# Chapter 5 – Microservices

More and more, software engineers are being asked to code microservices. Microservices are an enabler for the containerization that supports rapid deployment. Microservices also fit  naturally into the agile processes that most organizations use to develop their code. For these reasons, it is important for software engineers to understand microservices, their properties, and how they are used.

When you completed this chapter you will be familiar with microservices, how they relate to team size and team interactions, and the quality attribute characteristics of microservices. Just as important, you will be familiar with how microservices fit into the ecosystem that is a modern large internet application. You will learn how microservices communicate, how to design microservices to support rapid deployment and protect against certain types of cloud failure, how to enable your microservice to be discovered once it is deployed, and how to build your microservice so that you can monitor its activities once it is deployed.

## Microservice architecture defined.

Microservices date from roughly 2002 when Amazon promulgated the following rules for their developers, although the term *microservices* came later:

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they [services] use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable.

Notice the linkage between teams and services in these rules. We will return to this in a moment but first we discuss the key elements of these rules.

- *A service is the basic packaging unit*. Services are independently deployable.  Independent deployment of Service A means that its current location must be dynamically discovered. We have seen one means of discovery (DNS) and there are others but discovery is an important element of a microservice architecture.
- *Microservices communicate only via messages*. Message passing is an inherent portion of a microservice architecture. It is not a coincidence that microservice architectures date from around 2002. This is when the cloud platform became mature enough to support fast message passing. Data interchange protocols are therefore another important element of microservice architectures and we discuss them shortly.
- *It doesn't matter what technology they use*. A major cause of integration errors is version incompatibility. Suppose your team is using Version 2.12 of a particular library and my team is

using Version 2.13. There is no guarantee that these two versions are compatible. More fundamentally, suppose your team wishes to use Java and my team wishes to use Scala. As long as both teams include support for message passing in their libraries we do not need to agree on a development language. Technology independence actually follows from the first two points but it is an important enabler for modern development practices, as we shall see.

Although not inherent in the definition, a microserivce is intended to perform only a single function. You can visualize it as a packaging mechanism for a procedure. Although this visualization is not strictly accurate, it helps when you think about what to place in a microservice. An application designed around a microservice architecture will consist of a large number of small microservices that coordinate to provide the application. In fact, Amazon's web page has upwards of 140 services and a larger number of downstream services. Netflix has over 800 microservices.

Microservice architectures can be viewed as a superclass of Service Oriented Architectures (SOA). That is, they are independent services that communicate via messages. SOA has a number of other elements however: Service Buses, Brokers. Elaborate protocols for interoperability, independent owning organizations, and so forth are all a part of SOA. Not so with microservice architectures. Microservice architectures are owned by a single organization and, although services may be externalized, the ownership of these services is not spread across organizations. In fact, as we shall see next, the ownership is not even spread across multiple teams.

## Microservices and Teams

Amazon also, famously, has a "two pizza" rule.  Every team can be fed with two pizzas. Of course, this depends on the appetites of the team members but what this means in practice is that a team is roughly 7 people.

Furthermore, each microservice is owned by a single team. A single team may own multiple microservices but no microservice has multiple owners. A consequence of the team size and the ownership practice is that microservices are small. "Small" is another vague term but 5K-10K lines of code is a common size for a microservice. The fact each microservice is owned by a single team means that coordination between teams is limited. Some organizations such as Netflix have a separate team whose responsibility is coordinating requirements among teams. In any case, limited coordination leads teams to treat other teams as they would outside entities and this leads to defensive programming. That is, every service should verify that the parameters it has been sent make both syntactic and logical sense. Every team should be prepared for invocations that do not conform to their current specification and these invocations should be treated gently. Gently in this case means an error return that says "I do not understand your invocation". This is as opposed to returning an uninformative error message or, even worse, just failing.

In addition, microservices evolve. As a microservice supports more features, it may grow beyond the owning team's capacity. In this case, the microservice is split. One portion of the split microservice remains with the original owning team and the other portion is assigned to another, possibly new, team. The portion not with the original team must be understandable by the new team and may or may not conform to the conventions of the new team. Thus, all of the standard software engineering problems with maintainability do not disappear with the adoption of microservice architectures.

