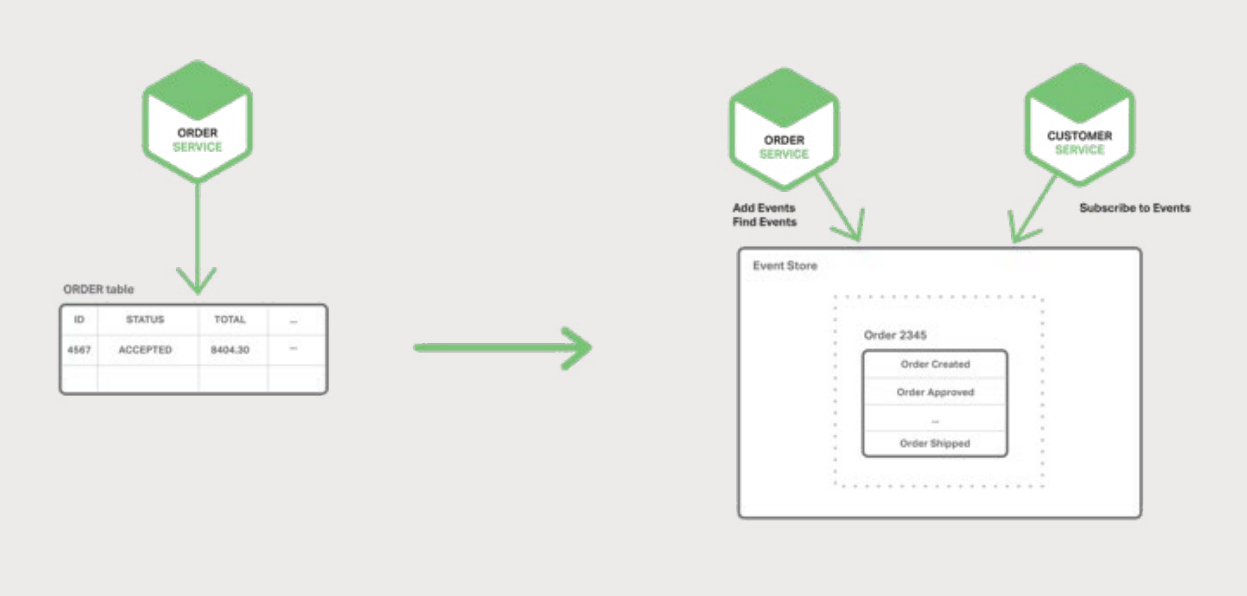

Figure 5-8. Events can have complete recovery data.

Events persist in an Event Store, which is a database of events. The store has an API for adding and retrieving an entity's events. The Event Store also behaves like the Message Broker in the architectures we described previously. It provides an API that enables services to subscribe to events. The Event Store delivers all events to all interested subscribers. The Event Store is the backbone of an event-driven microservices architecture.

Event sourcing has several benefits. It solves one of the key problems in implementing an event-driven architecture and makes it possible to reliably publish events whenever state changes. As a result, it solves data consistency issues in a microservices architecture. Also, because it persists events rather than domain objects, it mostly avoids the object-relational impedance mismatch problem. Event sourcing also provides a 100% reliable audit log of the changes made to a business entity and makes it possible to implement temporal queries that determine the state of an entity at any point in time. Another major benefit of event sourcing is that your business logic consists of loosely coupled business entities that exchange events. This makes it a lot easier to migrate from a monolithic application to a microservices architecture.

Event sourcing also has some drawbacks. It is a different and unfamiliar style of programming and so there is a learning curve. The event store only directly supports the lookup of business entities by primary key. You must use command query responsibility separation (CQRS) to implement queries. As a result, applications must handle eventually consistent data.

Summary

In a microservices architecture, each microservice has its own private datastore. Different microservices might use different SQL and NoSQL databases. While this database architecture has significant benefits, it creates some distributed data management challenges. The first challenge is how to implement business transactions that maintain consistency across multiple services. The second challenge is how to implement queries that retrieve data from multiple services.

