SERVICE COMPUTATION 2020

The Twelfth International Conferences on Advanced Service Computing

ISBN: 978-1-61208-777-1

October 25 - 29, 2020

SERVICE COMPUTATION 2020 Editors

Andreas Hausotter, Hochschule Hannover - University of Applied Sciences and Arts, Germany

SERVICE COMPUTATION 2020

Forward

The Twelfth International Conferences on Advanced Service Computing (SERVICE COMPUTATION 2020), held on October 25 - 29, 2020, continued a series of events targeting computation on different facets.

The ubiquity and pervasiveness of services, as well as their capability to be context-aware with (self-) adaptive capacities posse challenging tasks for services orchestration, integration, and integration. Some services might require energy optimization, some might require special QoS guarantee in a Web-environment, while others a certain level of trust. The advent of Web Services raised the issues of self-announcement, dynamic service composition, and third party recommenders. Society and business services rely more and more on a combination of ubiquitous and pervasive services under certain constraints and with particular environmental limitations that require dynamic computation of feasibility, deployment and exploitation.

The conference had the following tracks:

- Service innovation, evaluation and delivery
- Service quality
- Challenges
- Advanced Analysis of Service Compositions

Similar to the previous edition, this event attracted excellent contributions and active participation from all over the world. We were very pleased to receive top quality contributions.

We take here the opportunity to warmly thank all the members of the SERVICE COMPUTATION 2020 technical program committee, as well as the numerous reviewers. The creation of such a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and effort to contribute to SERVICE COMPUTATION 2020. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the SERVICE COMPUTATION 2020 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope SERVICE COMPUTATION 2020 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the area of computation.

SERVICE COMPUTATION 2020 Steering Committee

Paul Humphreys, Ulster Business School/University of Ulster, UK
Arne Koschel, Hochschule Hannover, Germany
Annett Laube, Bern University of Applied Sciences (BUAS), Switzerland
Eugen Borcoci, University "Politehnica" of Bucharest, Romania
Andreas Hausotter, Hochschule Hannover - University of Applied Sciences and Arts, Hannover, Germany

SERVICE COMPUTATION 2020 Publicity Chair

Javier Rocher, Universitat Politecnica de Valencia, Spain

SERVICE COMPUTATION 2020 Industry/Research Advisory Committee

Steffen Fries, Siemens Corporate Technology - Munich, Germany
Matthias Olzmann, noventum consulting GmbH - Münster, Germany
Rong N. Chang, IBM T.J. Watson Research Center, USA
Jan Porekar, SETCCE, Slovenia
Bernhard Hollunder, Hochschule Furtwangen University - Furtwangen, Germany

SERVICE COMPUTATION 2020

Committee

SERVICE COMPUTATION 2020 Steering Committee

Paul Humphreys, Ulster Business School/University of Ulster, UK
Arne Koschel, Hochschule Hannover, Germany
Annett Laube, Bern University of Applied Sciences (BUAS), Switzerland
Eugen Borcoci, University "Politehnica" of Bucharest, Romania
Andreas Hausotter, Hochschule Hannover - University of Applied Sciences and Arts, Hannover, Germany

SERVICE COMPUTATION 2020 Publicity Chair

Javier Rocher, Universitat Politecnica de Valencia, Spain

SERVICE COMPUTATION 2020 Industry/Research Advisory Committee

Steffen Fries, Siemens Corporate Technology - Munich, Germany
Matthias Olzmann, noventum consulting GmbH - Münster, Germany
Rong N. Chang, IBM T.J. Watson Research Center, USA
Jan Porekar, SETCCE, Slovenia
Bernhard Hollunder, Hochschule Furtwangen University - Furtwangen, Germany

SERVICE COMPUTATION 2020 Technical Program Committee

Jocelyn Aubert, Luxembourg Institute of Science and Technology (LIST), Luxembourg
Carlos Becker Westphall, Federal University of Santa Catarina, Brazil
Eugen Borcoci, University "Politehnica" of Bucharest, Romania
Uwe Breitenbücher, University of Stuttgart, Germany
Antonio Brogi, University of Pisa, Italy
Isaac Caicedo-Castro, Universidad de Córdoba, Colombia
Wojciech Cellary, Poznan University of Economics, Poland
Rong N. Chang, IBM T.J. Watson Research Center, USA
Dickson Chiu, The University of Hong Kong, Hong Kong
Leandro Dias da Silva, Universidade Federal de Alagoas, Brazil
Erdogan Dogdu, Angelo State University, USA
Sebastian Floerecke, University of Passau, Germany
Sören Frey, Daimler TSS GmbH, Germany
Steffen Fries, Siemens Corporate Technology - Munich, Germany
Somchart Fugkeaw, Thai Digital ID Co. Ltd., Thailand
Katja Gilly, Miguel Hernandez University, Spain
Victor Govindaswamy, Concordia University - Chicago, USA
Maki Habib, The American University in Cairo, Egypt
Andreas Hausotter, Hochschule Hannover - University of Applied Sciences and Arts, Germany

Copyright Information

For your reference, this is the text governing the copyright release for material published by IARIA.

The copyright release is a transfer of publication rights, which allows IARIA and its partners to drive the dissemination of the published material. This allows IARIA to give articles increased visibility via distribution, inclusion in libraries, and arrangements for submission to indexes.

I, the undersigned, declare that the article is original, and that I represent the authors of this article in the copyright release matters. If this work has been done as work-for-hire, I have obtained all necessary clearances to execute a copyright release. I hereby irrevocably transfer exclusive copyright for this material to IARIA. I give IARIA permission or reproduce the work in any media format such as, but not limited to, print, digital, or electronic. I give IARIA permission to distribute the materials without restriction to any institutions or individuals. I give IARIA permission to submit the work for inclusion in article repositories as IARIA sees fit.

I, the undersigned, declare that to the best of my knowledge, the article is does not contain libelous or otherwise unlawful contents or invading the right of privacy or infringing on a proprietary right.

Following the copyright release, any circulated version of the article must bear the copyright notice and any header and footer information that IARIA applies to the published article.

IARIA grants royalty-free permission to the authors to disseminate the work, under the above provisions, for any academic, commercial, or industrial use. IARIA grants royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

IARIA acknowledges that rights to any algorithm, process, procedure, apparatus, or articles of manufacture remain with the authors and their employers.

I, the undersigned, understand that IARIA will not be liable, in contract, tort (including, without limitation, negligence), pre-contract or other representations (other than fraudulent misrepresentations) or otherwise in connection with the publication of my work.

Exception to the above is made for work-for-hire performed while employed by the government. In that case, copyright to the material remains with the said government. The rightful owners (authors and government entity) grant unlimited and unrestricted permission to IARIA, IARIA's contractors, and IARIA's partners to further distribute the work.

Table of Contents

Lightweight Offline Access Control for Smart Cars

Gian-Luca Frei
Zühlke Engineering AG
Bern, Switzerland
emails: hello@gianlucafrei.ch
gifr@zuehlke.com

Fedor Gamper
Swiss Federal Railways
Bern, Switzerland
email:
fedorgamper@outlook.com

Prof. Dr. Annett Laube
Bern University of Applied Sciences TI -ICTM
Biel/Bienne, Switzerland
email:
annett.laube@bfh.ch

***Abstract*—In this paper, a novel access control protocol that offers appealing features for carsharing is presented. It describes how a user can authenticate and authorize himself using a smartphone on an immobilizer in a car. First, it requires no online connection to open cars. Therefore, it is suitable for applications where the cars and the users have no network connection. Second, the protocol is designed for low-bandwidth channels like Bluetooth Low Energy and transports around 210 bytes per car access. Third, it enables users to delegate their access rights to other users. These properties were achieved by using custom public key certificates and authorization tokens with a public key recovery mechanism.**

Keywords– access control; authentication; authorization; bluetooth low energy; carsharing; cryptographic protocol; public-key cryptography; public-key recovery.