## Microservice qualities

As with any other architectural style, a microservice architecture favors certain quality attributes over others. The quality attributes are the "ilities" of a system. The four that are most commonly referred to are performance, availability, security, and modifiability. There are many others. In this section we analyze the microservice architectural style from the perspective of these quality attributes. We order the quality attributes alphabetically so as not to be accused of favoritism and begin with availability.

### Availability

Availability refers to a property of a microservice that it is there and ready to carry out its task when you need it to be. It is typically measured in terms of percentage of uptime over the course of some period. 4 9s (99.99% uptime) translates to 52.56 minutes downtime over a year. Scheduled downtime is typically excluded from this measurement. Other numbers relating to availability are MTTD (Mean Time To Discovery) and MTTR (Mean Time To Repair). MTTD is how long it takes to discover that a service failed and MTTR is how long it takes to repair a service once it has failed.

We distinguish between the failure of an instance of a microservice and the failure of the microservice itself. If there is only one instance of a microservice, the two are identical but they have different diagnostics and recovery mechanisms. You may discuss availability percentage, MTTD, and MTTR for either case but the failure of a microservice is more consequential than the failure of an instance of that microservice, depending on how the microservice is architected.

We begin by discussing the failure of an instance of a microservice.

In practice, it is difficult to distinguish an instance failing from an instance rendering poor performance. In a distributed system the main mechanism for determining failure is time out. That is, an instance fails to respond to a message or fails to send an "I am alive" message within a specified time. Failure to respond or failure to send a health message may be due to the instance being overloaded and not because the instance has failed.

In this section, we discuss failure and recovery from failure. We will deal with poor performance in the section on performance.

As we said, the fundamental failure detection mechanism in a distributed system is timeout. In order to maintain availability, the timeout must be recognized by an entity that has the power to take action to repair the service. Your browser, for example, may have a message time out when talking to a web server but its action is limited to retrying the message, sending the message to another server, or reporting failure to you, the user.

In a microservice architecture, the failure of an instance is detected by the load balancer. It recognizes failure through the lack of an "I am alive message". An instance sends this message to the load balancer periodically. A typical period is 90 seconds. If the load balancer does not get a health message in an appropriate time, it puts the instance on an "unhealthy" list and does not send it any more messages. If it eventually does get a message from the instance, it removes it from the unhealthy list. The load balancer should log all of these interactions with its instances.

Recovery from failure in the multiple instance case, is done by creating another instance of the microservice. As long as the microservice is stateless, creating another instance is all that is required. If the microservice is stateful, then the state that the microservice has retained must be recovered. A distributed coordination service such as Zookeeper or Memchached can be used for this purpose.

As an aside, use of a distributed coordination system is, in essence, sharing state among microervices. It thus violates Amazon's original prohibition about shared memory. Sharing state is necessary for purposes such as distributed locks and, in the case we are discussing, recovering state in the case of failure of an instance.

The particular message that the instance was processing when it failed will not generate a response and so the client will retry the message which will be routed through the load balancer to a healthy instance.

Suppose the instance has not actually failed but is overloaded. Then it does not respond in a timely fashion and the client may send the message again. When coding a microservice, you should be aware of the possibility that a message may arrive twice. One possibility to protect against error is to code the microservice in an "idempotent" fashion. I.e. processing a message twice yields the same result as processing it once. In some cases, it is not possible for a microservice to be idempotent. Again, a distributed coordination service such as Zookeeper or Memcached can be used to share information about the state of each request among the instances of a microservice.

Now suppose that the microservice itself has failed. That is, all of the instances of the service have failed. In this case, the failure is recognized by the client. There are three actions it might take.

1. It could report the failure to the discovery service so that other services will not attempt to use the failed service.
2. It could set the "circuit breaker" in the circuit breaker pattern so that it does not attempt to call the service again.
3. It could attempt an alternate method for getting the service it wishes. This alternate method might possibly be degraded in some fashion but will result in some response to the client of the client of the failed service.

