Orchestration of Microservices Using Conductor

Rakshata Karlingannavar[1], Dr.
Nagaraj Bhat[2]

Electronics and Communication
Dept.[1,2]R.V College of
Engineering , Bangalore, India

rakshatak.ec17@rvce.edu.in[1], nbhat437@gmail.com[2]

Abstract: *The Microservices architectural design is widely used today which helps one to build an application as a set of services which can be developed and deployed independently. Each service is independent and gives a set of functions or features that can be individually serviced. In spite of the fact that microservices design has been advanced as the fix just for all cutting-edge application development ailments and is viewed as the replacement for API first application advancement, its execution needs undeniably more idea and practicality. In order for these independent services to work together towards a common goal we need something that will stitch them together because they cannot work in complete isolation and need to share data and interact with one another. There are two ways to do this - microservice choreography and microservice orchestration. This paper tries to explain the difference between choreography and orchestration of microservices, and why the latter is better. We will then discuss about orchestration of microservices using an open sourced microservices orchestrator - Conductor.*

Keywords: Orchestration, choreography, tasks, workflow, microservices.

1. INTRODUCTION

Today, microservice architecture has emerged as one of the best and easiest way to build and manage any application. This style of architecture is a collection of independent services which offer the advantage of being loosely coupled and independently deployed [8], [9]. Hence, they can be developed by a smaller set of people which leads to better organization in the team and the responsibilities and can be separated by specific tasks. Since the services are independently deployed, the required services can be scaled independent of the application [7]. Microservices additionally offer improved error separation whereby on account of an error in one service the entire application does not really quit working. After fixing the error only that particular error needs to be redeployed instead of the whole application. Microservices also offer the flexibility to choose the technology stack which best suits the individual services as opposed to choosing a single technology stack for the entire application [5].

Considering the example of a simple purchase transaction, it requires many services like the payment service, inventory service, a service to manage the shipping process and a service to manage the delivery services. All these services have to run in a specific predefined order. If any service fails, then the tasks executed before that should all be rolled back, just as in the case of a database transaction. All this appears to the end used as a single process, but internally it requires dozens of services to communicate with each other and exchange data. There are two solutions to get the services to work together and to manage the issues in data management in distributed microservices, they are Orchestration and Choreography. Orchestration addresses a solitary concentrated executable business measure that arranges the association among various services. The orchestrator is liable for conjuring and joining the services [4]. There is a controller services which calls the services to be executed, analyses the results and decides if the next services can be called or a rollback has to be performed. The relationship between all the participating services are defined via a single endpoint (i.e., the controller service). Orchestration is a centralized approach [2], [3]. It offers a very tight control of each step in the whole process.

On the other hand, choreography utilizes a decentralized approach. Choreography does not require any central controller process [11]. Choreography differs from orchestration in a major way by the asynchronous nature of choreography. It basically reduces the dependency between the services and allows them to function independently. Since the functionality is independent of any orchestrator, the execution of tasks is faster in choreography. Any request can go from one service to another and back and forth several times and it becomes difficult to track it without an orchestrator to control the end to end transaction. But in case of orchestration you can just ask the orchestrator for the status of any task. Choreography also makes debugging and testing a very tedious procedure. But, in orchestration due to the monitoring by the orchestrator, it is very easy to identity exactly from where any error is coming [6], [12].

2. SURVEY

2.1 Implementation using Conductor

For this use case, the example of a cab rental service is considered, and the working of Conductor will be studied. Conductor is a microservices orchestration engine that is developed and open-sourced by Netflix.