For many applications, the solution is to use an event-driven architecture. One challenge with implementing an event-driven architecture is how to atomically update state and how to publish events. There are a few ways to accomplish this, including using the database as a message queue, transaction log mining, and event sourcing.

Microservices in Action: NGINX and Storage Optimization

by Floyd Smith

A microservices-based approach to storage involves a greater number and variety of data stores, more complexity in how you access and update data, and greater challenges for both Dev and Ops in maintaining data consistency. NGINX provides crucial support for this kind of data management, in three main areas:

1. **Caching and microcaching of data** – Caching static files and microcaching application-generated content with NGINX reduces the load on your application, increasing performance and reducing the potential for problems.
2. **Flexibility and scalability per data store** – Once you implement NGINX as a reverse proxy server, your apps gain great flexibility in creating, sizing, running, and resizing data storage servers to meet changing requirements – vital when every service has its own data store.
3. **Monitoring and management of services, including data services** – With the number of data servers multiplying, supporting complex operations is critical, as are monitoring and management tools. NGINX Plus has built-in tools and interfaces to application performance management partners such as Data Dog, Dynatrace, and New Relic.

Examples of microservice-specific data management are included in the three Models of the NGINX Microservices Reference Architecture, giving you a starting point for your own design decisions and implementation.

6 Choosing a Microservices Deployment Strategy

This is the sixth chapter in this ebook about building applications with microservices. Chapter 1 introduces the Microservices Architecture pattern and discusses the benefits and drawbacks of using microservices. The following chapters discuss different aspects of the microservices architecture: using an API Gateway, inter-process communication, service discovery, and event-driven data management. In this chapter, we look at strategies for deploying microservices.

Motivations

Deploying a monolithic application means running one or more identical copies of a single, usually large, application. You typically provision N servers (physical or virtual) and run M instances of the application on each server. The deployment of a monolithic application is not always entirely straightforward, but it is much simpler than deploying a microservices application.

A microservices application consists of tens or even hundreds of services. Services are written in a variety of languages and frameworks. Each one is a mini-application with its own specific deployment, resource, scaling, and monitoring requirements. For example, you need to run a certain number of instances of each service based on the demand for that service. Also, each service instance must be provided with the appropriate CPU, memory, and I/O resources. What is even more challenging is that despite this complexity, deploying services must be fast, reliable and cost-effective.

There are a few different microservice deployment patterns. Let's look first at the Multiple Service Instances per Host pattern.

Multiple Service Instances Per Host Pattern

One way to deploy your microservices is to use the Multiple Service Instances per Host pattern. When using this pattern, you provision one or more physical or virtual hosts and run multiple service instances on each one. In many ways, this is the traditional approach to application deployment. Each service instance runs at a well-known port on one or more hosts. The host machines are commonly treated like pets.

Figure 6-1 shows the structure of this pattern:

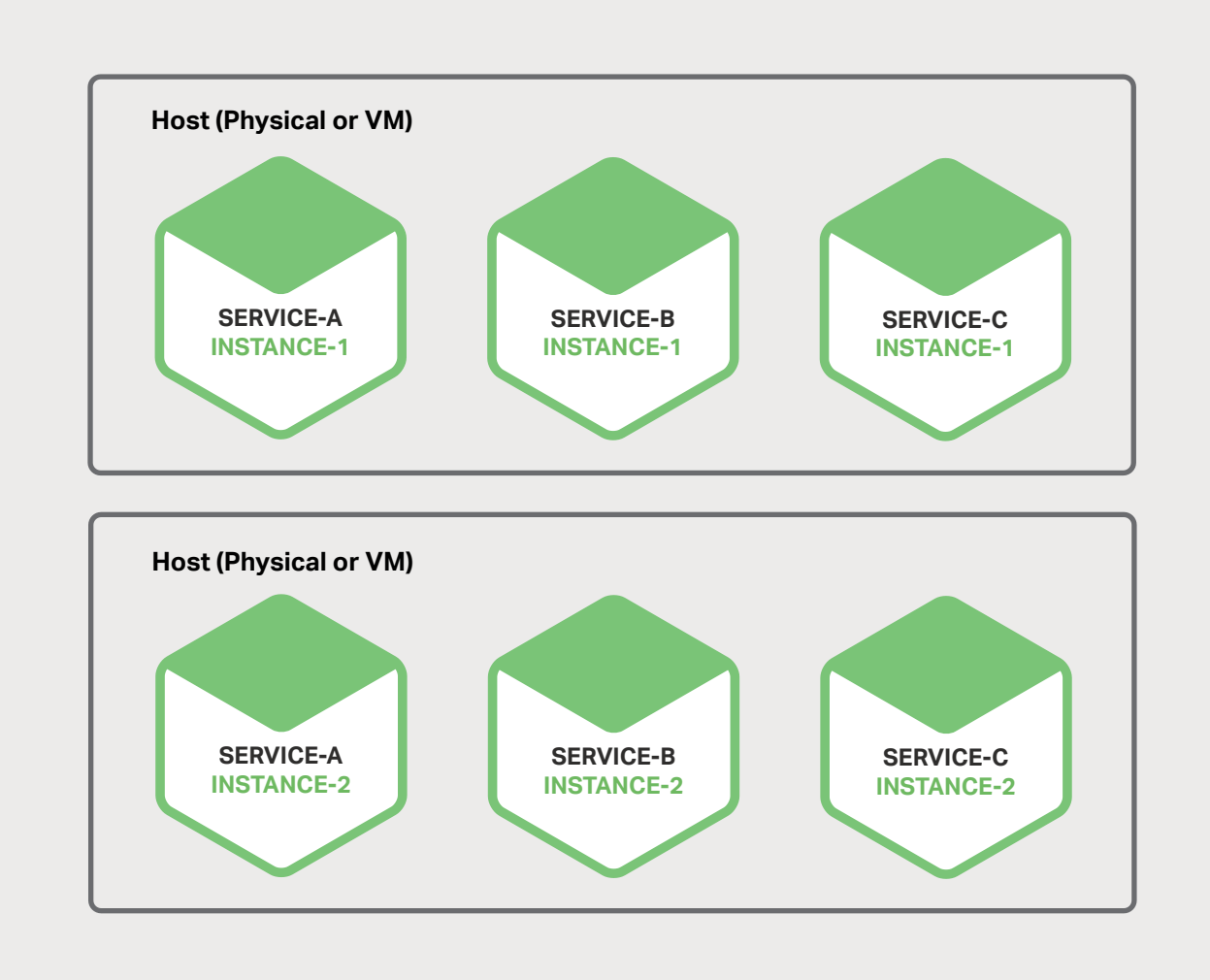

Figure 6-1. Hosts can each support multiple service instances.

There are a couple of variants of this pattern. One variant is for each service instance to be a process or a process group. For example, you might deploy a Java service instance as a web application on an Apache Tomcat server. A Node.js service instance might consist of a parent process and one or more child processes.

The other variant of this pattern is to run multiple service instances in the same process or process group. For example, you could deploy multiple Java web applications on the same Apache Tomcat server or run multiple OSGI bundles in the same OSGI container.

The Multiple Service Instances per Host pattern has both benefits and drawbacks. One major benefit is its resource usage is relatively efficient. Multiple service instances share the server and its operating system. It's even more efficient if a process or group runs multiple service instances, for example, multiple web applications sharing the same Apache Tomcat server and JVM.

Another benefit of this pattern is that deploying a service instance is relatively fast. You simply copy the service to a host and start it. If the service is written in Java, you copy a JAR or WAR file. For other languages, such as Node.js or Ruby, you copy the source code. In either case, the number of bytes copied over the network is relatively small.

Also, because of the lack of overhead, starting a service is usually very fast. If the service is its own process, you simply start it. Otherwise, if the service is one of several instances running in the same container process or process group, you either dynamically deploy it into the container or restart the container.