In any of these cases, it should record the failure in the log for monitoring purposes.

## Interoperability

Interoperability is about the degree to which two or more microservices can usefully exchange meaningful information via interfaces in a particular context. In this section, we are only concerned with microservices internal to a single application. The interoperability problem comes from the fact that there may be version incompatibilities. As we will see when we discuss deployment, it is possible that a microservice may have been updated, possibly adding a new feature or changing its interface, without its clients or its dependent microservices having been updated.

A microservice accommodates version incompatibility by being both forward and backward compatible. It achieves backward compatibility by never changing an interface but only extending its current interface. Figure 5.xx shows a translation layer between interfaces exposed to other microservices and internal interfaces. That is, suppose a new feature has been added to your microservice and this new feature requires additional information that previously was not included in your interface. Extending the interface to include this new information without changing the preexisting interface will allow those microservices that have not yet been updated to invoke your microservice without encountering an error. It will allow those microservices that have been updated to use your new interface. Having the translation layer allows you the freedom to structure the microservice in any fashion that makes sense and using the translation layer to convert from the external interface to the internal interface.

Forward compatibility means being graceful about calls to methods that do not exist. Instead of failing in some form or another, return a code that indicates that your microservice does not understand the call. Then the client can decide whether this is truly an error or just an attempt to use a feature that your service does not yet support. This also holds for calls you make to dependent services. A return that indicates that the method does not exist allows your microservice to take remedial action if necessary.

## Modifiability

Modifiability is the ability of a microservice to be conveniently and easily changed. Change is a constant for any long lived system and most of the work of software engineers these days involves making changes to existing systems. Coupling and cohesion are the traditional measures of modifiability. The goal when making a change is to limit the number of microservices that are impacted by that change. You want to have low coupling between any two microservices and high cohesion within a microservice. What this means is that each microservice should provide a well defined function and that function should have minimal overlap with functions provided by other microservices. Overlap of function will cause multiple microservices to be modified when one of them is changed.

The proliferation of services within a microservice architecture adds another element to consider. When a change is to be made, finding the microservices affected by that change becomes more difficult. Your organization must have a process for allocating changes to microservices. This process will involve maintaining a catalog of microservices and their functions but it will also involve people with an overall view of the system and knowledge of the interactions among the microservices. Netflix, for example, has a team whose responsibility it is to coordinate among the microservice owning teams. This involves allocating functions or modifications to individual microservices but also coordinating interface descriptions and assumptions.

## Performance

Performance refers to the ability of the service to process requests. There are two fundamental measures of the performance of a microservice – latency and throughput. Latency measures how long it takes to respond to a request and throughput measures how many requests are processed in a given amount of time. Both can be measured directly by the microservice by using the internal clock. We said earlier that the clock may very across two distinct devices. Since a single instance is on a single device, the measurements taken from the device's clock are consistent.

In addition to the time taken by the microservice, there are two other times that are important

1. The time taken by messages over the network and
2. The time a message spends in a queue prior to being processed.

All of these times can be directly measured. In addition, the time spent marshalling and unmarshalling mmessages contributes to the overall latency and throughput of a microservice. Finally, there will be time spent in overhead functions such as discovering the IP address of dependent microservices, sending health checks, and so on. All of the times being measured should be logged for monitoring purposes.

Two questions come up at this point

1. What should the latency and throughput be?
2. How do I improve the values if they are not being met?

Setting an appropriate desired latency value depends on what kind of microservice you are responsible for. Microservices that manage data have different characteristics than microservices that compute functions. There is human factors data that end users perceive 200ms as instantaneous. If your mmicroservice is in the direct line of responding to a user request then you need to determine a) is this the kind of request for which the user expects instantaneous response? And b) how many smicroervices are involved in responding to the user request?

Users have expectations about the latency of responding to one of their requests. Some requests seem trivial and so should have low latency. Other requests seem complicated and users are willing to take non-instantaneous responses. If a request goes through a large number of microservices, then the budget for each service is small. You can also use the history of your microservice, or similar microservices, to determine your latency budget.

