Consistency for Microservices
A Legacy Insurance Core Application Migration Example

Arne Koschel
Andreas Hausotter

Hochschule Hannover
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science
Hannover, Germany
Email: arne.koschel@hs-hannover.de
Email: andreas.hausotter@hs-hannover.de

Moritz Lange
Phillip Howeihe

Hochschule Hannover
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science
Hannover, Germany
Email: moritz.lange@stud.hs-hannover.de
Email: phillip.howeihe@stud.hs-hannover.de

***Abstract*—In microservice architectures, data is often hold redundantly to create an overall resilient system. Although the synchronization of this data proposes a significant challenge, not much research has been done on this topic yet. This paper shows four general approaches for assuring consistency among services and demonstrates how to identify the best solution for a given architecture. For this, a microservice architecture, which implements the functionality of a mainframe-based legacy system from the insurance industry, serves as an example.**

Keywords–Microservices; Consistency; Insurance Industry.

I. INTRODUCTION

A current trend in software engineering is to divide software into lightweight, independently deployable components. Each component of the system can be implemented using different technologies because they communicate over standardized network protocols. This approach to structure the system is known as the microservice architectural style [1].

Typical goals of a microservice architecture are an overall resilient system and independent scalable components. To reach these goals, it is beneficial to decouple the services as much as possible. Therefore, according to Martin Fowler, each microservice should have its own data management [1]. Furthermore, services often hold redundant versions of data records to be able to operate independently. The synchronization of these data records however is a significant challenge. In context of our research we identified four general approaches for assuring consistency among services. This paper presents these and demonstrates how to identify the best solution for a given architecture. A migrated application from the insurance industry serves as an example.

The *Competence Center Information Technology and Management* (CC_ITM) is an institute at the University of Applied Sciences and Arts Hanover. Main objective of the CC_ITM is the transfer of knowledge between university and industry. As part of ongoing cooperation between the CC_ITM and two regional insurance companies, the research project *Potential and Challenges of Microservices in the Insurance Industry* was carried out. The goal was to examine the suitability of microservice architectures for the insurance industry. One part of this project was the migration of a monolithic mainframe-based core application, namely the `Partner Management System`. The resulting microservice architecture holds some data records redundantly and is hereby a good object for scientific research in context of consistency assurance.

The remainder of this article is organized as follows: After discussing related work in Section II, we show the core application system and address issues with the monolithic approach in Section III. Section IV introduces the architecture of the migrated system. In Section V we evaluate the outcomes with a focus on consistency aspects. Section VI discusses general approaches to ensure consistency in microservice architectures and how these approaches can be applied to get a suitable consistency solution for the `Partner Management System`. Section VII summarizes the results and draws a conclusion.

II. RELATED WORK

The basis of our research is the literature of well-known authors in the field of microservices. Worth mentioning are the basic works of Martin Fowler and James Lewis [1] as well as those of Eberhard Wolff [2]. For practical parts of our research, mainly the elaborations of Sam Newman (see [3]) were used. Especially for the migration of the legacy application, the works of Knoche and Hasselbring (see [4]) were consulted. As a study from the year 2019 shows (see [5]), microservice architectures are barely used in the insurance and financial services industry in Germany. Therefore, results from other industries had to be used for our research (for example [6]).

Although the basic literature is extensive, not much scientific research has been done about synchronizing services. Because microservices should use independent database schemes and can even differ in persistence technology, the traditional mechanisms of replicating databases (see, e.g., Tanenbaum and Van Steen [7, chap. 7]) cannot be applied as well. Instead, ideas and patterns from other areas of software engineering had to be transferred to the context of microservices.

So, in addition to general microservices research, more fundamental concepts of operating systems (e.g. [8]) and object-oriented programming (e.g. [9]) were considered by our research. Furthermore, concepts of general database research like the SAGA-Pattern [10], which was already applied to microservices by Chris Richardson [11], were considered as

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-702-3

well. Event-based approaches like Event Sourcing, described by Fowler [12], were also applied to microservices within our research.

Previous work has already evaluated the outcomes of the research project mentioned in the introduction (see [13]). However, the project and the evaluation based on it did not discuss the issue of consistency in detail. For this reason, our research focused on this topic after the completion of the project. In addition, a bachelor thesis written by one of the project members further dealt with the issue of consistency assurance in context of the project results (see [14]).

