# Consistency and Autonomy in the Microservice Architecture

Michael Müller
michael.mueller2@haw-hamburg.de
University of Applied Science (HAW) Hamburg
Berliner Tor 7
20095 Hamburg, Germany

31. January 2019

**Abstract**

This paper discusses the balance between autonomy and consistency in a microservice environment. Three consistency solutions will be selected and compared. These three are 'Shared Database', 'SAGA Orchestration' and 'SAGA Choreography'. With these solutions it is also possible to compare Orchestration and Choreography. Each solution has it benefits, problems and can be considered in different situations. Even though the need for strong consistency may be better fulfilled with a monolithic or 'Service-Oriented' architecture.

**Keywords:** Microservice, Architecture, Autonomy, Consistency

## 1 Introduction

The Microservice Architecture is currently widely known and applied in distributed systems. In a microservice architecture each service is independent of other services (autonomy). The important autonomy aspects for this paper are that each service is the owner of its data and can have a different techstack. ([2], [11], [9])

If multiple microservices use the same data, it can get tricky to preserve their independence while keeping the shared copies consistent across all services. This gets even more tricky when the data representations of the same data or the used technology varies throughout the distributed microservice system. The heterogeneous technology stacks may be a big problem as some technologies are not compatible with each other. ([2], [11], [10])

The first question of this paper is, how the autonomy of microservices and the consistency of their data interfere with each other. The second question is, if its better to have a central service that checks and enforces

the global consistency ('Orchestration') or if this is better be done by the services themselves ('Choreography'). Performance, scaling, crash detection and recovery of microservices architectures are not essential parts of this paper.

In the upcoming section Fundamentals will be described. In section 3, different consistency solutions will be introduced and three are selected for comparison in section 4. The last two sections answer the specified questions and give a future outlook.

# 2 Fundamentals

## 2.1 Consistency Models

A system or a state is consistent when all copies of the same data have the same value. There are different **consistency models** which describe how consistent the system actually is. 'Strict Consistency' for example is the strongest consistency model and defines that the system is always consistent. On the other end we have 'Eventual Consistency' which doesn't really give a consistency guarantee at all, the system will become consistent sometime in the future. Between these two extremes are several models, one is 'Causal Consistency' which guarantees that related operations are always in the correct order, but it doesn't ensure any guarantee for non-related operations. [6]

## 2.2 Autonomy

[1] describes autonomy as one aspect of loose coupling. Autonomy means the decomposition of a system into independent modules on a management level. The owner of the module has complete control over the internal aspects. In [3] this is described in other words, that a team has the responsibility and ownership of a microservice and can independently design, implement, test and extend the microservice.

# 3 Solutions

In this section, seven solutions will be described and compared. Based upon the given descriptions and properties in the beginning, three solutions will be picked in the end for further investigations.

## 3.1 Important solution properties

To gain a better overview over the solutions, important properties will be specified. These properties can be split in two different groups. While the

first group describes the algorithm, the second group concentrates on the solutions effect on the architecture (P2).

- P1.1 How does the solution know the location of all replicas of a certain data object?
- P1.2 How does the solution know about changes of these data objects and which process executes this algorithm (Orchestration or Choreography)?
- P1.3 How does the solution apply the change to all replicas and which process executes the update?
- P1.4 Which consistency model can be granted?
- P2.1 Which changes would need to be applied upon an existing system?
- P2.2 How does the solution affect the microservice autonomy?
- P2.3 How does the solution work with heterogeneous technologies?

## 3.2 Solutions

Seven unique solutions will be introduced. Including the variations, this gives a total of eight solutions.

### 3.2.1 SAGA Choreography with Events

SAGA is a pattern introduced by [5] in 1987 and can be described as the distributed and more loosen version of the Two Phase Commit. A change message will be distributed to all applications and if a failure occurs the already done changes will be rolled back. Depending on the implementation the forwarding and controlling of the messages' success is either done by an Orchestrator or by the applications themselves (Choreography).