If you are not meeting your time budget, the first step in improving the latency of your microservice is to determine where time is being spent. It is usually a waste of time to optimize a portion of the microservice that is only contributing 10% of the time being taken. Focus on the places where substantial portions of your budget are being spent. Remember that processors and disk fail or partially fail. Try your microservice on a different device to ensure the problem is in your code, not in the hardware.

We defer the discussion of several related issues: where are the measurements recorded and how does the fact that there are multiple instances of your microservice executing affect your performance measurements. We will discuss these issues when we discuss logging in Section xx.xx.

## Reusability

Being able to reuse code is, in many ways, a Holy Grail for software engineering. But like many other absolutes the desire to reuse code must be tempered.  The virtues of reuse are that the code does not need to be redeveloped, thus saving time during development, and there is a single point of correctness, thus saving time during repair.

We distinguish between large grained reuse which is encompassed in the architecture and small grained reuse which is reuse of sections of code. Each microservice is owned by a single team so for that team to reuse code from a different team (small grained reuse) they must discover the code, verify it is suitable for their anticipated use and, if not, adapt it. All of this takes time. Depending on the size of the code to be reused, it may be faster and more expedient to develop the code independently.

When designing a microservice architecture, you will make a decision whether to have large or small fan out. Large fan out means that each microservice has a large number of children and each request chain is short. Thus, the amount of message passing is reduced and so this design favors performance. Small fan out means that each microservice has a small number of children and each request chain may be long. This design allows for choosing the microservices in the chain to maximize reuse. Thus, the trade off is between performance and reusability.

The single point of correctness argument makes sense if the computations are mission critical, such as how much is owed on a particular account. For mission critical computations, there should be a set of test cases that are used to test mission critical computations. Thus, if a team chooses to replicate mission critical computations, the replication is subject to the same test suite as if they had reused the mission critical computation from a different team. Thus, the responsibility for correctness of mission critical computations lies in the test cases, not in the implementation.

## Scalability

Scalability is the property that a microservice can add resources to serve more requests. In the microservice context, scalability means “scaling out”. That is, the additional resources come from adding additional VMs, not from getting larger and more powerful VMs. As we discussed previously, adding new servers for a service is easy if the service is stateless and more difficult if the service is stateful.

## Security

Security is best remembered by the acronym CIA – confidentiality, integrity, and availability. Confidentiality means that information can only be seen by authorized users. Integrity means that information can only be modified by authorized users, and availability means that the service is available to authorized users.

Security is a complicated subject with many subtleties. We discuss security in more detail in Section xx.xx. For now, we enumerate some practices you, as a microservice developer, should use.

- Use https instead of http. https encrypts traffic sent over the internet to and from web servers and since that traffic is open to eavesdropping, it should be encrypted.
- Apply patches promptly. Most complicated software has security vulnerabilities and these are always being discovered and patches released by the vendors. You should apply these patches promptly so prevent the vulnerabilities from being exploited by bad guys who monitor newly discovered vulnerabilities. CVE (Common Vulnerability Enumeration) is a data base that lists known vulnerabilities in a variety of software packages.
- Delete unused resources promptly. It is easy to lose track of virtual machines in the cloud. VM sprawl is the term given for having so many VMs that you lose track of them. If you lose track of a VM, then it does not get patched and its vulnerabilities remain. Breaking into an unpatched VM can provide access to sensitive information or to credentials that can be used to break into active VMs.
- Do not write security sensitive code such as password managers yourself. As we said, there are many subtleties associated with security sensitive code and there are certified packages that provide security services.
- Do not embed credentials into scripts. We discuss credential management in Section xx.xx but embedding credentials into scripts makes it difficult to modify the credentials and provides access to anyone who gains access to the script. Placing a script with credentials into Github, for example, happens frequently enough so that there are monitors that check submissions and warn of credentials being contained in these submissions.

