An Evaluation of Saga Pattern Implementation Technologies

Karolin Dürr, Robin Lichtenthäler, and Guido Wirtz

Distributed Systems Group, University of Bamberg

Abstract. The Saga pattern is frequently mentioned in the literature to structure communication workflows in Microservices Architectures. To ease the implementation of the Saga pattern frameworks and tools have emerged. By implementing an exemplary use case, we qualitatively evaluate two of such technological solutions in this paper according to criteria relevant for Microservices Architectures. This evaluation can be considered when deciding on which technology to use for implementing the Saga pattern, or also as a more general insight into what should be kept in mind when implementing the Saga pattern in Microservices Architectures.

Keywords: Microservices, Saga pattern, Workflow

1 Introduction

The *Microservices Architecture* is a pattern that emerged from real-world usage and constitutes a fast-moving topic [5, 9, 16]. A microservice is a small and autonomous service modeled around a business domain. A distributed system that consists of numerous microservices represents the Microservices Architecture [5] where data storage is ideally not shared, but owned exclusively by each microservice. Communication between microservices happens via messages over the network [9]. The main advantage is service independence enabling independent deployability, maintainability and evolvability of services [11]. A main challenge, however, is inter-service communication, because communication over the network is comparatively slow and unreliable [5]. The question arises how this communication can be structured and managed, especially for complex business scenarios with multiple services which even require certain transactional guarantees [16].

Such a complex business scenario would be booking a trip where the booking includes several steps such as booking a flight, a hotel, and a rental car. Applied to a Microservices Architecture where different microservices are responsible for the different steps, this scenario has also been used by Catie McCaffrey in a conference talk[1] to motivate the usage of the Saga pattern.

Because all steps are required to book a trip as a whole, a classical approach would be to use a distributed transaction for example with the *2-Phase Commit*

[1] https://www.youtube.com/watch?v=xDuwrtwYHu8, last accessed: 2021-02-17

J. Manner, S. Haarmann, S. Kolb, N. Herzberg, O. Kopp (Eds.): 13th ZEUS Workshop, ZEUS 2021, Bamberg, held virtually due to Covid-19 pandemic, Germany, 25-26 February 2021, published at http://ceur-ws.org
Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2839/paper12.pdf

(2PC) protocol [1, 9]. However, classical distributed transactions contradict the service independence characteristic of a Microservices Architecture. First, the 2PC protocol depends on the availability of all participants [7, 11]. If one participant fails, the system as a whole becomes unavailable. Second, the scalability is affected, because 2PC participants need to lock resources which can affect the overall transactional throughput and lead to competitive situations [9, 12]. And third, distributed transactions are missing support from modern technologies, like NoSQL databases or message brokers [11].

Therefore, other approaches have been discussed [3, 7, 9] with the Saga pattern being mentioned frequently [6, 10, 11, 15]. The Saga pattern divides a transaction that might take a long time into multiple local ones. Thereby, it reduces the dependence on the availability of all participants at the same time and prevents the need to lock all included resources until full completion. Although it can therefore not provide the same transactional guarantees as, for example, the 2PC protocol, it aligns better with the characteristics of Microservices Architectures. Using the Saga pattern for the trip booking scenario means that still the whole trip booking needs to be supervised by one service. However, the included steps, such as booking a hotel or booking a flight, are done more independently in local transactions by the different services involved.

Because this separation into multiple more independent transactions leads to additional challenges, implementing the Saga pattern can get complex. Therefore framework support is desirable and some technological solutions have emerged. The goal of this paper is to investigate the capabilities offered by existing frameworks with a focus on orchestrated Sagas and the context of characteristics and challenges of Microservices Architectures. This is summarized in the following research question:

> **RQ:** How well do recent technological solutions support implementing the Saga pattern concerning the design, the execution and the visualization of communication between microservices?

In Sect. 2, our approach to answer the research question is described. In Sect. 3, the details of the Saga pattern are depicted based on literature and the already introduced example scenario. This is used as a foundation for the following evaluation in Sect. 4, our main contribution. Finally, we draw a conclusion in Sect. 5.