I. INTRODUCTION

Smartphones have become omnipresent devices. At the same time, the worldwide market for carsharing has grown exponentially over the last decade [1] [2]. As a result, many carsharing providers offer their customers the possibility to open rental cars with smartphones. Often, the security of such systems is unknown because the vendors keep the system design secret.

Designing access control solutions is quite easy if the cars have a stable network connection. The only parts needed are an authentication mechanism and a server that the car can query to check if a user is allowed to access it. However, maintaining a constant network connection is often not possible or not desirable because of the higher costs involved. Moreover, developing a protocol whereby all steps can be done with no network connection would not be meaningful in a world where most smartphones almost always have an internet connection. Therefore, we use the following networking model: The users are most of the time online and need a network connection for registration and to make bookings. Later, a user receives a credential that enables him to make use of his access rights in an offline fashion. This means that if he opens a car, he needs no network connection, nor does the car need a network connection. To illustrate this, imagine the car is located in an underground car park, where no cellular reception is available. In that case, the carsharing provider cannot communicate with the car, whereas a user can cross into the offline zone by entering the underground car park. Most carsharing providers allow their users to make spontaneous bookings over their smartphones. This means access control rules can change quickly. Therefore, the user needs to receive the credentials to authenticate and authorize himself outside before entering the underground car park.

The most convenient way to establish communication between a car and a smartphone is through either Bluetooth or Near-Field Communication (NFC) [3]–[5]. Bluetooth Low Energy (BLE) is part of the Bluetooth 4 specification and is designed to use little electric power [6]. Another advantage of Bluetooth Low Energy is that no device-pairing is needed. This makes Bluetooth Low Energy very convenient. NFC is also very convenient but is not fully supported on Apple devices [7]. This makes BLE a popular choice for real-world applications. A downside of BLE and NFC are that the transmission speed is low and often the theoretical bandwidth cannot be reached in practice. In our tests with Bluetooth LE, we measured a transmission speed of under 1,000 bytes per second [8]. It is, therefore, important to keep the sizes of the messages exchanged between the smartphone and the object as small as possible because large messages can have direct negative impacts on usability.

This paper is organized as follows: Section II presents the current state-of-the-art of access control protocols for carsharing. Then, in Section III the new protocol is presented. Section IV discusses possible ways to attack a system that uses the presented protocol. Finally, Section V concludes the paper.

II. STATE-OF-THE-ART

This section gives an overview of the existing work on carsharing systems. Dmitrienko et al. presented an offline access control system for free-float car sharing. This protocol is based on symmetric encryption, secure elements to store private credentials and a single carsharing provider that manages access rights [9]. Dmitrienko et al. also proposed a generic access control system based on NFC enabled devices which also supports offline validation and delegation of authorization. However, this work makes use of some proprietary protocols [10]. SePCAR is an access control protocol for smart cars. However, the focus of this protocol is more on user privacy than on bandwidth efficiency [11]. Mustafa et al. published a comprehensive requirements analysis for carsharing systems [12]. There exist also protocols that are not intended for carsharing but could also be used in this context. Grey is a research project which has been used to access physical space, computer logins and web applications based on asymmetric crypto on smartphones [13]. With Grey, users can pass their authorizations to other users. Arnosti et al. proposed a general physical access control system that uses NFC to communicate with digital and physical resources. However, their protocol requires a network connection between the resource and a central server [14]. Groza et al. explored the use of trusted platform modules along with identity-based signatures for vehicle access-control [15]. Ouaddah et al. developed an access control framework for internet of things applications based on blockchain technology [16]. Similar to public key recovery

 ISBN: 978-1-61208-777-1

which is used in the presented protocol is another technique called implicit certificates. Some IoT-Protocols make use of this technique, example are developed by Sciancalepore et al. and Ha et al. [17] [18]. Furthermore, there are many proprietary solutions mostly from carsharing companies, but no details are publicly available. Examples are from Zipcar, OTA-Keys, Continental Cars, and Valeo. To the best of our knowledge no prior work has focused on low bandwidth protocols for access control in carsharing. To fill this research gap, we propose a new lightweight access control protocol for offline cars.

III. PROTOCOL

In this section, the developed access control protocol is presented.

A. Overview

The protocol is based on the principle of authorization tokens and a strong authentication mechanism with public key certificates. Each user has a device containing a unique private key used for authentication and a public key certificate. Further, each user has one or multiple authentication tokens, which are digitally signed messages that link access rights to a specific user. These tokens are independent of the private key used for authentication and can be shared between multiple devices of one user. For example, if a user changes his smartphone, then he needs to onboard his new phone to generate a new private key and get a new public key certificate and then he can copy his authorization tokens to his new device. The user can then create an access request on his device. Each access request is authenticated with the private key and linked to the user with the public key certificate. Authentication with public key certificates is a widely used and secure authentication mechanism and removes many attack points because outsiders can forge access requests of regular users only with negligible probability. The most important advantage of an authorization token is that the car only needs to store a few public keys as a trust anchor. It uses these saved public keys to check access requests. This is useful for applications where the car is offline for a long time or the hardware of the car needs to be inexpensive.

a) Components and Roles: A user can access a car from different devices. A user device can be a smartphone, a smart card, or a computer. To use a new device, the user needs to introduce it to the system by performing the onboarding process with it. During device onboarding, a new private key is generated and an Identity Authority (IA) issues a public key certificate for the new device. The car needs to have a computing platform that communicates with the user device, validates the access requests and controls the immobilizer of the car. A car owner is a person who has administrative access to a car and configures its computing platform. There are two different authority roles. The IA checks the identity of users and issues public key certificates for new user devices. The Permission Authority (PA) issues authorization tokens. The cars can trust multiple authorities. Figure 1 visualizes the relationships between the different components. One party can be an IA and PA at the same time; however, the two roles can also be split between different parties. For instance, in peer-to-peer sharing, each car owner could trust a car-sharing platform to check identities and driving licenses but run a PA by himself. The protocol does not specify a user registration method. We assume that identity authorities have a way to manage users and that users can authenticate to identity authorities. The public key certificates are only used to authenticate a user to a car.

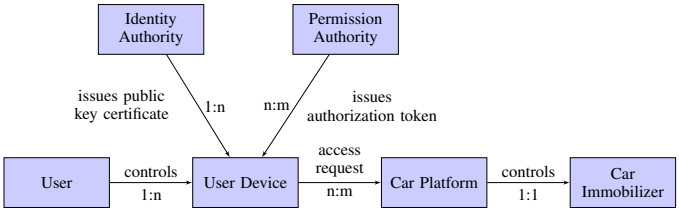

Figure 1. Components and Roles

b) Basic Description: Each user in the system owns an asymmetric key pair used for authentication. During the onboarding process, an Identity Authority signs a public key certificate for the user. Next, the user receives an authorization token which is signed by the Permission Authority. To access a car, the user signs an access request which contains a description of what he wants to do. This access request message, together with a certificate and authorization token, is then sent to the car. The car verifies the access request. If the car trusts all the involved authorities and the request is valid, it grants access. The car trusts a set of authorities by storing their public keys in a local trust store. An authorized user can delegate a subset of his access rights to another user. For instance, when a user booked a car, he might not use the car only by himself but wants that his travel companion is able to open the car too. To realize token delegation, all authorization tokens have a flag that indicates whether the authorized user is allowed to delegate his access rights to another user. To delegate an access right, an authorized user A signs a new token for another user B. User B then uses the chain of tokens, containing his token and the token of A, to claim or further delegate his access rights.

c) Cryptographic Primitives: The cryptographic primitives used are a cryptographic hash function $H(m)$ and the Elliptic Curve Digital Signature Algorithm (ECDSA) with public key recovery. $recoverPk(h, s)$ is a function which computes a public key pk which is valid for the digest h and signature s. ECDSA is one of the only digital signature schemes where this operation is possible [19] [20] [8].

