Exploration of orchestration challenges in a microservice-based application

Luca Marchiori
luca.marchiori.3@studenti.unipd.it
University of Padua
Runtimes for Concurrency and Distribution 2023-2024

Figure 1. Seattle Mariners at Spring Training, 2010.

Abstract

The following is a technical report related to the course, "Runtimes for Concurrency and Distribution," held by Prof. Tullio Vardanega at the University of Padua. This report aims to summarize what was learned when approaching the concepts of microservices and orchestrations, and highlight the discoveries made by implementing a proof of concept practical example.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Luca Marchiori. 2018. Exploration of orchestration challenges in a microservice-based application. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, **??** pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Problem statement The selection of this project, among others, was driven by the desire to dedicate time to exploring the microservices architecture. Microservices, like many other paradigms in the past, appear to be the (almost) new and shiny approach that developers seek to adopt. However, determining whether the microservices architecture is a good fit for a specific problem is a completely different matter and requires a good understanding of both technologies involved and the concepts behind the architecture.

Starting from the definition of microservices and reviewing the course material, I have adopted an exploratory approach to understand different areas of this architecture. I began by implementing virtualization and containerization, and then moved on to container orchestration, service state management, and finally, communication between services. implementing both synchronous and asynchronous communication styles.

Introduction After reviewing numerous resources, both online and in books, my focus has been centered on three different aspects that are included in every definition. These aspects are: Independent Deployability Communications Business domain logic

Each aspect covers multiple concepts. When exploring how microservices can be independently deployed, the principles of encapsulation, information hiding, and state management have been at the core of my experiments.

Communication plays a crucial role in defining how microservices can interact with each other and how this impacts the final responsiveness of the systems. This includes both the style of communication (synchronous, asynchronous) and the technology used (HTTP, RPC, etc.).

The selection of a business domain for a microservice is more conceptual than technological. It requires a holistic view of the entire architecture and meticulous planning to ensure that no business domain is shared or redundant. It is also necessary to ensure that each microservice can effectively address the needs of the domain it is designed for.

The recurring concept of scalability covers all of these three aspects, starting from the technological choices, up to how microservices can be managed effectively by multiple teams and make the development also scaling with the company. More on this in the following sections.

Experimental implementation During the exploration of these concepts and related technologies and tools, some basic components of a potential architecture have been developed as experimental examples. The aim was to implement

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

components commonly found in any software, such as user management and authentication.

Containerization The project has begun with the study of Docker, one of the most famous containerization platforms available today. Containerized microservices are crucial for ensuring the microservice's independence from the underlying infrastructure and for facilitating easy deployment. This is achieved by encapsulating everything needed to run the software logic into a single object, ranging from a minimal version of an operating system to runtimes, libraries, and storage volumes.

With containers, it is possible to create multiple execution environments that are completely isolated from each other, without the overhead of a hypervisor. This is achieved by leveraging a single kernel shared between containers and abstracting the processes of the individual virtualized environments. With tools such as Docker Swarm and Kubernetes, managing hundreds of containers running on different physical servers becomes possible. Indeed, containerization is at the base of the Independent Deployability property of microservices.

For the practical implementation, a small web service has been created that provides a handler to accept HTTP POST requests and create a user profile with the provided information. After this, a completely containerized version of this web service was developed.

The first thing I created was a DockerFile with a multi-stage build process. This type of DockerFile allowed me to containerize the compilation of the microservice, making it completely self-contained. The first stage, responsible for the actual build, uses a base image optimized to include all the necessary GO tools for the compilation of GO software. After downloading all the dependencies and required libraries, the actual compilation process begins. In the end, the binary is transferred to a more essential base image that uses Alpine Linux as its OS and simply runs the already created executable file. The multi-stage feature allowed me to compile the service in a heavier environment with all the necessary tools for compilation, and then transfer the binary to a minimal image that can be moved and deployed with minimal resources. Since the web service is set to provide service on port 4000, a declaration in the DockerFile is included to expose that port.