## Service Leve Objectives

Every service should have one or more Service Level Objectives (SLOs). These are different from Service Level Agreements (SLAs). SLOs are for internal monitoring purposes. SLAs are external agreements with clients. You want the SLO to be more constraining than the SLA. This gives you some breathing room if you violate the SLO. The SLOs are monitored on some periodic basis. The period will be shorter when a new version of the service is placed into production (on the order of minutes) and longer when the service has been operating successfully (maybe once a day).

An SLO is either an availability or a performance value. Having an SLO determines the basic level of recording that comes from the service. More detailed recording can be turned on to gather data about unsatisfactory performance but the base level is determined by having an SLO. The measures for the SLO should be made directly as we will see shortly. That is, if an SLO has to do with latency, then measure latency, do not use an indirect measure.

Some SLOs are:

- Latency and throughput. The time between a message arriving at a service and the response being returned. Recording time of arrival and time of response allows the calculation of both latency and throughput.
- Request satisfaction rate. Recording a request on arrival and whether it was satisfactorily served on response is an availability measure.

Note that these SLOs can be determined by recording information at message receipt and message response. Generating too many recordings will have a performance impact and generating too few will not give you the information you need to understand how well the service is performing.

The SLOs above are not the only possible SLOs. The appropriate SLOs depend on what type of microservice you are developing. Customer facing microservices use latency as an SLO and big data microservices use throughput.  The appropriate SLO is often a matter that the business cares about and so input from the business portion of your organization is helpful.

We will return to SLOs when we discuss monitoring in Section xx.xx. Monitoring software uses the data recorded by the microservices, among others, as basic input.

## Microservices in context

Microservices do not exist in a vacuum. They have particular functions they must perform in addition to their business logic function. They must interact with other microservices. They must be designed to support rapid deployment and to protect themselves against certain types of cloud failures. These are all topics we discuss in this section. We begin with data interchange.

### Data Interchange Protocols

Because applications in the cloud are dependent on message passing, how these messages convey application level information is important to the success of an application. In this section we briefly review some data interchange protocols. We present the protocols in the order of their emergence both because a particular protocol is in many ways a reaction to the deficiencies of its predecessor and because these protocols are still in use today.

#### REST

Representational State Transfer dates from the origins of the World Wide Web. It was used in HTTP1.1. The key elements of REST are

- The calls are stateless. That is, there is no assumption in the protocol that any information is retrained from operation to operation. This means that either each operation contains all of the information necessary to act on that operation or it is the responsibility of the two parties to decide what sate will be maintained where. In the former case, calls tend to contain a lot of information. Stateless calls support scalability and availability as we have discussed previously.

- The information exchanged is textual. The web from its inception was designed to be heterogeneous. Not only across different computer systems but across different binary representations of information. Textual representation is still a necessity when different organizations are involved but within a single organization, it is possible to have an organization wide binary representation.   Binary representations are more compact and, hence, more efficient to transmit over a network.
- The most common operations are GET, POST, PUT, and DELETE. These map into the database concept of CRUD (Create, Read, Update, and Delete). The textual portion of a REST operation can refer to any resource that has been assigned a name such as web pages, e-mail, or information from a data base.

## XML

Extensible Mark up Language. A problem with REST is that all of the parameters must have an agreed upon meaning between the sender of an operation and the recipient of that operation. Some simple concepts become very complicated when investigated. For example, does an address consist of a number, a street name, a city, a state, and a zip code? Maybe or maybe not. The authors have variously had addresses that consisted of RFD name, pole number (telephone pole) or PO Boxes. And that is just the US. Other countries have their own idiosyncrasies. Defining a parameter in a REST operation as an address is open to ambiguity unless there has been an agreed up meaning of that parameter. Enter XML.

XML is a tagged language similar to HTML. That is, it has a defined set of language markups and content. A markup has a tag associated with it and the content is between the opening markup symbol and the closing markup system. For example, <street>Elm Street <street/> defines the tag "street" together with the content "Elm Street". The < and /> are markup symbols defined by the XML standard. Tags may be nested to generate compound structures.XML is designed for the interchange of documents. Consequently, structure is imposed via the tags.  Figure 5.xx gives an example of an address expressed in XML.