This paper gives an overview of existing research and summarizes it in a single document. To the best knowledge of the authors, this is the first summarizing scientific work on this topic.

III. Core Application: Partner Management System

As mentioned in the introduction, one part of the research project was to design a system for managing partners of an insurance company - the `Partner Management System`. In this context, partners are defined as natural or legal persons who are in a relation with the insurance company (e.g., clients, appraisers, lawyers or other insurance companies). Additionally, to general information about the person, a partner may also have information on communication, bank details, business relations and relations with other partners. Figure 1 shows the domain model and additional information about the service design, which is covered in the next section. This section focuses on the domain model, which was developed in cooperation with the insurance companies and is strongly based on the reference architecture for German insurance companies (VAA) [15].

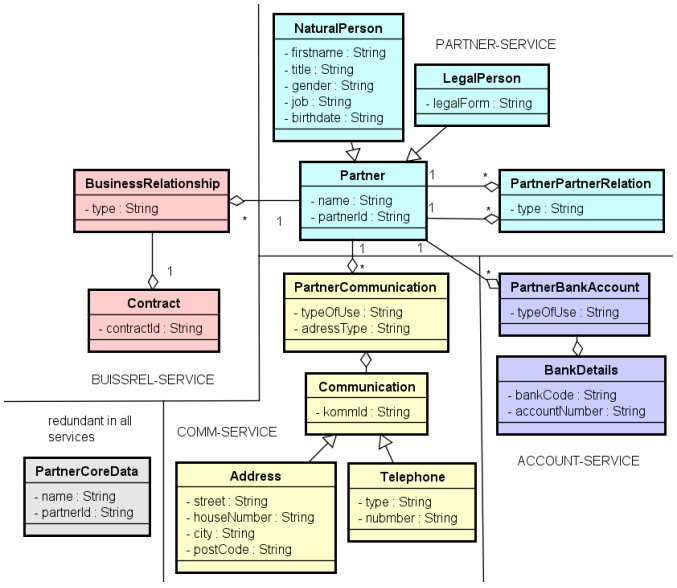

Figure 1. Domain model of the system.

At its core, the system is a simple CRUD application that manages the entity *Partner* and its properties. Usually, this functionality is implemented in a single SOA service. Modularization through SOA services provides significant benefits to the overall system (e.g., scalability and resilience), but the `Partner Management System` is still an atomic deployment unit that scales as a whole and fails as a whole. Since this system is a fundamental service of an insurance company, different parts of it are subject to varying levels of stress. Especially at night, the load profile differs. While only minor changes are made to existing datasets during the day, low occupancy during the night is used to slowly persist all new datasets collected on the day so as not to overburden the mainframe-based application. However, in practice this approach makes crashes at night extremely critical as the entire system does not work all night and only few people are available to fix the problem. The major issue of the existing implementation of the `Partner Management System` is obviously the poor flexibility, scalability and fault tolerance. This makes a microservices approach attractive for this use case.

IV. Microservice Architecture: Partner Management System

Figure 2 shows the architecture developed in close cooperation with the insurance companies, which subdivides the application into four independent services. Such a separation allows parts of the system to be scaled independently. Such a separation allows parts of the system to be scaled independently. For example, when the insurance company collects monthly premiums from its customers, this results in an increased load on the system that can be responded to by the scaling of the `account-service`.

To determine the separation into services, the original domain was divided into subdomains (as shown in Figure 1) and then the specific requirements (e.g., resilience) of each subdomain were analyzed. In order to make good use of the microservice architecture, the subdomains must be as independent as possible. That means there must be use cases where a subdomain can be used without any other. To achieve this independence, the architecture keeps certain parts of the partner (*PartnerCoreData*) redundant in all domains. This corresponds to the creation of bounded contexts, as described by Evans [16].