Events are often used in microservice architectures due to their very good scalability. For consistency, event messages can be used to signal other services that data has changed without even knowing the existence or location of these services. [9]

- P1.1 Every microservice listens to certain event messages distributed by a message broker.
- P1.2 The services will receive a message which is relevant for him and may insert extra event messages again. (Choreography).
- P1.3 Change Request is read by microservice which then will apply the change.
- P1.4 Eventual consistency.
- P2.1 In the new system, all change requests need to send and received from the message broker, only read requests are allowed to be sent directly. This message broker may have to be introduced. In addition, of receiving requests from the exposed interface (e.g. REST), the services now need to listen to event messages too.
- P2.2 The autonomy does not decrease as long as the event messages are treated like an external interface.

P2.3 The technology in use must understand the selected message broker.

### 3.2.2 SAGA Orchestration

Sources: [9], [5], [4]

P1.1 All Services and which data objects they have is known to a central service, here the so called 'SAGA Orchestrator'.
P1.2 When the 'SAGA Orchestrator' receives the change request, it will forward this request to the relevant services. (Orchestration)
P1.3 A Service receives a change request and applies it.
P1.4 Not specified in source.
P2.1 In the new system, all change requests need to be sent to the 'SAGA Orchestrator'.
P2.2 The autonomy decreases since the extra service has extended knowledge about the used data object in the services.
P2.3 The technology in use must be able to talk to the services.

A more general variation of this solution is the 'Aggregator Pattern' [9]. With the 'SAGA Orchestration' being the more sophisticated consistency solution the 'Aggregator Pattern' is not mentioned in more detail.

### 3.2.3 Shared Database Microservice Pattern

Sources: [9] [11]

P1.1 There is a fixed location for each data object, in SQL this would be one table per data object.
P1.2 One data object in a table changes and the owning service recognizes this change. (Choreography)
P1.3 This solution basically just says that every service uses the same database. How consistency is then achieved is not mentioned. This solution could be extended so the service adjust its table according to the recognized changes within other tables.
P1.4 Consistency up to Strict Consistency is possible. This solution is preferably done with SQL databases, as NoSQL databases may weaken the guaranteed consistency model or need more work to achieve strict consistency due to the lack of ACID transactions.
P2.1 In the new system, the services have to frequently check the database if changes have happened. For this, all services need to be able to connect to the database and know exactly where one data object is and how it is stored (in SQL e.g. the schema)
P2.2 The autonomy decreases since the services need to share where and how a data object is stored.
P2.3 The technology in use must understand the shared database

### 3.2.4 Listening to communication - Verification

Sources: [7]

If requests are sent that implicitly reflect the same internal state of (e.g.) sender and receiver, the global state is consistent (e.g. SYN and SYN/ACK in TCP). This solution could be expanded to also enforcing consistency.

- P1.1 Services and their behaviour about data objects ('Contracts') are predefined and hardcoded into an extra service.
- P1.2 When a request is read by the extra service that implicitly reflects an internal change of a service state. (Orchestration for the consistency check)
- P1.3 This is an algorithm to only verify if services are in the same state. This solution could be extended so the extra service will send change requests to the affected microservices.
  So this solution doesn't send changes to the services to enforce consistency.
- P1.4 This verifies strict consistency and what it enforces is up to the implementation.
- P2.1 In the new system, all change requests need to be observable by the extra service, which needs to be implemented too.
- P2.2 The autonomy decreases since the extra service has complete knowledge about the internal behaviour of the services.
- P2.3 The technology in use must send requests that can be understood by the extra service.

### 3.2.5 Listening to communication - Templates

Sources: [8]

Developer must define valid orders of causal operations (templates). If an order of operations matches the beginning of a template, this template is followed.

- P1.1 Services and their behaviour about data objects and requests are predefined and hardcoded in an extra service
- P1.2 When a change request is read by the extra service and the request matches a template. (Orchestration)
- P1.3 While following a template, the extra service keeps the state consistent as it delays requests that are sent too early to prevent the global state to become inconsistent.
- P1.4 Causally Consistency
- P2.1 In the new system, all change requests need to be observable by the extra service, which needs to be implemented too. Also it must be possible for this extra service to delay requests.

P2.2 The autonomy does not decrease as long as the messages are treated like an external interface.
P2.3 The technology in use must send requests that can be understood by the extra service.

#### 3.2.6 Synapse

Sources: [10]

Synapse is a service[1] on which databases can register with the info which data objects they need. Synapse then will publish all changes to the registered databases.