As you can see, XML partially solves our problem with addresses. You (an XML parser) can parse the XML and display its structure. Thus, the fact that an address is defined to be a number, a street, a city, a state, and a zip code is apparent. Addresses expressed as P.O. Box numbers or otherwise are excluded. They must be represented with other tags. Shortly after the introduction of XML a number of standards emerged that describe the vocabulary to be used in data interchange within a domain. The vocabulary used in mining, for example, is different from the vocabulary used in textiles.

A standard called WSDL (Web Services Description Language) is used to describe the functionality provided by web services. A web service has endpoints or ports and an external file is used to specify its parameters and the types it expects. WDSL is often used in conjunction with SOAP (Simple Object Access Protocol) that describes the objects being transmitted.

It is important to realize that WDSL and SOAP together with domain standards are intended to totally describe an interface to an SOA service. That is, there is no need for a client to communicate with the owner of a service to gather information about that service. An SOA service may be owned by an organization totally distinct from the organization that is the client for the service. Hence, communication about the details of a service is hindered by organizational boundaries and constraints.

## JSON

Java Script Object Notation is used to transmit attribute value pairs and array data types. Since XML is intended to describe documents (unstructured data) the transmission of structured data such as arrays or attribute value pairs requires more characters to be transmitted than if the structure was built into the notation itself. JSON format has language constructs to describe structured data and is widely used to communicate between web services and a browsers.

One use of JSON is to allow stateful communication between a browser and a web service. The protocol itself is not stateful but it assumes that the information being transmitted will be used to update objects residing in the recipient. These objects are where state is kept.

The data types that JSON supports are those familiar to users of programming languages – number, string, Boolean, array, object, and null. The built in data types make it easy to parse JSON strings into programming language constructs. Figure 5.xx shows our address example expressed in JSON.

## Protocol Buffers

Protocol Buffers build on JSON by allowing you to define a type structure for the transmission of data. JSON allows only the basic types we have enumerated. Protocol buffers allow you to define your own types, through constructs similar to type specifications in a programming language.

Type specifications are recorded in a file called a proto file. This proto file is interpreted both on the client side and the microservice side so that the transmission can occur. Protocol buffers are typically packaged with an RPC mechanism such as gRPC which we will discuss in the next section.

## Remote Procedure Calls

When one procedure calls another, it provides the name of the procedure it is calling and the parameters for that call. It would be convenient for the microservice developer if they did not have to be concerned about whether the called procedure was local sharing the same address space or remote on a different device on the network. This observation was made when networking became prevalent and that is the motivation for Remote Procedure Calls (RPCs). The microservice developer codes a procedure call and when it is executed, the infrastructure does the necessary manipulations to pass the parameters to the remote device and invoke the called procedure on that device. The call can be synchronous and block the calling procedure from other computations until a result is returned or asynchronous. In an asynchronous call, the calling procedure can perform other calculations while the called procedure performs its computations.

As we move into the microservice world, instead of the procedure world, the caller is a micrroservice as is the callee. There are RPC mechanisms that are local to the calling service to prepare the message for transmission and RPC mechanisms that are local to the called procedure to receive and translate the message into parameters for the called service. The terms “marshalling” and “unmarshalling” are used to describe the process on each side. Figure 5.xx shows how this works. The calling microservice invokes its local RPC marshalling procedure which packs the message and sends it to the receiving microservice. The receiving microservice routes the message to the RPC unmarshalling procedure which unpacks the message and invokes the called procedure. Replies perform this sequence in the reverse order.

Over the years, a variety of different RPC protocols have been developed. The most common one is now gRPC. gRPC is packaged with protocol buffers.  The specification of the service to be called is placed in the .proto file. A proto compiler processes the .proto file and generates the code for both the calling service and the called service. The generated code is then linked into the microservice to provide the functionality shown in Figure 5.xx as well as the marshalling and unmarshalling code for the protocol buffer. Since the .proto file contains the service name and its parameter descriptions down to the types of the parameters, the called microservice can be coded as a standard procedure.