The created image needs to be loaded into an image registry to make it available to other deployment systems like Kubernetes. A container registry is a repository (or collection of repositories) used to store and access container images. These can be hosted locally or on a cloud provider and provide storage to have all the images in one place that can be pulled by other software. The registry also provides a way of having version control to catalog all the different versions of a piece of software and security mechanisms to ensure that only authorized entities can access the images. Since cloud providers that offer image registries often require payment, I chose the Google Cloud Artifact Registry that has a free plan for testing purposes.

The process of configuring Docker to interact with Google Cloud has been quite difficult due to the numerous security policies needed to ensure authentication to the system and authorization of operations. After reading some documentation and installing the required packages by Google to enable some Docker configurations, I was able to push and pull images from Google Cloud Artifact Registry.

Since the application also needs a database connection to store and retrieve users, a new container built with a PostgreSQL DB image has been created. Thanks to a Docker Compose file, it's possible to configure the container of the application logic and the container with the database instance to communicate and run together. Of course, the database image needs to have a persistent state in order to keep the data across restarts. This is a problem that I have encountered many times during the project implementation, and I will focus on it in the next sections. For now, the solution is simply to map a folder on the host machine to the one that the containerized database uses to store data, thanks to the configuration of volumes in the docker-compose file.

Container limitations The solution developed so far has been working well and has had no issues handling the requests made by me and some tools such as curl and Swagger. In fact, many services are still deployed in this manner and are perfectly suitable for non-critical operations, test environments, or very controlled production environments with few users. However, it would be careless to adopt the same deployment strategy for a critical, high availability service that needs to be operational 24/7 with a large user base. Here are some of the most evident limitations:

Scalability: One of the limitations is the inability to scale the container dynamically. Since it is hosted on a single server, it cannot be replicated or have its resources scaled up to accommodate higher demand. The only option would be to manually move it to a more powerful server or upgrade the server itself. However, this process cannot be done dynamically to optimize resources and service costs depending on the real demand. Additionally, during the upgrade, there is a risk of downtime and potential compromise of the internal state.

Updates: In order to update the microservice, the container needs to be stopped, updated, and then started again. The issue here is not only the downtime but also the lack of available strategies in case the update encounters problems. Until the developer realizes that the update is faulty, users may have access to a flawed version of the software, which could potentially be dangerous for the whole system. Ideally, we would like to have updates that can be performed without any downtime, and have strategies to detect errors and rollback to the previous version without users being aware of it.

Errors: Although it is possible to define the behavior of a container in the event of a software crash (by setting the restart property), it can be challenging to define custom mechanisms to handle container failures. Additionally, the definition of failure itself requires careful attention, as we would want to restart the service not only in the event of a software crash but also in cases of unresponsiveness, custom errors, or timeouts. Furthermore, the restarting of a single container also takes time, during which the service would be unavailable.

Networking: When dealing with multiple microservices that need to be connected and exchange data, Docker provides limited control over networking by using only a virtualized internal network. Manually defining network links and addresses can be challenging, and failures can occur due to the dynamic nature of web services that may change IP addresses or be moved across servers. Additionally, we would want to define links that are accessible to users for accessing the services, while keeping internal links hidden from the public and used solely for internal communications.

Deployment: The manual installation and management of Docker containers on specific nodes can be challenging and require constant effort to ensure they stay online and error-free. Ideally, we would prefer to have a centralized method for managing the deployment of services across multiple nodes. This would simplify the process and make it easier to maintain and monitor the containers effectively.

Those are all desirable features that can only be achieved by using orchestration software, which provides automated deployment, coordination, and management of services. It's important to note that orchestration is not limited to microservices alone, it is a broader concept that can be applied to a wide range of distributed systems and application architectures. In the context of microservices, orchestration plays a crucial role, but it can also be used in other types of systems. One of the most well-known orchestration platforms is Kubernetes.