2 Methodology

First, we carried out a literature review to understand the Saga pattern itself and its applicability to the Microservices Architecture. As sources, we considered the original paper for the Saga pattern [6], as well as more recent books [3, 10, 11] and papers [8, 15] which add the context of microservices. The result is the description of the Saga pattern in Sect. 3 based on the trip booking example.

We then used this example to implement the Saga pattern with available technological solutions. Solutions that have emerged so far are Axon[2], Eventuate Tram[3], Netflix Conductor[4], and more recently Long Running Actions for MicroProfile[5]. However, because this work has been done as a part of the bachelor thesis of the first author, we had to limit the scope and therefore only selected two solutions. The first solution we selected for our evaluation is Eventuate Tram, because it is specifically designed for the Saga pattern and described in detail in [11]. And the second solution is Netflix Conductor, because Netflix as a company is well-known for its successful microservices approach [2]. Furthermore, Netflix Conductor was not considered in another similar study [15] which compared technological solutions for the Saga pattern. The study by Štefanko et al. [15] includes a small set of criteria for comparing the different solutions which are not explained in detail in the paper and additionally discusses problems of the solutions in a qualitative way. Furthermore, Štefanko et al. [15] conducted a performance test to measure processing times and throughput. In contrast, we derived a more comprehensive criteria catalog from general Saga execution characteristics as well as from considering Microservices Architecture characteristics and challenges to evaluate the solutions. Our evaluation is qualitative, because we assess the solutions according to the criteria catalog based on our implementations. We have not performed quantitative evaluations, like performance benchmarks to assess scalability and throughput or user experiments to assess the ease of use. With respect to the work of Štefanko et al. [15], our work extends it by evaluating additional criteria and considering an additional solution. The resulting evaluation of Eventuate Tram and Netflix Conductor based on these criteria is presented in Sect. 4.

3 The Saga Pattern

The Saga pattern was introduced by Garcia-Molina and Salem [6] for long lived transactions by designing them as a sequence of local transactions. Although they focused on a centralized system, they also mentioned the possibility of a distributed implementation [6]. Therefore, Sagas have been proposed for updating

[2] `https://docs.axoniq.io/reference-guide/axon-framework/sagas`, last accessed 2021-02-17
[3] `https://eventuate.io/`, last accessed 2021-02-17
[4] `https://netflix.github.io/conductor/`, last accessed 2021-02-17
[5] `https://microprofile.io/project/eclipse/microprofile-lra`, last accessed 2021-02-17

data in multiple services in a Microservices Architecture without using distributed transactions [11].

To clarify this, Fig. 1 shows an exemplary execution of the trip booking example. The system offers the possibility to book a trip which includes booking a hotel and a flight. This can be considered as a long lived transaction with three microservices involved: a *Travel Service* which accepts requests for booking a trip and initiates the execution, a *Hotel Service* which manages hotel bookings, and a *Flight Service* which manages flight bookings. Using the 2PC protocol would mean that booking the hotel and the flight would be done within one ACID [14] transaction coordinated by the Travel Service where the whole trip is either booked or rejected. During the transaction execution, all services must lock resources impacting throughput [9, 12]. If one service is temporarily unavailable, the transaction may fail reducing the availability of the system as a whole [7, 11].

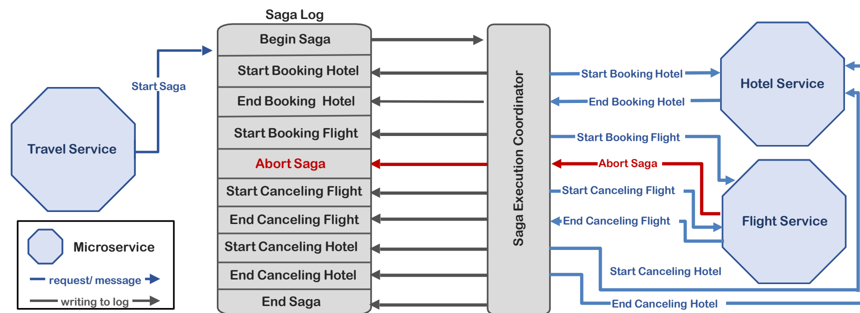

Fig. 1. Execution of a Saga's failure scenario based on [6].