## Platform as a Service

In Chapter 3 we discussed the cloud in terms of Infrastructure as a Service (IaaS). When the cloud offers more services than just management of the infrastructure, it is called Platform as a Service. We have alluded to many activities that a microservice must perform in addition to its business function. These include registration, logging, and discovery, among others. Now we will enumerate a partial list of services that a microservice might use. Organizations should standardize on this list and embody them as either additional services or as frameworks. In this case, the arguments for reuse are compelling. The services are common across multiple microservices, they can be documented and, thus, made easier to use, and they provide a single point for the improvement and correction of the services. Common services that should be in the PaaS for microservices are:

- gRPC. We discussed gRPC and protocol buffers in section xx.xx. Placing them in the PaaS makes these features easily usable from within a microservice.
- Discovery. Discovery allows a microservice to locate the IP of a dependent service. It invokes the discovery service with the dependent service's name and is returned an IP. The IP could be the service directly or it could be the IP of the load balancer currently managing the instances of the desired service.
- Registration.  Registration is the reverse side of discovery. A microservice registers with, potentially, two different services. First, is the load balancer. Recall that the instance/load balancer connection can be made either by the instance or by the instance creation service,  e.g. the  autoscaler. The second service that a microservice must register with is the discovery service. In order to be discovered, a microservice must register with the discovery service. Three services – load balancing, discovery, and registration – can be combined into a single service. The open source package Eureka from Netflix includes such services.
- Configuration. Configuration parameters are tailorable portions of a microservice. These parameters include settings for the cloud such as security settings and settings for accessing external services. They also include settings for internationalization such as language and color usage. Having a single service in the PaaS responsible for acquiring configuration parameters is one method for supporting uniformity across different microservices. We discuss configuration parameters and how to manage them in more detail in Section xx.xx.
- Distributed coordination service. In Chapter 3 we discussed the distributed coordination problem. The provision of a solution to this problem should be a portion of the PaaS.  Any time you need to synchronize with another microservice or when two instances of your microservice need to share state, you should do it through either a distributed coordination service – for small amounts of state – or through  persistent storage – for larger amounts of state.
- Logging. A common logging service is used to provide uniformity in the logs. A log message should include identifying information such as source id, task id, time stamp, and log code. The

logs are sent over the network to a log repository.  Placing this service in the PaaS helps ensure that all the expected information is provided and the logs are generated in a common format.

- Tracing. Different circumstances call for different levels of insight into the action of the microservice. The tracing service allows for these different levels to be turned on or off at runtime.
- Metrics. Every microservice should have defined SLOs. These represent the key metrics that determine whether the microservice is performing as it should. This PaaS service will receive the metric values. It interacts with the dashboard service which is responsible for displaying the metric.
- Dashboards. The dashboard displays the metrics collected about the behavior of the microservice with respect to its SLOs. Making this common across all microservices ensures the information is displayed in the same fashion regardless of its source and assists the developer or other people who monitor the microservices in interpreting the output. For example, the values of an SLO can be coded as red, green, or orange to reflect acceptable, unacceptable, or borderline values. The dashboard service is informed of these values and displays them consistently for all microservices.
- Alerts. An alert results in a page. Setting the values at which pages are sent allows the microservice to monitor itself and inform relevant personnel when an event needs immediate attention.

We discuss monitoring in more detail in section xx.xx.  From the point of view of a microservice, the connections to these monitoring functions are available in a PaaS.

## Microservices and Containers

Although containers and microservices evolved independently, they are a natural fit. Microservices communicate only with messages, containers have IP addresses that are intended for message based communication. Furthermore, the container orchestration mechanisms that are evolving are spurring the adoption of microservices.

Containers provide more limited resources than do VMs. Microservices, since they are small and single purpose, typically require fewer resources than multifunction processes.