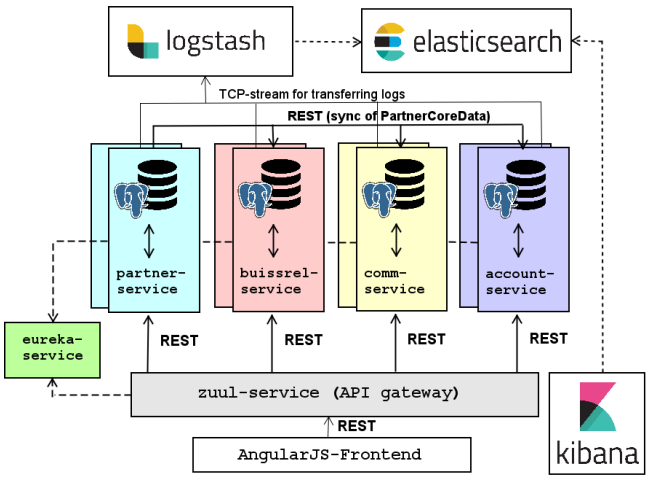

Figure 2. Infrastructure of the system.

Based on the technical specifications of the companies involved, the resulting subdomains were implemented as REST web services (see Figure 2) in Java using the Spring framework. As mentioned, each service should have its own data management, here realized as dedicated PostgreSQL databases. The *PartnerCoreData* is kept in sync across all services using REST calls of the `partner-service`. Parts of the Netflix OSS stack were used for the system infrastructure: Netflix Eureka (`eureka-service`) as a service discovery and Netflix Zuul (`zuul-service`) as an API gateway. Zuul also provides the web frontend of the application, which was realized as a single-page application using AngularJS. The ELK stack (Elasticsearch, Logstash and Kibana) was set up for monitoring and logging. All shown components of the architecture are deployed in separate Docker containers and connected by a virtual network using Docker Compose. In combination with the stateless architecture of the services, it is possible to run any number of instances of each service in a separate Docker container.

V. Challenges of the microservice architecture

Looking at the architecture described in sections III and IV, it looks like the microservice architecture can solve the problems of the current implementation. In particular, the scalability and fault tolerance of individual parts of the system are a crucial advantage over the current solution. The system can adapt to the changing load during the day, eliminating the need for risky nightly batch jobs. With the benefits of finer granularity however, there are also many new challenges that need to be mastered. One major challenge, for example, is the distributed monitoring and logging, which is handled by the ELK stack. As already mentioned, another key challenge of the developed microservice architecture is the consistency assurance across the services. More specifically, the synchronization of the *PartnerCoreData*, which serves as an example for the application of our research results.

As mentioned briefly in Section IV, the synchronization is realized by REST calls of the `partner-service`. Whenever a *PartnerCoreData* record is created, deleted or changed, the `partner-service` distributes this information to the other services in a synchronous way. This means that the `partner-service` is responsible for ensuring the consistency of the overall system. Furthermore, the development team of the `partner-service` is responsible for the data model of the *PartnerCoreData*, because it is part of their bounded context. This approach is known as Customer/Supplier Development (described by Evans [16]) in the context of domain-driven design.

This ensures that services are as independent of each other as possible. Even if, e.g., the `partner-service` is unavailable, the other services can still resolve foreign key relationships to partner data, because they keep a redundant copy of it. Moreover, the system reduces service-to-service calls, because other services don't need to call the `partner-service` on every operation. This ensures loose coupling of services, which is a key aspect of microservice architectures [1].

Since the synchronization of the *PartnerCoreData* is a critical part of the application, its implementation must be closer discussed. Since the goal of the first phase of the project was building the architecture in general, a synchronous solution was chosen for simplicity. This has several drawbacks:

- **Fault tolerance.** If the `partner-service` crashes during synchronization, some services might not be notified about the changes. Conversely, if another service can not be contacted by the `partner-service`, it will also not be notified. This is due to the transient communication.
- **Synchronicity.** After a change of *PartnerCoreData*, a thread of the `partner-service` is in a blocked state until all other services have been notified. Because multiple network calls are necessary for the synchronization, the general performance of the `partner-service` is affected. Since microservices should be lightweight, a large number of network calls and busy threads are a serious problem.
- **Extensibility.** Since the extension of a microservice architecture is a common occurrence, extensibility is a major aspect. In the current implementation, the `partner-service` holds a static list of services that need to be notified upon a change of *PartnerCoreData*. If a new service is added to the system, which is interested in *PartnerCoreData*, the `partner-service` must be redeployed. Additionally, the bigger the number of services to notify gets, the more the performance of the `partner-service` is impaired.