B. Protocol Phases

In this section, we describe the different phases of the protocol. Generally, if any check fails, the process must be aborted. Figures 2 to 7 illustrate the different processes.

1) Car Initialization: (Figure 2) During the initialization process, a car owner CO adds a new car to the system by setting up the car platform CP. First, the CO sends the $SystemParameters$ consisting of the digital signature algorithm and the hash function to the CP. Next, he sends the set of trusted public keys PK_{IA}, PK_{PA} on the CP. These public keys are later used to check the authenticity of public key certificates and authorization tokens. Further, the CO should check if the clock to the CP is precise enough. How precise the clock must be can vary between different systems; however, the divergence should usually not exceed a few seconds. A precise time source is necessary because the car needs the current time to check if the access tokens are valid at the moment of usage.

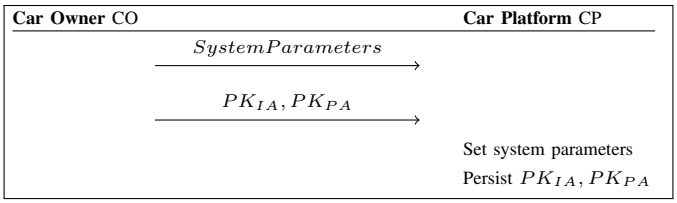

Figure 2. Car Initialization Sequence Diagram

2) Device Onboarding: (Figure 3) During onboarding, a user U sets up a new personal device D to later access a car. The device usually communicates with the authorities over an internet connection; therefore, minimal message sizes are not important during this process. The device needs to generate a new private key. Then it requests a public key certificate from an IA for the public key which belongs to the private key. This process needs a mutually authenticated channel between D and IA. D first generates a key pair (sk, pk) where sk is the platforms new private key and pk is the corresponding public key. D then generates a proof of knowledge of sk by signing $H(u, pk)$ with sk. Then it sends this signature s together with is username u and public key pk to the IA. This proof of knowledge is needed to prevent a user from getting a certificate for a key pair without the knowledge of the private key. The IA then checks this proof and then signs $s_C = sign_{skIA}(H(u, pk, v))$ where v is the validity period of the public key certificate and sends the certificate $cert = (u, v, s_C)$ to D. Note that the public key is hashed into the signed part of the certificate but not contained in the certificate itself. The public key can later be recovered and then the authenticity of the certificate can be checked.

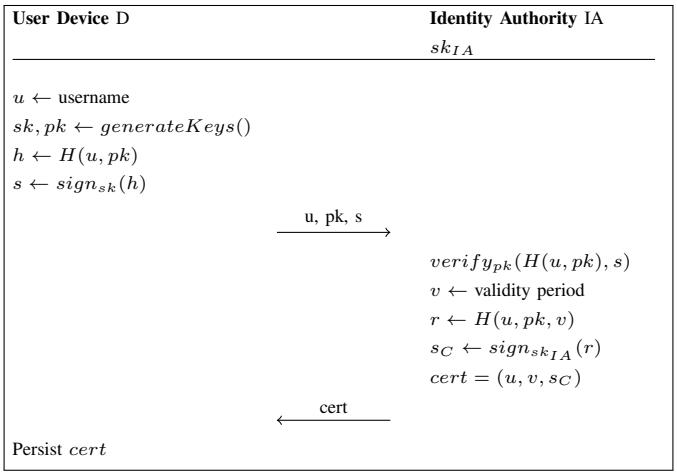

Figure 3. Device Onboarding Sequence Diagram

3) Root Token Issuance: (Figure 4) The result of the root token issuance process is an authorization token t. This token allows the user to claim access rights to a car or to delegate his rights to another user. If a token t_n is delegated, all tokens used $t_1, ..., t_{n-2}, t_{n-1}$ to delegate the last token t_n are needed to check the validity of the last delegated token. This sequence of tokens $t_1 - t_n$ is called a chain of tokens T. The first token t_1 in such a chain needs to be issued by a PA and is called the root token. To issue a root token, a PA signs an authorization token t and sends this token to the device D of the user. How the PA manages the access rules depends on the application of the protocol and is not specified. The token t is a signed message consisting of p, which is a description of the access rights of the user. It also contains a bit-flag d, which indicates if the user can delegate his access rights and a signature $s_{T1} = sign_{sk_{PA}}(H(u, p, d))$. To keep the protocol flexible, the content of p is not specified. It should contain a set of cars that the user may access and a validity timespan. The root token then is $t_1 = (p, d, s_{T1})$. The PA sends the sequence T_1 to the D of the user. D stores T_1 together with $C_1 = (cert)$ which is the sequence of the corresponding public key certificates. Note that the username u itself is part of the signed hash. However, it is not part of the token message like the public keys in certificates. The reason for this is that an authorization token must always be checked with the corresponding public key certificate containing the same username. Instead of testing if both usernames are the same, u can be taken from the public key certificate and so u can be omitted from the authorization token.

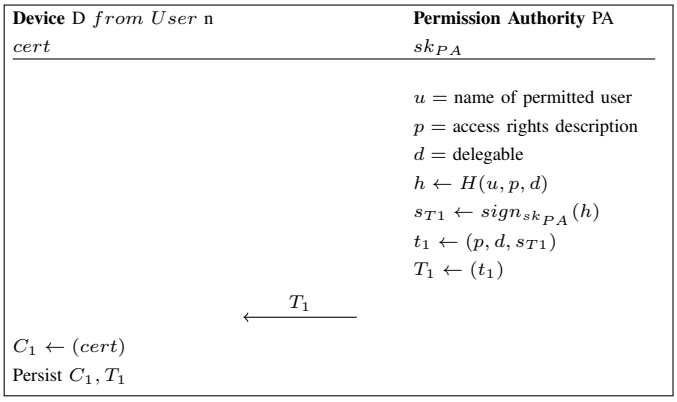

Figure 4. Root Token Issuance Sequence Diagram

4) Token Delegation: (Figure 5) To delegate a token, the delegating user with username u_{n-1} enters the name of the receiver u_n, the description of the rights he wants to delegate p_n and the flag d_n, which indicates if the new token can further be delegated to his device D. D then sign the new token t_n with the private key sk_{n-1} in the same way as in the root token issuance process, except that in the new sequence of tokens T_n, t_n is appended to the prior sequence of tokens T_{n-1}. D sends the new chain of tokens T_n and the prior chain of certificates C_{n-1} to the device of the receiver D'. D' appends his public key certificate to the chain of certificates C_{n-1}. The receiving user can further delegate his token, if the received token is delegable, by performing this process again.