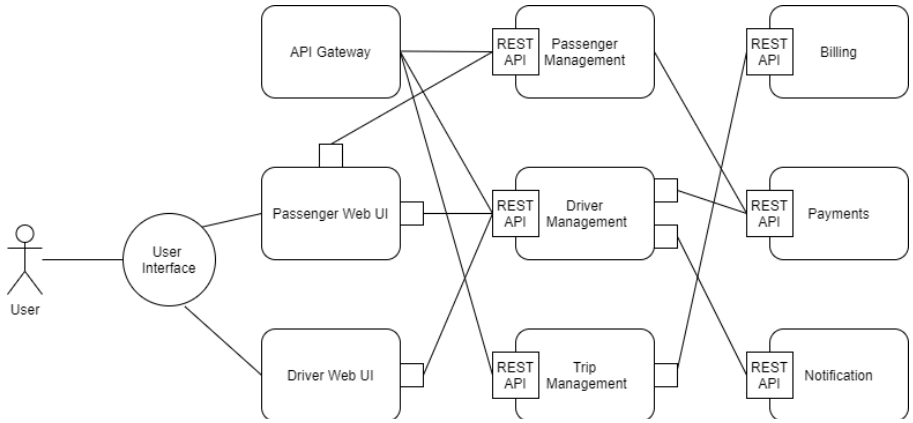

Fig. 1. Uber's higher level microservices architecture

Considering Uber's higher level microservices architecture as shown in Figure 1. We have considered few of the services such as passenger management, driver management, trip management, billing, payments and notifications. These microservices will have a number of tasks that have to be called in a particular order to ensure the smooth management of the application. This is the work of the orchestrator to ensure that the microservices are called in the right order. So, getting into the orchestration part, Conductor has something called as tasks and workflows. Each microservice can be a task and one or more tasks can be added to a workflow which will run those tasks in the given order. Tasks have to be defined before adding them to a workflow. So, the first job is to define the tasks. For this we need to create a JSON body with the parameters that are defined by Conductor. Each task has to be given a name, the description which is optional, the retry count which defines the number of times that will be attempted if the task is marked as failure, the retry logic which defines the mechanisms to schedule the retry, the retry delay seconds defines the time interval between two retries, the timeout policy, the timeout seconds defines the time after which the task is marked a timed out, if it is not completed after transitioning to in progress state, the poll timeout seconds defines the time after which the task is marked as timed out if it is not polled by a worker and the owner email ID. These are the mandatory fields in order to define any task. There are also some other optional parameters that can be used according to the use case.

So once the tasks are created a workflow needs to be created, to which we can add more tasks. A workflow can be created in a similar way that a task was created, that is, by posting a JSON body. The fields to define a workflow include the name of the workflow, the description which is optional, an array of the tasks that should be added to this workflow, the schema version which refers to the conductor schema version which must be 2 currently as 1 is

discontinued, timeout seconds, timeout policy and the owner email ID which is mandatory unless it is disabled. Once the workflows are created, we need to create workers. A worker is a simple code which runs when it gets called by any task. Once the worker is ready, Conductor needs to know where to find that worker. For this a listener is required. A main listener is required in order to listen when any task needs to be executed. The listener will know when there is a task to be executed on the worker. A task listener can be implemented either as a simple main class which runs continuously in the background or it could be a complex Java Spring Boot application [13] running within a Docker container. The latter is used for this example. So, conductor uses a polling model, the workers will poll for the tasks in the workflows which will then execute those tasks that they have polled for. Once a task is executed, it is removed from the queue and the workers can poll for the remaining tasks in the queue.

So, considering the above example, a number of microservices have been defined as mentioned above for this use case. Each microservice will have many tasks inside it. So, in the passenger management microservice tasks like tracking the location of the passenger, estimating the cost of the ride from the starting location to the destination, finding nearby drivers, booking the ride, etc can be present. In the driver management microservice tasks to notify the driver when a nearby customer is trying to book a ride, to accept or cancel a ride request, to get the route to the passenger's location, to get the route from the passenger's location to the destination after verifying the OTP, etc can be added. Under the billing and payments microservice tasks like estimating the final price of the ride, connecting to the bank servers, to make the payment, etc can be added. In the notifications microservice tasks to send different types of notifications for e.g. when the driver has arrived at the location, to send the OTP to the passenger, to give inform about any offer codes, etc can be added.

Task - worker implementation: The tasks communicate via the API layer. These tasks are implemented by the workers. Workers achieve the communication between tasks by either implementing a REST endpoint or by implementing a polling loop that periodically checks for pending tasks in the queues. The polling model permits us to deal with the backpressure on the workers. It also gives auto-scalability depending on the queue length when the situation allows. Conductor gives APIs to monitor the workload size on each worker that can be utilized to auto-scale the worker instances. The workers are planned to be idempotent stateless functions.