Implementing the example as a Saga means that the long lived transaction is split into three local transactions: Save Trip Information, Booking Hotel, and Booking Flight. The Save Trip Information transaction is executed upon a trip booking request locally by the Travel Service to initiate the Saga and ensure Durability. It is part of the Begin Saga step and does not require communication with another service which is why it is not explicitly shown in Fig. 1. The Booking Hotel transaction and the Booking Flight transaction are executed locally in the Hotel Service and Flight Service, respectively. Each local transaction updates only the data within one service and then triggers the next one [8, 11] until all transactions are completed, and hence the Saga itself completes. If one transaction fails, the Saga aborts and all previously completed transactions have to be compensated. This case is shown in Fig. 1, where after the booking of a flight fails, the previously done hotel booking has to be canceled. Consequently,

[6] `https://speakerdeck.com/caitiem20/applying-the-saga-pattern?slide=70`, slide 70, last accessed: 2021-02-17

for each local transaction, a compensating transaction also needs to be provided, which can compensate the transaction completely or at least semantically [6, 10].

In contrast to the 2PC protocol, a service therefore only holds locks for local transactions and not for the whole Saga execution enabling it to effectively serve more requests. One consequence is that the Isolation property is not satisfied because intermediate results are visible to other Sagas before the executing one is fully committed [15]. Therefore, countermeasures need to be taken to prevent anomalies resulting from the lack of Isolation [11]. Also, Atomicity is not given for the Saga as a whole, solely for each local transaction [6, 10]. Instead of strict Consistency, only eventual consistency [13] is provided [10, 15]. During execution, a trip with a hotel, but no flight would be inconsistent, but after completion consistency is again achieved, when necessary through compensations. Durability is fully guaranteed through the durability of local transactions and the Saga log, which is a distributed log to persist every executed transaction. The Saga log is managed by a component called the Saga Execution Coordinator which is itself stateless and uses the log to trigger transactions and thereby proceed Saga executions [6]. Having a Saga Execution Coordinator either as a separate service or within a service exemplifies the orchestrated Saga approach with the Saga Execution Coordinator being called the orchestrator [3, 10]. Although out of the scope of this work, also a choreographed approach would be possible where the coordination is distributed [10].

4 Technological Evaluation

Before we discuss the evaluation based on a set of criteria, some fundamental differences need to be mentioned, because they also affect our evaluation results. Eventuate Tram specifically focuses on Sagas by offering a Java-based Domain Specific Language (DSL) for specifying a sequence of transactions and corresponding compensating transactions inside the service acting as the Saga orchestrator. It is then executed together with the so-called CDC service and infrastructure components such as a database for persisting the Saga log and a message broker for communication. The DSL can also be used for the participants, if implemented with Java. For other languages, participants have to be integrated based on the used communication mechanisms. We implemented all services as Spring[7] services with the DSL included. In contrast, Conductor is not designed explicitly for Sagas, but distributed workflows in general. The central component is the Conductor server which accepts workflows in the form of a JSON-based DSL. A Saga is registered as a workflow, with tasks representing transactions for which different types are offered. We used so-called worker tasks which are more customizable than others. They need to be registered, and the services, again implemented in Java, can then poll and update these tasks to proceed with the workflow. All implementations with examples and detailed information on execution can be found online[8].

[7] `https://spring.io/`, last accessed 2021-02-17

[8] `https://github.com/KarolinDuerr/BA-SagaPattern`

Table 1. Evaluation overview

Criterion	**Eventuate Tram**	**Netflix Conductor**
General Saga Characteristics		
Specifying compensating transactions (CT)	✓	✓
Automated execution of CTs	✓	✓
Compensation only where needed	✓	not directly supported
Parallel execution of transactions	✗	✓
Choreographed Sagas	✓	✗
Monitoring		
Runtime state of Sagas	via database	UI visualization
Orchestrator metrics	from CDC service	from Conductor server
Tracing	Zipkin integration	not directly supported
Logging	microservices logs	Conductor server logs
Expandability		
Relatively simple integration	✓	✓
Terminating or pausing running Sagas	not directly	via UI
Versioning Sagas	✗	✓
Built-in language support	Java	Java, Python
Any language for orchestrator	✗	✓
Any language for participant	✓	✓
Failure performance		
Enforced execution timeouts	✗	✓
Retry of failing participant without restart	✗	✓
Independent compensating transactions	✓	✗
Auto-continuation after orchestrator crash	✗	✓
No. of services for orchestration	2	1
New Sagas while orchestrator unavailable	✓	only with buffering
High availability	through replication	through replication