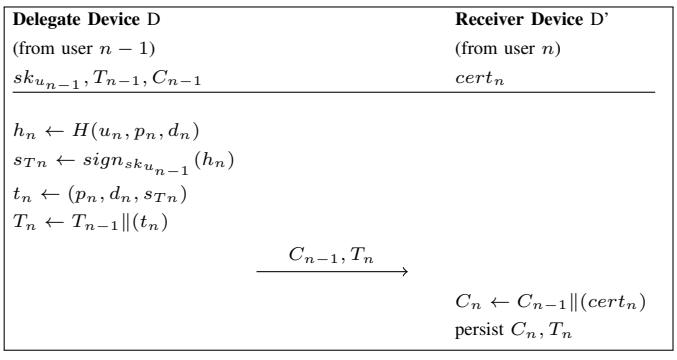

Figure 5. Token Delegation Sequence Diagram

5) Access Request: (Figure 7) To create an access request, the user needs a list of tokens $T = t_1, \ldots, t_{i-1}, t_i$ and a list of certificates C. Each t_i in T except t_1 must be signed by the public key corresponding to the certificate c_{i-1}. The root token t_1 must be signed by a Permission Authority. Also, the user of c_i and t_i must be the same. Figure 6 illustrates a chain of tokens and the corresponding certificates resulting from two delegations. The fields in parentheses are hashed into the signature but are not stored in the message. To access a car, the user U enters a description of the access request $desc$ into his device D. This is needed in case a car allows different accesses or needs additional information. For instance, the description can specify that the car should open the trunk only. For this description, together with, the current time τ and the name of the car r, D creates a signature $s = sign_{sk}(H(desc, \tau, r))$. Finally, τ, the list of tokens T, the list of certificates C, the description $desc$ and s are sent to the car platform CP. CP validates the chain of permissions according to the $validateRequest$ procedure (Figures 8 and 9) and if no check fails, it grants accesses to the car according to the description $desc$. The authentication mechanism depends on the hardness to forge the digital signature s. For an attacker, without the knowledge of the private key sk that belongs to the public key certificate, it is not realistic to forge a valid signature s. An attacker could record and try to replay an access request. To prevent this, the timestamp τ is also included in the signed part of the request. The car must check if the timestamp is close to the current time provided by the clock of the car platform. How small the derivation can be is a trade-off between susceptibility for timing errors and security.

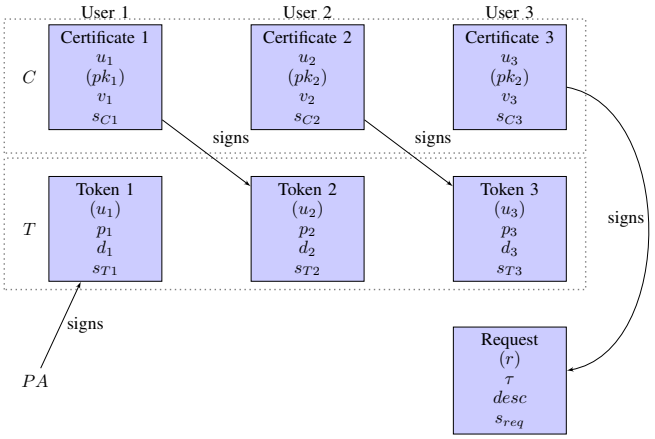

Figure 6. Access Request Example

C. Analysis of the Access Request Size

In this section, the size of the access request message is analyzed. This size is very important because the channel between the user device and car platform often has a low bandwidth.

a) Security Parameter: When implementing an application of this protocol, a security parameter S has to be chosen. This parameter is a way to define how difficult it should be for an attacker to break the cryptographic primitives of the application. More precisely: A polynomial bound attacker is expected to break the primitives in $O(2^S)$ computing steps. Nowadays a security parameter of about 112 is recommended to protect secrets for about 10 years [21] [22]. However, this

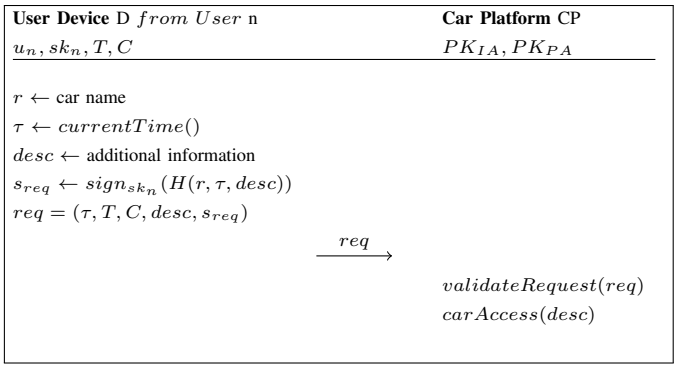

Figure 7. Access Request Sequence Diagram

$validateRequest(\tau, T, C, desc, s_{req}, PK_{IA}, PK_{PA})$

$n \leftarrow \mathsf{length}(C)$

/ Authentication Check
check $currentTime()$ is near τ
$r \leftarrow$ car name
$h \leftarrow H(r, \tau, desc)$
/ c_n^u means the username of the nth certificate
$signatureCheck(h, s_{req}, c_n, c_n^u, PK_{IA})$

/ Check if the access description is within the claimed access privileges
/ t_n^p means the privileges of the nth token
Check $desc$ is within t_n^p
Check context of t_n^p

/ Check the root token
/ t_i^s means the signature of the ith token
$h \leftarrow H(c_1^u, t_1^p, t_1^d)$
$pk_{PA} \leftarrow recoverPk(h, t_1^s)$
Check $pk_{PA} \in PK_{PA}$

/ Check the chain of tokens
for $i \leftarrow 2, 3, ..., n$
  $h \leftarrow H(c_i^u, t_i^p, t_i^d)$
  $signatureCheck(h, t_i^s, c_{i-1}, c_{i-1}^u, PK_{IA})$
  / t_n^p means the privileges of the nth token
  Check $t_i^p$are within t_{i-1}^p
  Check $t_i^u = c_i^u$
  Check $t_{i-1}^d = 1$

Figure 8. Validate Request Procedure

protocol only provides authentication and authorization. It does not provide confidentiality. The keys used to sign the messages are only valid for a specific time and should then be renewed. An attacker is not interested in breaking old keys. Therefore, a smaller security parameter is also possible when all keys (especially the authority keys) are regularly replaced by freshly generated keys. To achieve a security level of S a $2S$ bit curve must be used (for instance a 256 bit curve for 128 bit security).

$signatureCheck(h, s, cert, u, PK_{IA})$

$pk \leftarrow recoverPk(h, s)$
$r \leftarrow H(u, pk, cert_v)$
$pk_{IA} \leftarrow recoverPk(r, cert_s)$
Check $pk_{IA} \in PK_{IA}$
Check $currentTime()$ within $cert_v$

Figure 9. Signature Check Procedure

An ECDSA signature then has a size of $4S$ plus two bits for the recovery parameter, which are negligible for our calculations.

b) Size of an Access Request: An access request consists of at least one certificate and one token, each containing one digital signature. Additionally, we have one signature for the access request message itself. For each delegation, we need to add one certificate and one token. Let $s = 4S$ be the size of a digital signature. The total size of the access request is therefore $s + a + (d + 1)(2s + b)$ where d is the number of delegations. a is the size of the fields in the request (τ, $desc$), and b is the size of the fields in the certificate and token (username, validity, privileges). Because a and b are relatively small, we assume them to be $a \approx 20$ bytes and $b \approx 40$ bytes. Table I shows the calculated access request message sizes.

TABLE I. CALCULATED SIZE OF THE ACCESS REQUEST

		for S=100
No delegation:	$3s + a + b \approx 3s + 60B$	≈ 210 bytes
One delegation:	$5s + a + 2b \approx 5s + 100B$	≈ 350 bytes
Per additional delegation:	$2s + b \approx 2s + 50B$	≈ 140 bytes

D. Proof of Concept

To test the efficiency of protocol, we created a prototype which consists of a mobile application and a Raspberry Pi 3b+ running a Node.js application [23]. An iPhone X was used to transmit an access request over Bluetooth Low Energy (BLE) to the raspberry witch was simulating the car. It took in average 495 ms to transmit and compute an access request message with no delegations. This was measured in an environment with no other BLE devices nearby and 15 cm distance between the devices. With one delegation, the transmission and computation time increased to in average 566 ms and with two delegations to in average 732 ms. This value could be minimized by improving the computational performance of the prototype [8].