Despite its appeal, the Multiple Service Instances per Host pattern has some significant drawbacks. One major drawback is that there is little or no isolation of the service instances, unless each service instance is a separate process. While you can accurately monitor each service instance's resource utilization, you cannot limit the resources each instance uses. It's possible for a misbehaving service instance to consume all of the memory or CPU of the host.

There is no isolation at all if multiple service instances run in the same process. All instances might, for example, share the same JVM heap. A misbehaving service instance could easily break the other services running in the same process. Moreover, you have no way to monitor the resources used by each service instance.

Another significant problem with this approach is that the operations team that deploys a service has to know the specific details of how to do it. Services can be written in a variety of languages and frameworks, so there are lots of details that the development team must share with operations. This complexity increases the risk of errors during deployment.

As you can see, despite its familiarity, the Multiple Service Instances per Host pattern has some significant drawbacks. Let's now look at other ways of deploying microservices that avoid these problems.

Service Instance per Host Pattern

Another way to deploy your microservices is the Service Instance per Host pattern. When you use this pattern, you run each service instance in isolation on its own host. There are two different different specializations of this pattern: Service Instance per Virtual Machine and Service Instance per Container.

Service Instance per Virtual Machine Pattern

When you use Service Instance per Virtual Machine pattern, you package each service as a virtual machine (VM) image such as an Amazon EC2 AMI. Each service instance is a VM (for example, an EC2 instance) that is launched using that VM image.

Figure 6-2 shows the structure of this pattern:

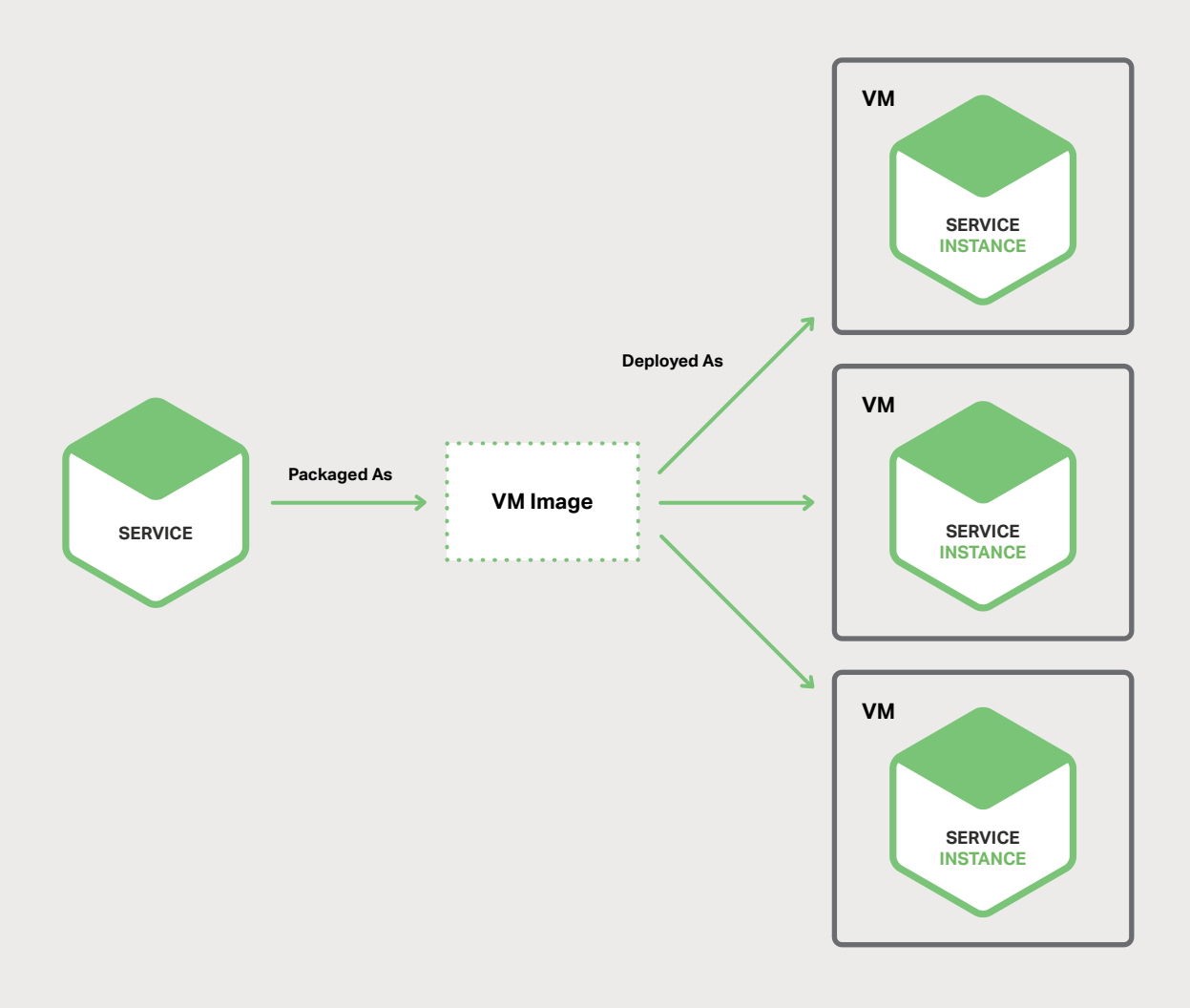

Figure 6-2. Services can each live in their own virtual machine.

This is the primary approach used by Netflix to deploy its video streaming service. Netflix packages each of its services as an EC2 AMI using Aminator. Each running service instance is an EC2 instance.

There are a variety tools that you can use to build your own VMs. You can configure your continuous integration (CI) server (for example, Jenkins) to invoke Aminator to package your services as an EC2 AMI. Packer is another option for automated VM image creation. Unlike Aminator, it supports a variety of virtualization technologies including EC2, DigitalOcean, VirtualBox, and VMware.

The company Boxfuse has a compelling way to build VM images, which overcomes the drawbacks of VMs that I describe below. Boxfuse packages your Java application as a minimal VM image. These images are fast to build, boot quickly, and are more secure since they expose a limited attack surface.

The company CloudNative has the Bakery, a SaaS offering for creating EC2 AMIs. You can configure your CI server to invoke the Bakery after the tests for your microservice pass. The Bakery then packages your service as an AMI. Using a SaaS offering such as the Bakery means that you don't have to waste valuable time setting up the AMI creation infrastructure.

The Service Instance per Virtual Machine pattern has a number of benefits. A major benefit of VMs is that each service instance runs in complete isolation. It has a fixed amount of CPU and memory and can't steal resources from other services.

Another benefit of deploying your microservices as VMs is that you can leverage mature cloud infrastructure. Clouds such as AWS provide useful features such as load balancing and autoscaling.

Another great benefit of deploying your service as a VM is that it encapsulates your service's implementation technology. Once a service has been packaged as a VM it becomes a black box. The VM's management API becomes the API for deploying the service. Deployment becomes much simpler and more reliable.

The Service Instance per Virtual Machine pattern has some drawbacks, however. One drawback is less efficient resource utilization. Each service instance has the overhead of an entire VM, including the operating system. Moreover, in a typical public IaaS, VMs come in fixed sizes and it is possible that the VM will be underutilized.

Moreover, a public IaaS typically charges for VMs regardless of whether they are busy or idle. An IaaS such as AWS provides autoscaling, but it is difficult to react quickly to changes in demand. Consequently, you often have to overprovision VMs, which increases the cost of deployment.

Another downside of this approach is that deploying a new version of a service is usually slow. VM images are typically slow to build due to their size. Also, VMs are typically slow to instantiate, again because of their size. Also, an operating system typically takes some time to start up. Note, however, that this is not universally true, since lightweight VMs such as those built by Boxfuse exist.