Our first set of criteria (see Table 1 for an overview of all results) covers *general characteristics*. Both technologies allow for specifying compensating transactions which are also automatically triggered. However, only Eventuate allows for mapping compensating transactions to transactions so that only needed compensating transactions are executed while Conductor allows for one failure workflow per workflow. That means the failure workflow must contain all compensating transactions and even compensating transactions which would not have been necessary are executed in case of a Saga abort. This is because Conductor is not specifically focused on Sagas. With Conductor, the central component is the Conductor server which orchestrates the Saga execution, and participants are connected to the Conductor server, which is why it does not support a choreographed approach to Sagas. With Eventuate as a framework however, the participants could also be connected directly with each other, enabling also a choreographed approach. In contrast, transaction execution in Eventuate is strictly sequentially, while Conductor also allows for parallel execution of transactions.

The second set of criteria considers *monitoring*, a challenge in Microservices Architectures [2, 4]. To get insights at runtime, Eventuate offers no pre-built tool, but the database tracking all transactions and messages could be used as a source for building a custom monitoring solution. Instead, Conductor offers a UI which visualizes current workflows and provides useful functionalities for runtime insights. A metrics endpoint exists for both technologies, which can be used to collect metrics like the number of sent messages, average execution times, or the number of failed Saga workflows. A possibility to use distributed tracing is only given by Eventuate which offers a pre-built Zipkin[9] integration. Additionally, logs are written by both technologies which could help with troubleshooting.

Because Microservices Architecture-based systems change and evolve, the third set of criteria covers *expandability*. We extended the example with an additional service, which can also be found in the repository. For both technologies, the integration was possible without significant problems. Nevertheless, Conductor is suited better for updating or extending a running system because currently executing Sagas can be managed via the UI and workflows can be versioned. This means that Sagas of a new version can be started at the same time as there are still Sagas of an old version executing. With Eventuate, handling updates at runtime requires more effort. Regarding polyglot programming as a characteristic of Microservices Architectures [9, 16], Eventuate is a bit more restricted because the DSL is based on Java which means that the orchestrator also needs to be written in Java. With Conductor, the Conductor server is mainly responsible for the orchestration which means that the service starting a Saga can be written in any language. In addition to a pre-built Java client for writing participants, Conductor also offers a Python client.

As a final set of criteria, we consider the technologies' *handling of failures* which have to be expected in a distributed system. Both technologies tolerate possible crashes of Saga participants by retrying communications. However, only

[9] `https://zipkin.io/`, last accessed 2021-02-17

Conductor enforces an execution timeout to be set while Eventuate might, per default, wait indefinitely for a service to restart. Depending on the use case and volume of requests, this can become an issue. If there is no execution timeout for Sagas, a service being unavailable for an extended period together with a high volume of requests might lead to an overloaded system as a whole, because Saga executions pile up and cannot make progress. An execution timeout can then protect the system from consequential failures. Then again, there might be use cases where Sagas should not be stopped at all because of a timeout. In such a case, the enforced execution timeout of Conductor might be problematic. A participant responding with a failure is unsubscribed from the message broker with Eventuate, requiring a full restart of the participant so that it can re-register. In contrast, Conductor retries even if a participant responded with a failure that might only be temporary. Because compensating transactions are executed only where needed with Eventuate, they can be executed independently from participants where no compensation is necessary. Thus, also crashes of such participants can be tolerated. Crashes of the Saga coordinator are tolerated by both technologies and execution can continue afterwards because all necessary information is persistently logged. However, merely Conductor automatically continues while Eventuate needs a trigger after restart, such as a new Saga start. With Eventuate, two services are required for orchestration: The CDC service as orchestrator and a service in control of the Saga. Therefore, new Sagas can still be started if only the orchestrator is unavailable. In case of Conductor, the Conductor server is the exclusive orchestrator and additional logic would be needed to buffer new requests in another service. Finally, both can be set up as a highly available system by replicating the CDC service or the Conductor server, respectively.