As part of our research, further alternative solutions were explored, which will be discussed in the next sections.

VI. Consistency assurance in the Partner Management System

In order to find a suitable solution for the specific problem of synchronizing *PartnerCoreData*, the general approaches have to be examined.

A. General approaches

The central research question to identify general approaches for consistency assurance is how a change in master data can be distributed to other interested services without breaking general microservices patterns like loose coupling and decentral data management. Especially the latter makes this a major challenge: Because the data stores and schemes should be separated, the standard mechanisms of synchronizing databases cannot be used here.

Based on our research, there are four possible solutions for synchronizing redundant data in microservices:

- **Synchronous distribution.** One approach is that the owner of the data distributes every change to all interested services. As discussed in Section V, the developed microservices architecture already follows this approach. To provide loose coupling however, the addresses of services to notify should not be contained in the master service's code. A better solution is to hold those addresses in configuration files, or even better, establish a standard interface where services can register themselves at runtime. For example, an existing service registry (e.g. Netflix Eureka) can be used to store this information. This approach roughly corresponds to the *Observer-Pattern* of object orientated software development.

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-702-3

As already discussed, notifying a large number of interested services might cause significant load of the service containing the master data. This can become a disadvantage. The distribution takes place in a synchronous fashion however, directly after the change of data itself. This means that this solution provides a high degree of consistency among services.

- **Polling.** One other solution might be to relocate the responsibility of aligning the redundant data to the interested services themselves. A straight forward approach here is to periodically ask for new data using an interface provided by the service containing the master data. Based on timestamps, multiple data updates can be transferred in one go. The size of the inconsistency window can be controlled by each interested service independently via the length of the polling interval. However, despite being consistent in the end, the time frame in which the data sets might differ is a lot larger than the one when using a synchronous solution. This model of consistency is known as eventual consistency (see [17]).
- **Publish-Subscribe.** To completely decouple the service containing the master data from the other services, message-oriented middleware can be used. On every data change, an event is broadcasted on a messaging topic following the publish-subscribe-pattern. Interested services subscribe to this topic, receive events and update their own data accordingly. Multiple topics might be established for different entities. If the messaging system is persistent, it even makes the architecture robust against services failures. This approach suits the resilient and lightweight nature of microservices. It must be noted however that it also falls in the category of eventual consistent solutions - until the message is delivered and processed, the system is in an inconsistent state.
- **Event Sourcing.** Instead of storing the current application state, for some use cases it might be beneficial to store all state transitions and accumulate those to the current state when needed. This approach can also be used to solve the problem of distributing data changes. Upon changes in master data, events are published. Unlike the publish-subscribe solution however, the history of all events is persistent in a central, append-only event storage. All services can access it and even generate their own local databases from it, each fitting their respective bounded context. This solution provides a high degree of consistency: Each data change can be seen immediately by all other components of the system. It must be noted that a central data store, which microservices try to avoid, is introduced. This weakens the loose coupling and might be a scalability issue - the append-only nature of the data storage enables high performance though.

If none of the consistency trade-offs above is bearable, this can be an indicator that the determined subdomains are not optimal. In some cases, subdomains are coupled so tightly that keeping data redundantly is not feasible. In this case, it should be discussed if the services can be merged. Furthermore, if this issue occurs in several parts of the architecture, it should be evaluated if a microservices approach is the right choice for this domain.

B. Approaches for synchronizing PartnerCoreData

The discussion in the previous section has shown that some approaches tend to guarantee a stronger level of consistency then others. This means that before all non-functional requirements can be considered as decision criteria, the required consistency degree of the underlying business processes must be examined. This can be done by first specifying the possible inconsistent states and then combining them with typical use cases of the system.

In case of the `Partner Management System`, only the *PartnerCoreData*, containing the name and id of every *Partner*, is saved in a redundant fashion. Combining these with the CRUD-operations, the following inconsistent states are possible:

- A new *Partner* might not yet be present in the whole system.
- Name or Id might not be up-to-date.
- A deleted *Partner* might not yet be deleted everywhere.

As part of our research, we combined these inconsistent states with typical use cases and business processes in which the `Partner Management System` is involved, like sending a letter via mail or a conclusion of an insurance contract.