IV. ANALYSIS OF ATTACK VECTORS

This section discusses different vectors an adversary could use to attack a system using this protocol and how the protocol protects against such attacks.

a) Denial of Service: Communication channels over the air are in general vulnerable to a denial of service attacks. Therefore, an attacker could prevent an authorized user from accessing a car. For applications where availability is important, it is better to use a low range technology such as NFC instead of Bluetooth.

b) Man-in-the-middle: An attacker can capture a message in transit and forward it to the car, but since all part of the access request are authenticated he cannot alter it. However, if an attacker captures the request messages, it could have a privacy impact. For applications where privacy is important, it is recommended to encrypt the access requests in such a way that only the targeted car can decrypt it. Another type of attack would be when an attacker tries to extend the range of the communication channel between the user and the car. He could trick the user to unintentionally open a car. To prevent this, the user device should only send access requests when the user confirms that he is near the car.

c) Clock-Synchronization: If the time source of the car is not correct, a user with a valid token could access the car outside the validity timespan of the token. Also, the validity of the authentication certificates could be circumvented. It is therefore important that only the car owner can adjust the time source of the car. Depending on the application a small derivation of a few seconds can be unproblematic, but longer differences should be prevented.

d) Replay Attacks: An attacker can copy a transmitted access request and replay it later. However, the time stamp τ in the access request prevents the car from accepting the replayed access request because it compares τ to the current time.

e) Abuse of the Car: A malicious user who has access to a car could use it in an unintended way. For example a customer could try to manipulate the car's computing platform. To prevent this, the computing platform should be physically protected against such and similar manipulations, or at least able to detect it. Also, a user could rent a car and not return it on time. Such and similar attacks should be regulated in the general business terms of the carsharing provider.

f) Attacks on the Authentication Mechanism: An attacker could try to circumvent the authentication mechanism of the access request and impersonate another user. To do this, the attacker would need to forge a valid public key certificate of a trusted IA or forge a valid signature in the access request. Both types of attack are prevented by the difficulty of forging a digital signature.

g) Attacks on the Authorization Mechanism: An attacker which is a registered user could also try to circumvent the authorization mechanism by trying to forge a valid chain of authorization tokens. To do this, the attacker would need to forge either the signature of one authorization token or forge a public key certificate.

h) Attacks on the Devices of Other Users: An attacker could try to steal the private key of another user's device or make another user's device to sign a token or access request by installing malware on the victim's device. To prevent this, the users' device must be secured against such attacks. To make this attack more difficult, the private key should be stored on a trusted platform module with a key activation function, which most modern devices provide.

i) Attacks on the Identity Authorities: An attacker could try to attack an IA directly. An attacker that intruded into an authority could either steal the private key or make the authority to sign a public key certificate. The attacker then could impersonate any other user. Once the intrusion is detected, all car owners would need to remove the public key of the corrupted authority on all cars that trusted that authority.

 ISBN: 978-1-61208-777-1

To prevent such an attack, the IA must be secured against different cyberattacks to make such an attack unprofitable for an attacker.

j) Attacks on the Permission Authorities: Like the attack on the identity authorities, an attacker could also steal the private key of a PA. He then could issue arbitrary root tokens and could access all cars that trust the corrupted authority. However, to use the car, the attacker also needs to authenticate himself and therefore needs a public key certificate. This means that such an attack would only be interesting for an already registered user or in combination with another attack. Similar to the attack on the identity authorities, the PA must be secured against different cyber attacks.

To sum up, attacks on the authorities are the most promising attacks. As a result, these authorities must be well secured to mitigate such threats. Splitting up the privileges of the authorities also minimizes the impact of a successful attack.

V. Conclusion

The presented protocol enables carsharing providers to use a secure access control mechanism over Bluetooth Low Energy or Near-Field Communication. It is based on well-known security mechanisms but uses the less-known technique of public key recovery to reduce the size of the messages. The security mechanism of the protocol is based on the principle of public-key certificates and digitally-signed authorization tokens. Both mechanisms are well known and used for a wide variety of applications. The main novelty of our approach is the use of public-key recovery, to drastically reduce the message sizes of the custom certificates. The custom public key certificates we developed, have a size of only about 100 Bytes for a 256-bit key. Compared to traditional X.509 [24] or GPG [25] certificates for the same ECDSA key, this is about 5 to 10 times smaller. The networking model of this protocol assumes that only the car and the smartphone can communicate with each other during the access phase. A consequence of this model is that the revocation of an access right is not possible. If a carsharing service allows a user to open a car in places where no network connection is possible, it gives up the possibility to communicate with the car. Thus, the car cannot ask if the user's access rights have been revoked.

To the best of our knowledge, it is the first access control protocol that makes use of this technique. As a result, it has powerful and interesting properties, which makes it suitable for carsharing-applications. Nevertheless, the protocol could also be used in other domains such as building door systems. The developed prototype proves that the protocol runs fast on today's smartphones and is very convenient for the users.

References

[1] S. Le Vine, A. Zolfaghari, and J. Polak, "Carsharing: evolution, challenges and opportunities," Scientific advisory group report, vol. 22, 2014, pp. 218–229.

[2] S. Shaheen, E. Martin, and B. Apaar, "Peer-to-peer (p2p) carsharing: Understanding early markets, social dynamics, and behavioral impacts," 2018.

[3] S. International Organization for Standardization, Geneva, "Iso/iec 18092:2013," Tech. Rep., [retrieved: March, 2020]. [Online]. Available: https://www.iso.org/standard/56692.html

[4] R. Want, "Near field communication," IEEE Pervasive Computing, no. 3, 2011, pp. 4–7.

[5] K. Finkenzeller, RFID handbook: fundamentals and applications in contactless smart cards, radio frequency identification and near-field communication. John Wiley & Sons, 2010.

[6] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," Sensors, vol. 12, no. 9, 2012, pp. 11 734–11 753.

[7] Apple. Core nfc documentation. [retrieved: March, 2020]. [Online]. Available: https://developer.apple.com/documentation/corenfc

[8] G.-L. Frei and F. Gamper, "Design and implementation of a digital access control protocol," B.S. thesis, Bern University of Applied Science, 2019.

[9] A. Dmitrienko and C. Plappert, "Secure free-floating car sharing for offline cars," in Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. ACM, 2017, pp. 349–360.

[10] A. Dmitrienko, A.-R. Sadeghi, S. Tamrakar, and C. Wachsmann, "Smarttokens: Delegable access control with nfc-enabled smartphones," in Trust and Trustworthy Computing, S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 219–238.

[11] I. Symeonidis, A. Aly, M. A. Mustafa, B. Mennink, S. Dhooghe, and B. Preneel, "Sepcar: A secure and privacy-enhancing protocol for car access provision," in Computer Security – ESORICS 2017, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 475–493.

[12] I. Symeonidis, M. A. Mustafa, and B. Preneel, "Keyless car sharing system: A security and privacy analysis," in 2016 IEEE International Smart Cities Conference (ISC2). IEEE, 2016, pp. 1–7.

[13] L. Bauer, S. Garriss, J. M. McCune, M. K. Reiter, J. Rouse, and P. Rutenbar, "Device-enabled authorization in the grey system," in Information Security, J. Zhou, J. Lopez, R. H. Deng, and F. Bao, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 431–445.

[14] C. Arnosti, D. Gruntz, and M. Hauri, "Secure physical access with nfc-enabled smartphones," in Proceedings of the 13th International Conference on Advances in Mobile Computing and Multimedia, ser. MoMM 2015. New York, NY, USA: ACM, 2015, pp. 140–148.