2.2 Orchestration VS Choreography

Why orchestration is better than choreography?Orchestration basically controls all the microservices inthe architecture actively, it is similar to a conductor whodirects the musicians in an orchestra. Each musician inan orchestra may be an expert in playing an instrumentbut they wait for the conductor to give the command. Inorchestration there is a single central service that controlsall the communication between the microservices and gives directions to each microservice to perform an intendedfunction. So, in case any error pops up, the orchestrator orthe central task can be asked as to from where exactly thatparticular error is being thrown. While orchestration can be compared to a symphony, choreography can be said tobe similar to a dance group. In a dance group each dancerknows what has to be done, which step has to be performed,each dancer is able and is required to do the right steps atthe right time. In choreography all the microservices needto exchange messages among themselves when somethinghappens. For this an event broker will be required. If anymicroservice sends any message, it doesn't bother about whathappens next or wait for the response. Everything after thathappens in an asynchronous way. Every microservice onlyobserves its environment. If any other service subscribesto this channel of messages will know what to do next. Sobasically, choreography is an event driven model and allthe microservices publish some event when some businessrelevant task takes place within them. This process proceedstill the last service which does not publish any more events,there by marking the end of transaction. It can be visualized as shown in Figure 2.

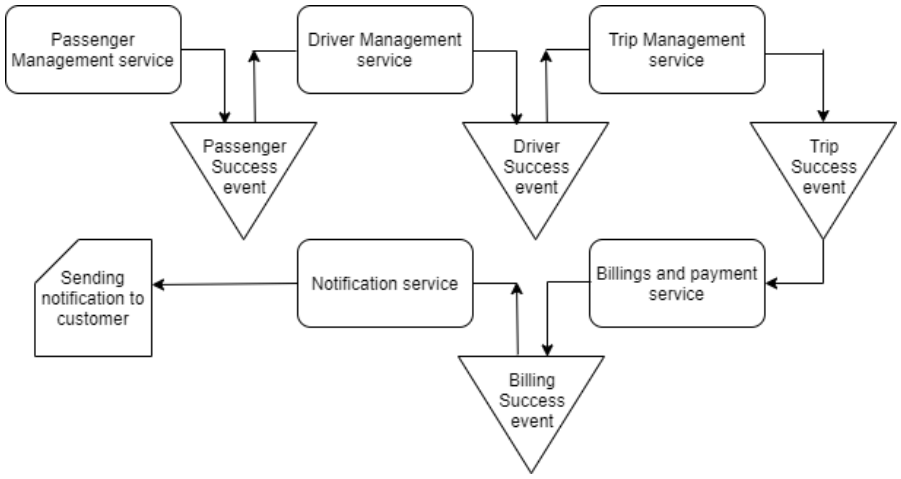

Fig. 2. Event Choreography Flow

Considering the cab rental example, in orchestration a central orchestrator will be present which will handle all the tasks and workflows. So, the orchestrator will take care of the whole flow say from finding a ride to making the payment. Suppose in the payment method if something goes wrong, the orchestrator will know the flow completely and can revert back. But in the case of choreography, if the payment method fails, the payment microservice only has to trigger an event to roll back the changes. This needs to trigger many other events to cause the roll back in each service. This might lead to an error as the payment service might not have all the information to do so. And if the roll back didn't take place in any microservice due to an error it will lead to database inconsistency. Whereas in orchestration the orchestrator will have all the information and will know exactly how to roll back. Some other issues with choreography are that there is a tight coupling and assumptions around input or output which make it very hard to adapt to the changing requirements, the process flows are embedded inside the code of numerous applications and also there is no practically real way to answer how much part of any process is completed. When there are smaller number of microservices choreography is significantly faster than orchestration. But as the number of events go on increasing it become very difficult to manage all the microservices individually in choreography. But orchestration can handle multiple events very smoothly without much confusion as all the events are orchestrated at a central location. The Orchestration flow can be visualized as in Figure 3.