P1.1 Databases subscribe to certain data objects.
P1.2 When a change operations comes, Synapse will publish the change to all subscribed databases. (Orchestration)
P1.3 Subscribed databases apply the change.
P1.4 This leads to sequential, causal or eventual consistency.
P2.1 In the new system, all change requests need to go through the Synapse service. The databases need to register at the extra Synapse service, which needs to be implemented too.
P2.2 The autonomy decreases since the synapse service has extended knowledge about the used data objects in the databases.
P2.3 The database technology in use must be understood by the extra synapse service.

### 3.3 Selected Solutions

In the upcoming sections the solutions will be compared in more depth. To be able to compare in a certain amount of time, the number of solutions will be reduced to three. The selection will consider the questions from section 1.

To answer the first question, the solution which guarantees **the strongest consistency** is chosen. The strongest consistency is guaranteed by the 'Shared Database' and 'Listening to communication - Verification' solutions. Both solutions are not complete in terms of enforcing consistency. As the 'Shared Database' solution allows (internal) changes without emitting messages beforehand, it is chosen.

When converting a monolith system to a microservice system, it maybe a question how to keep the new system consistent. And since this conversion is a lot of work, it's helpful to save some time. So the second solution will be the solution that, when implementing, needs the least work. Or in the case of an already existing microservice system, the least changes. **The**

[1] See code at https://github.com/nviennot/synapse

**least changes when implementing in existing environments** can be achieved when solution specific changes are not within the services or within the databases, since these layers may already have been through a lot of time and effort. So the solution should be external. The solutions in this area are numerous: 'SAGA Choreography with Events', 'SAGA Orchestration', 'Listening to communication - Verification' and 'Listening to communication - Templates'. The last two solutions need a lot of knowledge about the services and would need a lot of work to implement. 'SAGA Choreography with Events' would need extra work within the services, so they are able to read event messages. 'SAGA Orchestration' is chosen as it would not need such work since it uses the already existing external service interfaces for interaction.

The second question how **Orchestration or Choreography** is better to achieve consistency, is done with choosing one solution that exists in both variants. As 'SAGA Orchestrator' is an already chosen orchestration solution, the chosen choreography variant is 'SAGA Choreography with Events'. It also promises **the highest autonomy** along with 'Listening to communication - Templates'. Furthermore, the usage of events is intuitive in the microservice environment as they scale well and provide loose coupling between sender & receivers [11].

# 4 Comparison

The **'Shared Database'** solution can lead to a strict consistency in the system. Furthermore, it is easy to understand, implement, test and debug. The problem of this solution is, that the location and structure of common data must be shared between the microservices, which interferes with the microservice autonomy. Also, this solution limits the choices of database technologies, since the database must work with all used languages and frameworks.

The **'SAGA Orchestration'** solution doesn't require any internal changes of the microservices. The solution is expected to be easy to implement, test and debug. In contrast, there are changes needed on the user side of the system, as all change requests need to be sent to the Orchestrator instead of sending them to the services directly. Depending on the current state of the architecture, implementation and how the users interact with the services, this is a big problem or a little one. Also, the 'SAGA Orchestrator' can be seen as a potential bottleneck for the entire system as all write requests must go through it.

The highest autonomy is achieved with the **'SAGA Choreography with events'** solution which scales well and provide loose coupling of the used microservices. The main problem of this solution is, that it only guarantees 'Eventual Consistency', the weakest consistency model. It can also be

hard to test and debug, referring to its asynchronous nature. There is also the challenge to know when all services have got the change messages, to monitor & control this, an extra service maybe necessary. Another challenge we see is that the services have to forward the changes to other services. This behaviour may lead to a hardcoded implementation of a network if not done with indirect communication. This is the opposite of a flexible and failure tolerant system.

**Orchestration** has the benefit of having all logic in one place, this makes implementation, testing, debugging and changes easier. It also makes it easier to detect service failure and can start a failure specific recovery. One Orchestrator could be a bottleneck for the system, but if implemented stateless, it can be easily replicated. Another problem is, that this Orchestrator service is an additional application that has to be implemented and adjusted to changes. And in general, more orchestration and sharing of resources (e.g. database) shifts the architecture to a more 'Service Oriented Architecture'.