## Designing for deployment

As we will see in Section xx.xx, it is possible for there to be instances simultaneously active that are running different versions of the same microservice. “Feature toggles” are a mechanism to allow different versions of the same microservice to be simultaneously active. Feature toggles are “if” statements that make the new code for a microservice version conditional. The condition is that the feature toggle is on. If the feature toggle is off, then the old code of the microservice is executed. See Figure 5.xx for a template for feature toggles.

The key deployment idea is to differentiate between “installing” a new version of a microservice and “activating” that new version. The new version is installed with the feature toggled off and when it is time to activate the feature it is toggled on. The value of the feature toggle, whether toggled on or off, is maintained by a distributed coordination system so that all of the instances of the microservice are toggled on – or off – at the same time.

Feature toggles clutter up the code and made it difficult to understand so a feature toggle should be removed from the code as soon as the new feature is stable in production.

We return to this topic in Section xx.xx when we discuss deployment possibilities.

## Protecting against failure.

Two kinds of failure are possible due to cloud characteristics. First, the host for your instance may fail. We have discussed dealing with this type of failure above in our discussion on availability. Secondly, and the subject of this section, is the long tail phenomenon. In Chapter 3, we discussed the long tail. Knowing that this possibility exists, we now discuss how you, as a microservice developer, can protect against it.

- One technique is called "Hedged requests". Suppose you wish to launch 10 instances of a microservice. Issue 11 requests. Terminate the request that has not completed when 10 are completed.
- A variant of this technical is called "Alternative" request. In the above scenario, issue 10 requests. When 8 requests have completed issue 2 more. Cancel the last 2 to respond.

Note several characteristics of these two techniques. First, they are asking for a cloud service. "Launch instance" in the example but it could be any service provided by the cloud. The more hops that your message goes through, the more the likelihood of the long tail. If you are asking for a service from some portion of your application, the number of hops will be small unless that portion of the application is in a different availability zone or different region.

Secondly, there is no harm done if an extra instance is launched before the cancel takes effect. The service you are requesting should be "idempotent". Writing a data item to a remote data store for replication purposes, for example, is idempotent. Writing two copies of the same data item causes no harm since the second write will overwrite the first with the same value.

## Summary

Microservices exchange most information via messages. This is an inherent portion of their definition. The exception is when information is shared through a distributed coordination service or through persistent storage. A consequence of this reliance on message passing is that data interchange protocols are important. Protocol buffers are a structured method to pass strongly typed data structures. gRPC is the latest remote procedure call protocol and is widely used among internet companies. A proto compiler exists to package protocol buffers and gRPC into libraries for your microservice and dependent microservices.

Your microservice relies on a variety of additional services that are provided to you as a Platform as a Service within the cloud. These additional services include discovery and registration, distributed coordination service, the gRPC library, and logging and monitoring services.

Packaging your microservice as a container allows your microservice to be managed by one of the container management systems that we discussed in Chapter 2. It also provides a much lighter weight footprint than packaging your microservice in a virtual machine. Containerization is fast becoming the packaging mechanism of choice.

As  a designer of a microservice, you need to do special things to prepare for particular deployment processes. Feature toggles is a mechanism that you can use to do this preparation.

You also need to be aware of the possibility of a long tail for your cloud requests and hedge against that possibility through issuing extra requests that are cancelled if they are not needed.

## Exercises

These exercises use an assignment that you have written for another class. Choose such an assignment

1. Package your assignment as a microservice.
2. Define protocol buffers for your microservice
3. Write an invoking routine that is packaged as its own microservice and invokes the microservice from exercise 1 using protocol buffers.
4. Write a simple discovery service that stores key value pairs where the key is the name of a microservice and the value is its IP address.
5. Register the microservice from exercise 1 with the discovery service
6. Use the discovery service from the microservice your wrote in exercise 3 to invoke your microservice

## Discussion

1. An RPC call is routed to a port on the receiving side. This port is dynamically assigned. How does the mmessage get routed to the correct port?
2. Why does the fact that a microservice is independently deployable mean that its location must be discovered at run time?
3. What problems can arise with an application designed around microservices and how would you solve these problems?
4. Develop a context diagram for Eureka (the combination load balancer, registration, and discovery service released as open source by Netflix).