————

Kubernetes PODS DEPLOYMENT Services Since pods can be created, deleted and restarted both as a mechanism to recover from failures and as a way of dealing with updates and scaling, the ip addresses of the pods associated with a deployment change continuously. Configuring a cluster using fixed ip addresses would simply not be feasible and indeed is a big problem when dealing with orchestration of services. To solve this problem, kubernetes services use selectors to bind to other objects identified by labels. An internal DNS mechanism allows translating the binding between labels and selectors to actual ip addresses. Thanks to this the routing of connections between pods and nodes hosted on different servers can be abstracted and configured simply by using labels.

NODE PORT CLUSTER IP

Users Microservice The Users Microservice defines different handlers for the management of a user base. All the handlers use HTTP and the REST (Representational State Transfer) paradigm and are exposed on port 4000. POST: /users-ms/users receive a json containing name, surname, email and other user information to store in the database GET: /users-ms/users/id receive an Id used to retrieve the user with the corresponding id. GET: /users-ms/users return the list of all users. Additional parameters such as email, name, surname, can be passed to filter out the results. PUT: /users/id retrieves a user with the corresponding id passed as parameter and updates its information with the data provided. DELETE: /users/id simply deletes a user with the corresponding id.

An architectural choice needs to be made to define how the internal state of the microservice is handled. This state needs to be persistent to allow the stored users information to persist both across multiple requests and multiple restarts of the service. A database is a common solution to this problem.

As stated before, microservices should be self-contained and hide their internal state so that any mechanism to manage the internal state is considered to be hidden inside the microservice. At the same time we would want to have microservices independently deployable and to implement horizontal scaling through replication. This leads to the possibility of having multiple instances of the same microservice running at the same time. Having multiple instances of a service allows it to handle more traffic and improve the reliability of the microservice as you can more easily tolerate the failure of a single instance, restart it and divert the traffic to other instances at the same time.

In this case, the assumption of having one database included in each instance is completely wrong and can lead to distrasours outcomes in terms of state (and data) integrity. A desirable way to cope with this situation is to have the database in a single instance accessible only by all the instances of the microservice related to the database. For this reason, I proceeded by creating a new microservice, "Users DB Microservice".

The basic configuration of Kubernetes (a.k.a K8s) for this microservice involves the creation of a yaml file that defines three different objects: a development, a node port service and a cluster ip service. The deployment object is in charge of pulling the image from google cloud and deploying multiple pods depending on the specific configuration. The node port service maps the port 4000 of a pod to the port 30400 of the host. This configuration allowed me to directly access the service for testing and debugging purposes without having to pass through the cloister ip service. The cluster ip service assigns the port 4000 to the deployment in order for the microservice to be accessed internally by other objects of the cluster.

Users DB Microservice This microservice, as stated before is used by the users microservice only to keep a persistent state of the users information. The deployment object pulls an image of postgreSQL directly from the official image registry (in this case, Docker Hub) and by using environment variables configures the database in terms of name, user and password used for the access. Since the maximum instances of the database should be 1 at any time, the replicas parameter is set to 1. As before, the node port service allows the direct connection to the database pod for development purposes and the cluster ip service allows the connection to the database from other nodes in the cluster. In this case, only the user's microservice is configured with the right environment variables to access the database. Thanks to the service discovery mechanism, there is no need to configure the deployments with specific ip addresses. The database deployment is identified by the "users-db" label and thanks to the cluster ip service and the service discovery mechanism, the users microservice accesses the database by connecting to the endpoint "users-db-cluster-ip-service".

To allow persistence across service restarts, a PersistentVolume object is attached to deployment by using a pre-defined PersistentVolumeClaim. The ReadWriteOncePod policy is used so that the volume can be mounted as read-write by a single node and a single pod, in this case, the single instance that is running the users database.