The benefit of using **Choreography** is that it is expected to be faster than Orchestration, since the logic and work is distributed between the services. Also, the services are more autonomous from each other. In the 'SAGA Choreography with Events' events are specified globally to signal other that certain data has changed. Changes to these messages may lead to additional code changes in the services. Challenges are that this distribution also makes failure detection and recovering harder. It is also more difficult than in the Orchestration version to determine the point in time when all services got the change request and successfully updated their data.

The pros and cons of both abstract patterns are valid vice versa.

# 5 Conclusion

Three solutions to guarantee consistency in a microservice environment were presented. First 'Shared Database' as it guarantees the strongest consistency, second 'SAGA Orchestrator' as it needs the least changes when implementing in an existing system and third 'SAGA Choreography with Events' as Events are a frequently used in microservice architecture and to have a comparison between orchestration and choreography.

A higher consistency directly interferes with the microservice autonomy. For the purpose of having a global consistent state, microservice internal knowledge (data or data structures) needs to be shared across applications. Changes to internal s lead to changes of other applications too, which is the opposite of autonomy. How much knowledge needs to be shared depends on the selected solution and implementation.

If the distributed microservice system needs a higher consistency model, the applications become closer and share more knowledge. In the extreme this can lead to a shared orchestrator or shared database which in combina-

tion brings it very close to a 'Service-Oriented Architecture'. This behaviour can be explained as microservices were created for performance and specific scaling which benefits from loose coupling and high autonomy.

It can also be observed that high autonomy leads to systems that are harder to implement, test and debug since they may use asynchronous and indirect communication.

# 6 Future Work

First the selected solutions could be implemented for further and deeper testing. These implementations could then be used for performance testing or how well the solutions scale vertically / horizontally, this is especially interesting as scaling is important for microservice architectures. Another idea is to test how resilient the solutions are to failures.

# References


[1] Nils Barnickel and Matthias Fluegge. Transferring the principle of loose coupling to the semantic level. In *Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications*, ISWSA '10, pages 5:1–5:6, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1874590.1874595`, `doi:10.1145/1874590.1874595`.

[2] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. Contextual understanding of microservice architecture: Current and future directions. *SIGAPP Appl. Comput. Rev.*, 17(4):29–45, January 2018. URL: `http://doi.acm.org/10.1145/3183628.3183631`, `doi:10.1145/3183628.3183631`.

[3] Jacob Donham. A domain-specific language for microservices. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, pages 2–12, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3241653.3241654`, `doi:10.1145/3241653.3241654`.

[4] Alan Fekete, Paul Greenfield, Dean Kuo, and Julian Jang. Transactions in loosely coupled distributed systems. In *Proceedings of the 14th Australasian Database Conference - Volume 17*, ADC '03, pages 7–12, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. URL: `http://dl.acm.org/citation.cfm?id=820085.820089`.

[5] Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, December 1987. URL: `http://doi.acm.org/10.1145/38714.38742`, `doi:10.1145/38714.38742`.

[6] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. *SIGPLAN Not.*, 51(1):371–384, January 2016. URL: http://doi.acm.org/10.1145/2914770.2837625, doi:10.1145/2914770.2837625.

[7] Paul Greenfield, Dean Kuo, Surya Nepal, and Alan Fekete. Consistency for web services applications. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1199–1203. VLDB Endowment, 2005. URL: http://dl.acm.org/citation.cfm?id=1083592.1083731.

[8] João Loff, Daniel Porto, Carlos Baquero, João Garcia, Nuno Preguiça, and Rodrigo Rodrigues. Transparent cross-system consistency. In *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '17, pages 8:1–8:4, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3064889.3064898, doi:10.1145/3064889.3064898.

[9] Chris Richardson. *Microservice Patterns:*, volume 1. November 2018. URL: https://microservices.io/book.

[10] Nicolas Viennot, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. Synapse: A microservices architecture for heterogeneous-database web applications. *Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015*, 04 2015. doi:10.1145/2741948.2741975.

[11] Eberhard Wolff. *Microservices : Grundlagen flexibler Softwarearchitekturen* -. Dpunkt-Verlag, Köln, 1. auflage edition, 2015.