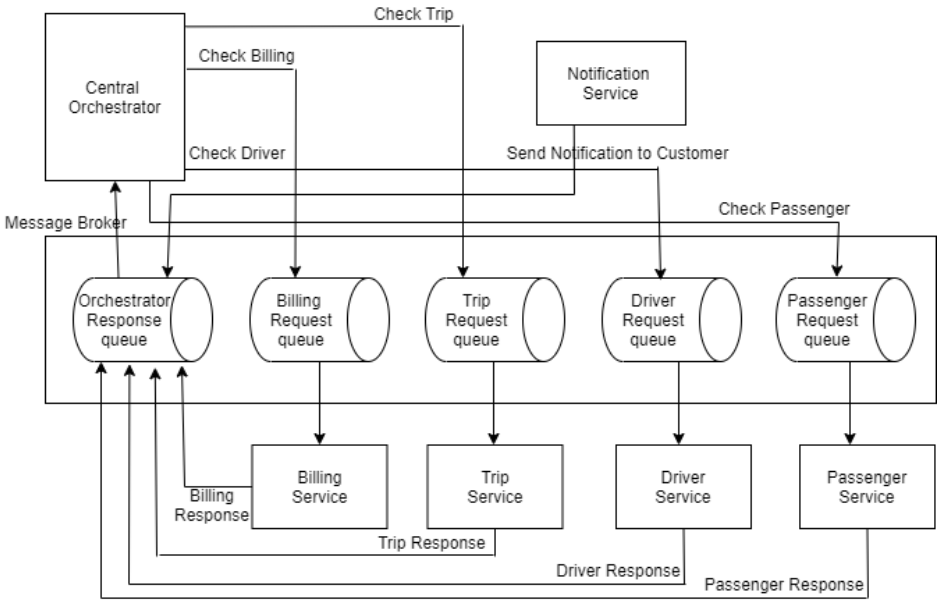

Fig. 3. Orchestration Flow

3. RESULTS

From the above experiment it was found that Conductoras an orchestrator is a very good solution to management ofmicroservices because of the various characteristics of thisengine which make it very easy to use. These include thecapacity tomake complex workflows in a simple manner.The tasks are executed by the microservices. All that has tobe done is to writeworkers to poll on those tasks. We cangive the task and workflow blueprints in a JSON DSL, whichis very easy for anybody to understand who are startingfrom scratch. The engine allows easy traceability of any taskexecution and also provides visibility to track the execution.It also provides the ability to pause, stop or resume tasks atany point during the execution. It also has the ability to scalemillions of workflows. Conductor also provides pluggablestorage and API layers. Hence it gives the users the flexibilityto choose any different queues or storage engines dependingon the use case provided the interface is implemented.

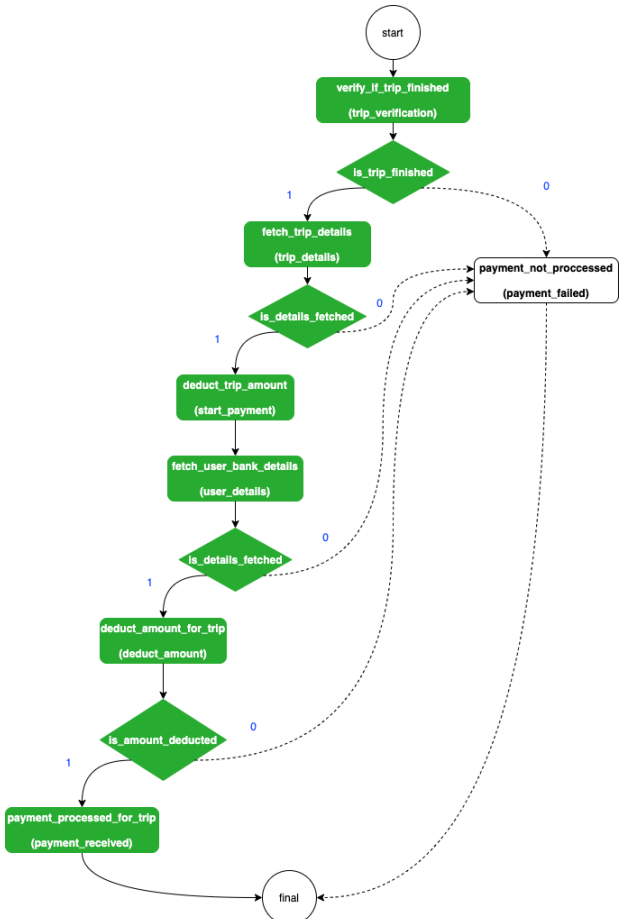

Fig. 4. Billing Workflow in Conductor

Figure 4 shows how the billing workflow looks aftercompletion in the Conductor UI. In Conductor we can createdifferent types of tasks, here Simple and Decision tasks areused to create this workflow. The decision task works similarto the case switch statement in any programming language,so it can have multiple outcomes. In this example, we haveused only two cases, either true or false. In this workflowtasks such as verifying if the trip is completed, fetching thetrip details, fetching user bank details, deducting amount from the user's bank account are used. In this case, it can be seen that all the tasks executed successfully hence the failure task wasn't executed and the payment process went throughsuccessfully.