To summarize, both technologies enable robust Saga implementations. The characteristics of the Saga pattern are represented more clearly with Eventuate than with Conductor. However, Eventuate comes with limitations regarding the flexibility in operation which is in turn better supported by Conductor.

5 Conclusion and Outlook

Given the Microservices Architecture as a popular software architecture approach, patterns and technologies are needed to efficiently implement these systems and tackle their accompanying challenges. Our evaluation of Saga pattern implementation technologies covers one of the aspects software engineers should consider to make an informed decision on which technologies to use based on their specific needs. As future work, we want to include additional technologies into our evaluation, such as Axon and Long Running Actions for MicroProfile, but also the possible usage of BPMN workflow engines as proposed in a recent talk by Bernd Rücker[10] and also by Niall Deehan at the ZEUS 2020 workshop. Furthermore, extending the evaluation with quantitative methods is imaginable, for example by doing a performance benchmark.

[10] `https://www.youtube.com/watch?v=7uvK4WInq6k`, last accessed: 2021-02-17

References

1. Al-Houmailya, Y.J., Samaras, G.: Two-Phase Commit. In: Encyclopedia of Database Systems, pp. 3204–3209. Springer US (2009), https://dx.doi.org/10.1007/978-0-387-39940-9_713
2. Alshuqayran, N., Ali, N., Evans, R.: A Systematic Mapping Study in Microservice Architecture. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). pp. 44–51. IEEE Computer Society (2016), https://dx.doi.org/10.1109/SOCA.2016.15
3. Bruce, M., Pereira, P.A.: Microservices in Action. Manning Publications, 1st edn. (2018), ISBN: 9781617294457
4. Cerny, T., Donahoo, M.J., Trnka, M.: Contextual Understanding of Microservice Architecture: Current and Future Directions. ACM SIGAPP Applied Computing Review 17(4), 29–45 (2018), https://dx.doi.org/10.1145/3183628.3183631
5. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, Today, and Tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer International Publishing (2017), https://dx.doi.org/10.1007/978-3-319-67425-4_12
6. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the 1987 Association for Computing Machinery Special Interest Group on Management of Data (ACM SIGMOD) International Conference on Management of Data. vol. 16, pp. 249–259. ACM Press (1987), https://dx.doi.org/10.1145/38714.38742
7. Helland, P.: Life Beyond Distributed Transactions: An Apostate's Opinion. ACM Queue 14(5), 69–98 (2016), https://dx.doi.org/10.1145/3012426.3025012
8. Limón, X., Guerra-Hernández, A., Sánchez-García, A.J., Arriaga, J.C.P.: SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture. In: 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT). pp. 50–58. IEEE Computer Society (2018), https://dx.doi.org/10.1109/CONISOFT.2018.8645853
9. Newman, S.: Building Microservices - Designing Fine-Grained Systems. O'Reilly Media, Inc., 1st edn. (2015), ISBN: 9781491950357
10. Newman, S.: Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, Inc., 1st edn. (2019), ISBN: 9781492047841
11. Richardson, C.: Microservices Patterns. Manning Publications, 1 edn. (2019), ISBN: 9781617294549
12. Thomson, A., Diamond, T., Weng, S.C., Ren, K., Shao, P., Abadi, D.J.: Calvin: Fast Distributed Transactions for Partitioned Database Systems. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. pp. 1–12. Association for Computing Machinery (2012), https://dx.doi.org/10.1145/2213836.2213838
13. Vogels, W.: Eventually Consistent. Communications of the ACM 52(1), 40–44 (2009), https://dx.doi.org/10.1145/1435417.1435432
14. Vossen, G.: ACID Properties. In: Encyclopedia of Database Systems, pp. 19–21. Springer US (2009), https://dx.doi.org/10.1007/978-0-387-39940-9_831
15. Štefanko, M., Chaloupka, O., Rossi, B.: The Saga Pattern in a Reactive Microservices Environment. In: Proceedings of the 14th International Conference on Software Technologies (ICSOFT) 2019. pp. 483–490. SciTePress (2019), https://dx.doi.org/10.5220/0007918704830490
16. Zimmermann, O.: Microservices Tenets. Computer Science - Research and Development 32(3-4), 301–310 (2016), https://dx.doi.org/10.1007/s00450-016-0337-0