[15] B. Groza, L. Popa, and P.-S. Murvay, "Carina-car sharing with identity based access control re-enforced by tpm," in International Conference on Computer Safety, Reliability, and Security. Springer, 2019, pp. 210–222.

[16] A. Ouaddah, A. Abou Elkalam, and A. Ait Ouahman, "Fairaccess: a new blockchain-based access control framework for the internet of things," Security and Communication Networks, vol. 9, no. 18, 2016, pp. 5943–5964.

[17] S. Sciancalepore, A. Capossele, G. Piro, G. Boggia, and G. Bianchi, "Key management protocol with implicit certificates for iot systems," in Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems. ACM, 2015, pp. 37–42.

[18] D. A. Ha, K. T. Nguyen, and J. K. Zao, "Efficient authentication of resource-constrained iot devices based on ecqv implicit certificates and datagram transport layer security protocol," in Proceedings of the Seventh Symposium on Information and Communication Technology. ACM, 2016, pp. 173–179.

[19] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," International journal of information security, vol. 1, no. 1, 2001, pp. 36–63.

[20] C. Research, "Standards for efficient cryptography, SEC 1: Elliptic curve cryptography," September 2000, version 1.0.

[21] E. Barker and Q. Dang, "Nist special publication 800-57 part 1, revision 4," NIST, Tech. Rep, 2016.

[22] N. Smart et al., "Algorithms, key size and protocols report (2018)," ECRYPT—CSA, H2020-ICT-2014—Project, vol. 645421, 2018.

[23] G.-L. Frei and F. Gamper. Loac-protocol prototype. [retrieved: March, 2020]. [Online]. Available: https://github.com/gianlucafrei/LOACProtocol (2019)

[24] Microsoft. X.509 public key certificates. [retrieved: March, 2020]. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/seccertenroll/about-x-509-public-key-certificates

[25] GnuPG. The gnu privacy guard. [retrieved: March, 2020]. [Online]. Available: https://www.gnupg.org/index.html

 ISBN: 978-1-61208-777-1

Keep it in Sync! Consistency Approaches for Microservices An Insurance Case Study

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
Sina Gottwald

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

***Abstract*—Microservices is an architectural style for complex application systems, promising some crucial benefits, e.g. better maintainability, flexible scalability, and fault tolerance. For this reason microservices has attracted attention in the software development departments of different industry sectors, such as e-commerce and streaming services. On the other hand, businesses have to face great challenges, which hamper the adoption of the architectural style. For instance, data are often persisted redundantly to provide fault tolerance. But the synchronization of those data for the sake of consistency is a major challenge. Our paper presents a case study from the insurance industry which focusses consistency issues when migrating a monolithic core application towards microservices. Based on the Domain Driven Design (DDD) methodology, we derive bounded contexts and a set of microservices assigned to these contexts. We discuss four different approaches to ensure consistency and propose a best practice to identify the most appropriate approach for a given scenario. Design and implementation details and compliance issues are presented as well.**

Keywords–Microservices; Consistency; Domain Driven Design (DDD); Insurance Industry.

I. INTRODUCTION

A current trend in software engineering is to divide software into lightweight, independently deployable components. Each component 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].

As a study from 2019 (see [2]) shows, the microservice architecture style is already established in many industries such as e-commerce. However, this is not the case for the insurance and financial services industry. Therefore, as part of ongoing cooperation between the *Competence Center Information Technology and Management (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. The *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. The cooperating insurance companies currently both operate a service-oriented architecture (SOA). Over time, however, it has become apparent that this architectural style is not suitable for some parts of the system and a finer subdivision (microservices) would be advantageous. A specific example for such a part of the system is the `Partner Management System`, which was transferred into a microservice architecture in the context of our research.

This paper presents a case study based on our research project. The case study focuses on consistency issues when implementing a microservice architecture. Therefore it describes how the monolithic architecture of the `Partner Management System` was divided into several microservices, which problems occur regarding the data consistency across the microservices and presents different approaches to solve these issues. Implementation details and insurance specific topics such as special compliance requirements are presented as well.

We organize the remainder of this article as follows: After discussing related work in Section II, we present the domain and requirements of the `Partner Management System` in Section III. Afterwards, Section IV shows how we split the system into microservices, discusses compliance aspects and describes the benefits the new architecture offers. Section V provides details about technical aspects of this architecture. In Section VI we evaluate the outcomes with a focus on consistency aspects. Section VII 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 VIII summarizes the results and draws a conclusion.

II. RELATED WORK

Our research is based on the literature of well-known authors in the field of microservices, especially the ground works of Fowler and Lewis [1] as well as of Wolff [3]. For the practical parts of our research, mainly the elaborations of Newman (see [4]) were used. Moreover, we found valuable ideas (also) w.r.t consistency in the patterns work from Richardson [5]. Additional helpful microservices migration patterns are, for example, presented in [6] and some as well in [7]. Especially for the migration of the legacy application,

 ISBN: 978-1-61208-777-1

the contribution of Knoche and Hasselbring (see [8]) was consulted. As a study from 2019 shows (see [2]), microservice architectures are barely found in the insurance and financial services industry in Germany. Therefore, results from other industries had to be used for our research (for example [9]).

Although the basic literature is extensive, not too 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 [10, 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., [11]) and object-oriented programming (e.g., [12]) were considered by our research. Furthermore, ideas of general database research like the SAGA-Pattern [13], which was already applied to microservices by Chris Richardson [5], were considered as well. Event-based approaches like event sourcing, described by Fowler [14], were also applied to microservices within our research.

From a microservices design perspective, domain-driven design (DDD) from Evans [15], is currently considered to be a best practice to find suitable so called bounded contexts, which can form a functional basis for microservices. Going further than many microservices-based systems, we put a special emphasis on compliance aspects, when designing the bounded contexts. Since insurance companies are highly supervised by the government (in Germany: Federal Financial Supervisory Authority (BaFin)), several compliance rules apply for them, for example “Supervisory Requirements for IT in Insurance Undertakings (in German: VAIT [16])”.

Previous work has already evaluated the outcomes of the research project mentioned in the introduction (see [17]). However, the project and the evaluation only partially discussed the issues of consistency. In particular, it missed a fair bit of implementation details. Not discussed in [17] by us at all, are the aforementioned compliance aspects for our microservice design. For this reason, we have focused on those topics after the completion of the initial project, for example, with thesis work, that dealt with the issue of consistency in context of the project results as well a another work, which dealt with compliance aspects during service design.

In total, the present article is as such a significantly extended and updated version of our previous work, especially in the details for service boundaries, service compliance aspects, system architecture, and example implementation details [18].

III. Domain and Requirements of the Partner Management System

As mentioned in the introduction, one part of our research was migrating 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 relation to the insurance company (e.g., clients, appraisers, lawyers or other insurance companies). Additional to personal information, a partner may also have information on communication, bank details, business relations and relations with other partners.