The result of this examination is that the partner management of insurance companies is surprisingly robust against inconsistent states. This is mainly due to the reason that the business processes itself are already subject to inconsistency: If a customer changes its name for example, the inconsistency window of the real world is much larger than the technical one (the customer e. g. might not notify the insurance company until several days have passed). The postal service or bank already needs to cope with the fact that the name might be inconsistent. The discussion of other potential situations brought similar results. This makes sense because the business processes of insurance companies originated in a time without IT, which means that they are already designed resilient against delays and errors caused by humans. Cases where the customer notices the delay (e.g., a wrong name on a letter) are rare and justifiable.

In summary, the combination of the inconsistent states and the use cases of the `Partner Management System` revealed, that a solution which promotes a weaker consistency model can be used - no approach has to be excluded beforehand. So, the choice of a synchronization model is only influenced by the non-functional requirements. The analysis of the partner domain showed that the main non-functional requirements are lose coupling, high scalability and easy monitoring. Especially because of loose coupling, the publish-subscribe pattern is the most viable solution.

VII. Conclusion and Future Work

Synchronizing redundant data across services is a key challenge of microservice architectures. However, our research showed that the solutions can be reduced to four general approaches. For choosing a suitable solution for a given

architecture, the underlying domain needs to be analyzed: Possible inconsistent states need to be combined with typical use cases. As the example of the `Partner Management System` shows, this combination can reveal that domains might be much more resilient against inconsistencies as one would first assume.

The next steps will be to work out a specific design of the publish-subscribe approach for the `Partner Management System`. Furthermore, the complete implementation must be done, and the system must be tested under real-world conditions.

To demonstrate the procedure we have developed for finding a suitable consistency assurance solution, the example of the `Partner Management System` is sufficient. To further underpin our findings however, they need to be applied to more complex examples.

REFERENCES

[1] M. Fowler and J. Lewis, "Microservices a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, March 2014, [retrieved: 04, 2019].

[2] E. Wolff, Microservices: Flexible Software Architecture. Addison-Wesley Professional, 2016.

[3] S. Newman, Building microservices: designing fine-grained systems. O'Reilly Media, Inc., 2015.

[4] H. Knoche and W. Hasselbring, "Using microservices for legacy software modernization," IEEE Software, vol. 35, no. 3, 2018, pp. 44–49.

[5] ——, "Drivers and barriers for microservice adoption–a survey among professionals in germany," Enterprise Modelling and Information Systems Architectures (EMISAJ), vol. 14, 2019, p. 10.

[6] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). IEEE, 2017, pp. 243–246.

[7] A. S. Tanenbaum and M. Van Steen, Distributed Systems: Pearson New International Edition - Principles and Paradigms. Harlow: Pearson Education Limited, 2013.

[8] A. S. Tanenbaum, Modern Operating Systems. New Jersey: Pearson Prentice Hall, 2009.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software. Amsterdam: Pearson Education, 1994.

[10] H. Garcia-Molina and K. Salem, "Sagas," vol. 16, no. 3. ACM, 1987.

[11] C. Richardson, Microservices Patterns: With examples in Java. Manning Publications, 2018.

[12] M. Fowler, "Event Sourcing," https://martinfowler.com/eaaDev/EventSourcing.html, December 2005, [retrieved: 04, 2019].

[13] M. Lange, A. Hausotter, and A. Koschel, "Microservices in Higher Education - Migrating a Legacy Insurance Core Application," in 2nd International Conference on Microservices (Microservices 2019), Dortmund, Germany, 2019, https://microservices.fh-dortmund.de/papers/Microservices_2019_paper_8.pdf [retrieved: 04, 2019].

[14] P. Howeihe, "Transactions and consistency assurance in microservice architectures using an example scenario from the insurance industry," bachelor thesis at Univ. of Applied Sciences and Arts Hanover, 2018.

[15] GDV, "The application architecture of the insurance industry - applications and principles," 1999.

[16] E. J. Evans, Domain-driven Design - Tackling Complexity in the Heart of Software. Boston: Addison-Wesley Professional, 2004.

[17] W. Vogels, "Eventually Consistent," Commun. ACM, vol. 52, no. 1, Jan. 2009, pp. 40–44. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432

Copyright (c) IARIA, 2019. ISBN: 978-1-61208-702-3