It is evident that the use of a DB instance of this kind, has no way of scaling horizontally and can be only be upgraded by assign more resources to it. It is true tho that the microservice architecture itself help with the scalability of the database because the problem domain relative to the whole state of the system is already splitted in many different databases that can better handle the traffic. Anyway, there is still the possibility of having a microservice so important that still the database linked to it needs some way to scale and where this implementation can be a bottleneck. The first solution, is to delegate the management of the database to a third party service,usually a cloud provider that is responsible to scale the database depending on the traffic and the requirements for that service. This can introduce an economical cost for the whole system but it can make easier for small development team to make the service scaling up.The alternative is to adopt a distributed database that can be managed by kubernetes itself or other systems. By looking for documentation, i discovered that Zalando, the well know online shop, has deployed a Postgres Operator for kubernetes that supports functionalities such as load balancing, cluster deployments, rolling updates for db schema, point in time recovery and many other. The operatori is completely open source and available for implementation.

Auth Microservice As a way to test out the communication between two microservice, the Auth Microservice has been created. This service has just one handler (POST: /authms/auth/login) that receives a username and a password and returns a token if the credential matches a user registered in the system. The underlying idea was to implement a toy example of a standard authentication system that works through tokens. These systems works by generating a token to identify and authorize the subsequent request after a user has been authenticated using credentials. In this version, the auth microservice sends a GET request to the user's email of a user. If there is one, the users microservice returns the user and the auth microservice check is the provided password and the one associated with the users matches. If so, a random token is generated and returned as a response. Agin, the connection between the two services is handled by the internal service discovery mechanism, and the configuration do not involves the management of actual deployment addresses but just the use of labels. As a safety rule, the password should never be store in clear text, nor they should be transmitted on unsafe connection. In my implementation the password is hashed by bcrypt, one of the safest password hashing tools as soon as it is received by the microservice. The match between the password store and the one provided is also done by a function of bcrypt. Note that to keep the business domain of microservices separated, the auth service do not access directly the database, instead it simply ask the users service for the retrieval of the user. THis way, each microservice is just responsible to manage its functions and all the work behind is hidden and managed by the mciroservice itself. Also, the GET request to the userś mciroservice is synchronous and blocking: the auth microservice cannot do anything until it gets a response. Even tho synchronous connection can pose limitation in terms of systemś responsiveness, in this cane the use is totally fien since the retrieval of a user is a critical operation for the authentication and nothing can be done until it has completed successfully. In order to test out also an asynchronous communication technique, in this case, event-based, i decided that from a system like this, we world want to send some kind of notification to the user whenever a new login is triggered. Here the communication can be asynchronous since we do not want to wait until a notification is actually sent, we can just trigger the event and let some other piece of the system to handle this event. Using RabbitMq library, whenever a login happens, a new message is sent to an exchange using the mailbox style of communication. The notification microservice is in charge of processing the message queues and actually sending the notifications.

Asynchronous communication With asynchronous messaging, the act of transmitting a request over the network doesn't halt the microservice initiating the request. It can proceed with any other processing without waiting for a response. Within this project, event-driven asynchronous communication is employed to facilitate the delivery of notifications upon user login. With the microservice architecture, we aim to minimize the coupling of components as much as possible. Asynchronous communication aids in avoiding temporal coupling between the authentication and notification microservices. Temporal coupling refers to a situation

where one microservice requires another microservice to to do something at the same time for the operation to complete. In non-blocking asynchronous communication, the microservice making the initial request and the microservice receiving the request are temporarily decoupled such that the microservices that receive the request need not be reachable at the same time the request is made. In this project, the event-driven asynchronous communication is implemented using RabbitMQ, a popular message broker. RabbitMQ is installed inside the K8s cluster as a cluster operator such that the cluster created so far is extended to automatically manage the message broker. Inside the auth microservice, whenever a login happens, a message containing the login information is sent to an exchange that can forward the message to different queues depending on some criteria such as routing or topics. In this case, the echanche is identified by the name “user$_login_event”andthechosenroutingstrategyis“fanout”, meaning$[illegible]$exchange.Theexc$
$side, inthiscasetheNotificationmicroservice.Sincewewoldwantto$[illegible]$ofeventandthe$
$theexchangeisnamedtoindicatethetypeofeventtriggered(e.g.user$[illegible]$ekindofmessa$
$exclusivesothatmultipleinstancesofthesamemicroservicecanela$[illegible]$rtsto“consume”thequeu$