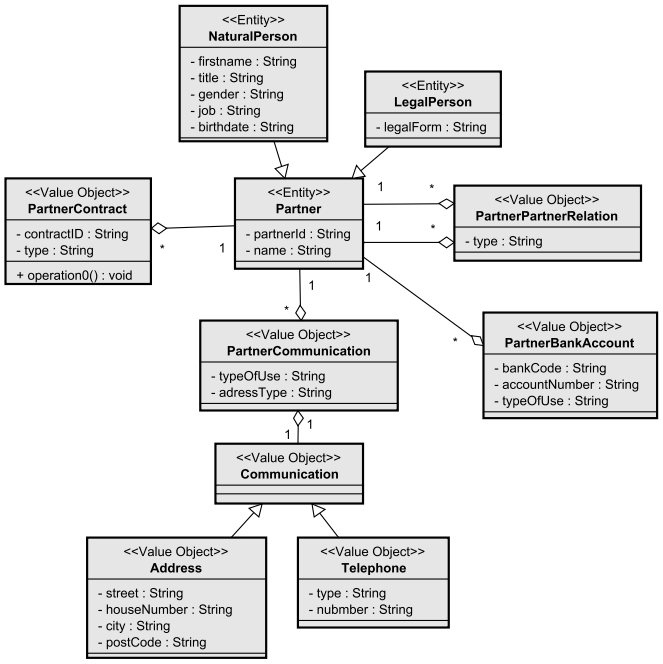

Figure 1. Simplified Model of the Overall Domain.

For introducing the overall domain, figure 1 shows a simplified model according to which the SOA service currently used by the partner companies was modelled. The presented model is strongly based on the reference architecture for German insurance companies (VAA) [19], which describes the monolithic way to implement the `Partner Management System`.

At its core, the system is a simple CRUD (Create, Read, Update, Delete) application that manages the entity `Partner` and its properties, even though the implementation as a single SOA service seems suitable at first glance.

Since the `Partner Management System` is a fundamental service of an insurance company, many other parts of the overall system are interested in the managed data. However, not all consumers of the `Partner Management System` are interested in the same subset of the data. Furthermore, some consumers should not have access to some data for security reasons. For example, the service that collects the monthly premiums does not need access to a client’s birth date or profession. These conditions result in a complex implementation of access rights and varying levels of stress for different parts of the `Partner Management System`. An efficient scaling is not possible. The `Partner Management System` is an atomic deployment unit that scales as a whole and fails as a whole.

Especially the inefficient scaling is critical in the case of partner companies, since the SOA service is implemented as a mainframe-based application that can only be scaled at great expense. Therefore, the partners use the low occupancy during the night to slowly persist all new datasets collected during 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.

 ISBN: 978-1-61208-777-1

The major issues of the current implementation of the `Partner Management System` are obviously a relatively poor flexibility, scalability and fault tolerance. In addition, the access rights management is so complex that a legally compliant implementation is difficult. This makes a microservices approach attractive for this use case.

IV. ANALYSIS OF THE MICROSERVICE ARCHITECTURE

In order to overcome the aforementioned limitations of the current `Partner Management System`, we designed a microservices-based approach. This section explains how the system has been split into independent services and presents the resulting benefits.

A. Dividing the Domain with DDD Strategic Design

In the context of our research, we used the principles of strategic design (see [15, Part IV]) as part of domain-driven design (DDD) to split the domain. Figure 2 shows the decomposition of the analysis model presented in the previous section.

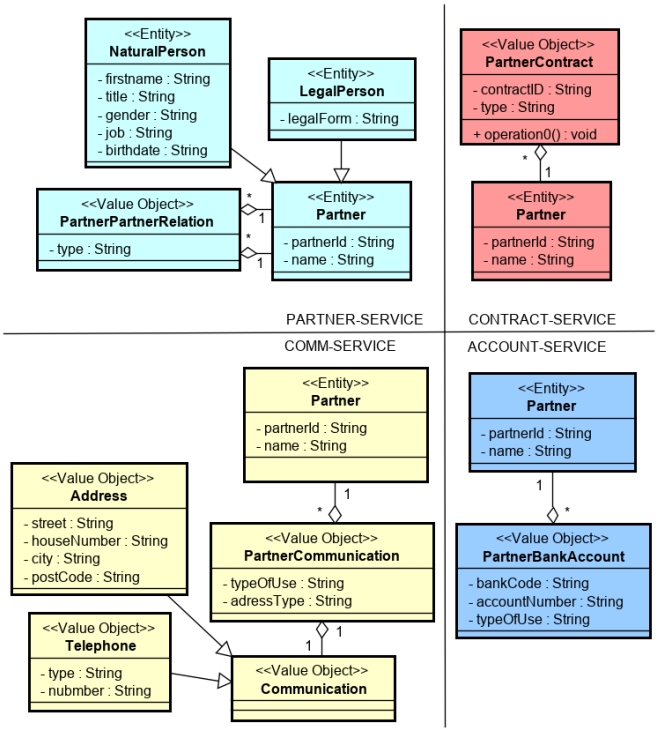

Figure 2. Bounded Contexts in the Microservice Architecture

It can be seen that the domain has been split into four bounded contexts. Bounded contexts are the central concept of strategic design. Vaughn Vernon (see [20]) describes it as an environment in which a specific language (ubiquitous language) is spoken and certain concepts are defined. Therfore, a bounded context is a conceptual boundary within which a particular domain model is applicable. In relation to Figure 2, this is shown by the fact that there are several separate models of the `Partner`, one for each bounded context. Every bounded context has a slightly different understanding of the entity `Partner`. For example, while for one context the `Partner` is a contractor, for another he is a bank account holder. In order to determine the presented contexts, we heavily discussed the domain and its processes with developers, architects and insurance domain experts. One technique that was used for this is event storming.

After we found the bounded contexts according to strategic design principles, the contexts were mapped to microservices. Leading microservices experts such as Chris Richardson [5] or Martin Fowler and James Lewis [1] recommend defining the microservices along the bounded contexts. This means that each bounded context should be implemented by one or more microservices. As figure 2 shows, in the case of the `Partner Management System`, a bijective mapping from bounded contexts to microservices was chosen. The system was divided into `partner-service`, `contract-service`, `comm-sevice` and `account-service`.

B. Compliance Aspects for Dividing the System

In addition to the better manageability of the business complexity as well as the more efficient scalability gained through the cut, the finer subdivision of the `Partner Management System` is also advantageous from a compliance point of view. This section describes the special requirements that insurance companies in Germany have to comply with and how these can be implemented by the designed microservice architecture.

TABLE I. Protection Level of Personal Data according to [21]

Protection Level	Personal data...	Example	Degree of Damage
A	...which have been made freely available by the persons concerned.	Data visible in the telephone book or on social media platforms.	minor
B	...whose improper handling is not expected to cause particular harm, but which has not been made freely accessible by the person concerned.	Restricted public files or social media not freely accessible.	minor
C	...whose improper handling could damage the person concerned in his social position or economic circumstances ("reputation").	Income, property tax, administrative offences.	manageable
D	...whose improper handling could significantly affect the social position or economic circumstances of the person concerned ("existence").	Prison sentences, criminal offences, employment evaluations, health data, seizures or social data.	substantial
E	...whose improper handling could impair the health, life or freedom of the person concerned.	Data on persons who may be victims of a criminal offence, information on witness protection program.	major

German insurance companies are supervised by the Federal Financial Supervisory Authority (BaFin) who published the "Supervisory Requirements for IT in Insurance Undertakings" [16]. These include instructions to comply with the basic principles of information security (confidentiality, integrity,

availability) since insurance companies are considered as a critical infrastructure in Germany. This means that they are essential for society and economy and therefore more worthy of protection. Since 2018 the General Data Protection Regulation (GDPR) is enforceable for processing personal data as well. This also means that companies need to ensure data protection and privacy.

The microservice architecture divides the overall system into multiple independent components. Therefore, it allows to implement different protection levels for every microservice. This perfectly fits with the well known security engineering principle of compartmentalization [22]. In case of the `Partner Management System`, it allows us to meet the legal requirements according to [21], which are shown in Table I, more easily. It can be distinguished between more sensitive data requiring a higher security level like contract or bank details and data that is more uncritical like the name and telephone number of a person.