4. CONCLUSION

In this paper, the microservices architecture was discussedin brief and its advantages were discussed which madeapplication development very easy. If we have too manymicroservices a mechanism to manage those microserviceswill be required. Hence microservices choreography andorchestration come into picture. We then saw the majordifferences between choreography and orchestration and howthe latter performed better

when the number of services goeson increasing. It was found that choreography is much fasterthan orchestration, but event choreography is very hard to codeand manage when there are a lot of events being triggeredfrom every microservice. It is also evident that handlingmultiple events without a central orchestrator becomes verydifficult as one team working on one service may not beaware about the other events being triggered. Moving further,the working of conductor and its terminologies was studied.A simple cab rental example was studied and executed inConductor in order to understand the working better.

5. FUTURE SCOPE

Looking from the Conductor point of view some of thedevelopments which might result in a more user friendlyexperience while using Conductor would be to add the featureto create and manage the workflows using JSON DSL, i.e.if the workflow functionality can be mentioned using JSONDSL, because currently workers and task listeners have to bewritten separately which is a little tedious. Another additionwould be to log the execution data of each task which willhelp in the troubleshooting any errors easily. Another usefuladdition will be to add support for the AWS Lambda functionas tasks to support serverless simple tasks.

REFERENCES

10.1 Journal Articles

[1] D. Luong, H. Thieu, A. Outtagarts and B. Mongazon-Cazavet, “Telecom microservices orchestration,” 2017 IEEE Conference on Network Softwarization (NetSoft), 2017, pp. 1-2, doi: 10.1109/NETSOFT. 2017.8004255.

[2] J. Rufino, M. Alam, J. Ferreira, A. Rehman and K. F. Tsang, “Orchestration of containerized microservices for IIoT using Docker,” 2017 IEEE International Conference on Industrial Technology (ICIT), 2017, pp. 1532-1536, doi: 10.1109/ICIT.2017.7915594.

[3] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah and Y. Chen, “Orchestration of Microservices for IoT Using Docker and Edge Computing,” in IEEE Communications Magazine, vol. 56, no. 9, pp. 118-123, Sept. 2018, doi: 10.1109/MCOM.2018.1701233.

[4] Guerrero, C., Lera, I. Juiz, C. “Resource optimization of container orchestration: a case study in multi-cloud microservices based applications,” J Supercomput 74, 2956–2983 (2018). https://doi.org/10.1007/s11227-018-2345-2

[5] A. Sill, “The Design and Architecture of Microservices,” in IEEE Cloud Computing, vol. 3, no. 5, pp. 76-80, Sept.-Oct. 2016, doi: 10.1109/MCC.2016.111..

[6] Chaitanya K. Rudrabhatla, “Comparison of event choreography and orchestration techniques in microservice architecture,” (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 9, No. 8, 2018

[7] A. Balalaie, A. Heydarnoori and P. Jamshidi, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture,” in IEEE Software, vol. 33, no. 3, pp. 42-52, May-June 2016, doi: 10.1109/MS.2016.64.

[8] N. Alshuqayran, N. Ali and R. Evans, “A Systematic Mapping Study in Microservice Architecture,” 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), 2016, pp. 44-51, doi: 10.1109/SOCA.2016.15.

[9] T. Salah, M. Jamal Zemerly, Chan YeobYeun, M. Al-Qutayri and Y. Al- Hammadi, “The evolution of distributed systems towards microservices architecture,” 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST), 2016, pp. 318-325, doi: 10.1109/ICITST.2016.7856721.

[10] D. Guo, W. Wang, G. Zeng and Z. Wei, “Microservices Architecture Based Cloudware Deployment Platform for Service Computing,” 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE), 2016, pp. 358-363, doi: 10.1109/SOSE.2016.22.

[11] F. Dai, Q. Mo, Z. Qiang, B. Huang, W. Kou and H. Yang, “A Choreography Analysis Approach for Microservice Composition in Cyber- Physical-Social Systems,” in IEEE Access, vol. 8, pp. 53215-53222, 2020, doi: 10.1109/ACCESS.2020.2980891.

[12] Neha Singhal, Usha Sakthivel, Pethuru Raj, “Selection mechanism of micro-services orchestration vs. choreography,” International Journal of Web Semantic Technology (IJWesT) Vol.10, No.1, January 2019.

[13] Prasad Reddy K.S., “Introduction to Spring Boot,” 2017 In: Beginning Spring Boot 2. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-2931-6 1.