Updates strategies Kubernetes allows to automatize the updates of images by simply changing the desired image version in the deployment configuration. If not specified the strategy field in the deployment configuration, the rolling update strategy is used with the default values of maxSurge and maxUnavailable set to 25

Other updates strategies are available but not implemented in the project: Blue-Green deployment and canary deployment are one of those.

"Blue" and "green" simply refer to the two versions of the deployment. The blue version is the current version of the deployment, and the green version is the new version of the deployment. With this strategy, two separate deployments (one for the blu version and one for the green one) are created. At first the service points to the blue deployment. The green deployment is created and tested. Once the green deployment is ready, the service is updated to point to the green deployment and the blue deployment is then deleted. If something goes wrong, the service can be easily rolled back to the blue deployment. While the green deployment is up but not yet connected to a service, it is possible to test it and make sure that everything is working as expected. As opposed to the rolling update strategy, all the clients are switched to the new version at the same time, so there is no risk of having too few pods running the new version at first. This can be useful for applications that always require to distribute the load evenly among the pods.

The Canary strategy is similar to the blue-green deployment, but instead of switching all the clients to the new version at the same time, only a small percentage of the clients are switched to the new version. This is useful to test the new version in production and to make sure that everything is working as expected before switching all the clients to the new version. It is possible to implement it by having both the blue and the green deployments running at the same time. Both the blue and the green deployments are connected to the same service. By adjusting the number of replicas of each deployment, it is possible to control the percentage of clients that are switched to the new version.

Using the latest tag for Docker images in Kubernetes environments is generally discouraged due to issues with stability, predictability, and control. The latest tag lacks clear version control, making it difficult to track and roll back versions, and can lead to inconsistent deployments since the tag can point to different images over time. This unpredictability complicates debugging and reproducing environments across development, testing, and production stages.

Scaling strategies

Health checks In the first sections, i highlighted that one of the challenges of orchestration is to be able to detect crashes and the malfunctioning components in order to restart them or take other action to limit the failure. Kubernetes allows to define custom health checks probes to understand if a service is working properly or if some problem is present.

The liveness probe is a configuration used to allow K8s to determine if the pod is healthy and running. The control plane checks if the applications running inside the pod containers are in some error state, and if so, it kills the container and tries to restart it. The probe can be configured to send requests (GET, POST, etc.) to a specific endpoint of the application running inside the container. If the application does not respond to the probe, the container is killed and restarted. Additional configuration parameters include the initial delay before the probe starts and the period between probes to handle custom behaviour.

The readiness probe is a feature of Kubernetes used to determine if the pod is ready to serve traffic. Sometimes, the application can be low to start because of initial setup, waiting for the loading of some data, or establishing a connection to a database.

This probe can be configured as the liveness probe, but instead of restarting the container, it tells the service that the pod is not ready to serve traffic. The service will not send traffic to the pod until the readiness probe returns a success status.

If applications takes a long time to start, the startup probe can be used to check if the application is ready to serve traffic. The startup probe just delays both the liveness and readiness probes for a specific amount of time.

In the practical implementation, i have defined a liveness probe on the users microservice that makes a get request to the “/users-ms/healthcheck” endpoint every 5 seconds. An initial delay of the probe with a value 5 seconds is also configured to leave time to the microservice to connect to the database when it is launched. Otherwise, wile the microservice is doing internal connections and configurations, it may not repsonde to the healthckec and K8s will mark the pod as in an error state.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009