The microservice architecture of the `Partner Management System` only processes data belonging to protection levels A to C. Data like the address, telephone number and bank details of a person aren't as critical as, e.g., health data. Since the processed data in our system would cause less damage to the person affected if protection of the data would fail, there is no need to implement a protection level as strong as it should be for more critical data. But in general, due to the division in microservices we are able to map each service to the appropriate protection level.

V. Design and Implementation of the Microservice Architecture

After the previous section presented the functional design of the microservice architecture, this section shows the technical design and some implementation insights.

A. The Technical Microservice Architecture

Taking the bounded contexts found as input, the next step is the overall technical design of the microservice architecture.

Based on the technical specifications of the insurance companies involved, the resulting microservices are designed to be implemented as REST web services (see figure 3) in Java using the Spring framework. As mentioned before, each microservice should have its own data management, realized here as dedicated PostgreSQL databases. Instances of a microservice share a database (cluster). `partnerId` and `name` are kept in sync across all microservices using REST calls of the `partner-service`.

Parts of the Netflix OSS stack are used for the system infrastructure: Netflix Eureka as a service discovery and Netflix Zuul as an API gateway. Zuul also provides the web frontend of the application, which is realized as a single-page application using AngularJS. The ELK stack (Elasticsearch, Logstash and Kibana) is 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 microservices, it is possible to run any number of instances of each microservice.

B. Integrating Services into the Microservice Architecture

Given the technical architecture of the overall microservices-based system, we now provide deeper technical details of particular microservices. Therefore, this section provides code snippets to show how services are integrated into the microservice architecture and how fault-tolerant calls between services are implemented.

```
1 @EnableEurekaClient
2 @SpringBootApplication
3 public class Application {
4  public static void main(String[] args) {
5   SpringApplication
6    .run(Application.class, args);
7  }
8 }
```

Listing 1. Registration with Eureka Service Registry

To integrate a microservice into the microservice architecture from figure 3, it only needs to be registered with Eureka under a specific name. As a result, the API gateway (and any other microservice) can find instances of the microservice at runtime and use the provided REST endpoints. Fortunately, as listing 1 shows, Spring (Cloud Netflix) provides an easy way to register a microservice with Eureka.

As the listing shows, the class with the main method needs to be annotated with `@EnableEurekaClient`. In addition (not shown in the listing), the address of Eureka and the name under which the microservice should be registered must be stored in the `application.properties` file. As a result, the instance of the microservice now registers with Eureka when starting up. Note: In practice, there will usually be a federation of Eureka instances to ensure their availability.

C. Implementing Network Calls

Since network calls (to other microservices) can fail, it is useful to implement a fault tolerance mechanism. In our case,

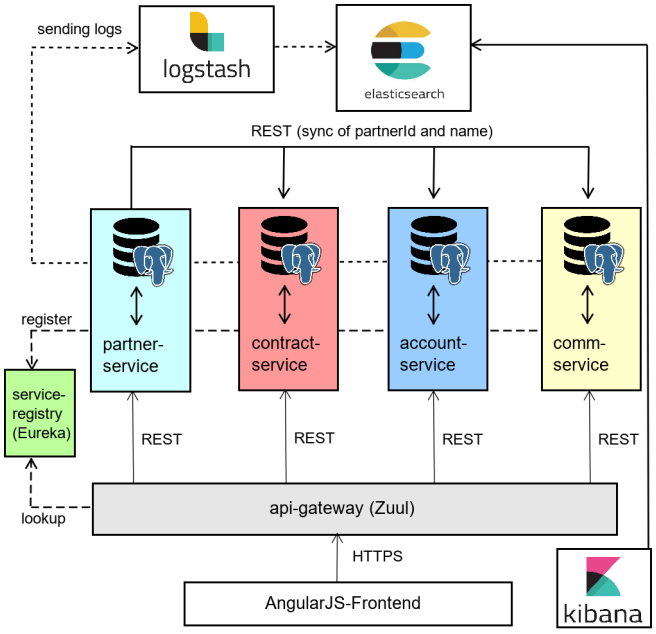

Figure 3. Technical Design of the Microservice Architecture

we used the Retry library from Spring. Just like the Eureka integration it is available via a simple annotation.

```
NaturalPerson createNP(NaturalPerson p){
 NaturalPerson savedInDB = repository.save(p);
 distribNP(p);
 log.info("A natural person was created.");
 return saved;
}

@Retryable(value={Exception.class}, maxAttempts=5)
void distribNP(NaturalPerson p) throws Exception{
 //network call
 dataDistributionService.update(savedInDB);
}

@Recover
void recover(Exception e, NaturalPerson p){
 //Rollback transaction to ensure
 //consistency and notify somebody
 //that something went terribly wrong.
}
```

Listing 2. Network Call with Retries

Listing 2 shows simplified parts of the implementation of the `partner-service`. Every time a new person is created, the change must be propagated to the other services via a synchronous network call. Since the successful execution of this call is a critical factor for consistency in the system, it makes sense to secure this call with a fault tolerance mechanism. The `@Retryable` annotation ensures that the annotated method is called several times if an exception occurs during execution. Listing 2 shows an implementation of five attempts to send the data to the other microservices. If an exception also occurs after the fifth time, the method annotated with `@Recover` is called instead.

D. Design of the microservices

This section presents the internal architecture of the microservices. The services are implemented in Java using the Spring Framework. Parts of the `partner-service` serve as an example to show the general architecture of the services.

Figure 4 shows the architecture of the `partner-service`. It follows a layered architectural style. Each layer will be explained in the following.

E. Routing Layer

The routing layer is the one that interacts with the consumers of the service. With the annotation `@RestController` on the `NaturalPersonController`, the Spring Framework creates an instance of the class and delegates incoming HTTP requests to its methods. Each of the methods in the `NaturalPersonController` is responsible for a REST endpoint. The functional scope of the controller is limited to the CRUD operations and an interface to search for natural persons based on certain properties.

In addition to the `NaturalPersonController`, there is also a `LegalPersonController` and a `PartnerRelationsController` to provide the remaining functionality of the `partner-service`. Since the routing layer responds to incoming HTTP requests, it is responsible for encoding and decoding objects. In our

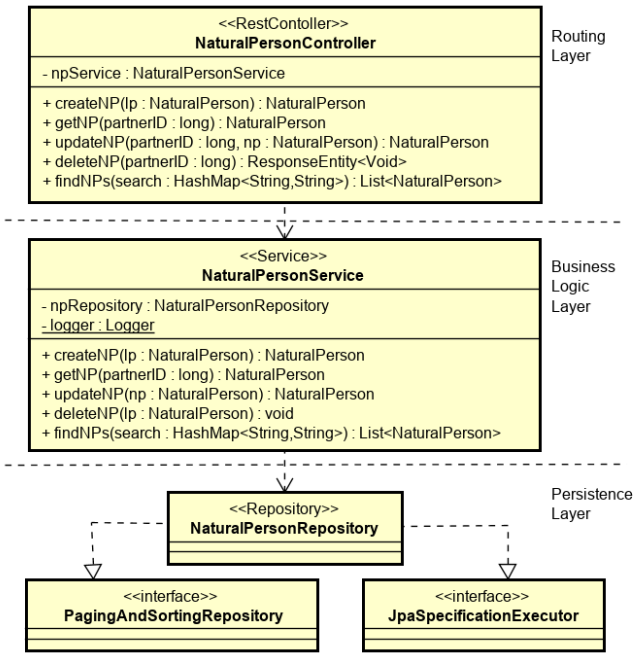

Figure 4. Architecture of the Partner-service

implementation, the objects are serialized as JSON. The layer is also responsible for catching exceptions from the business logic layer and translating them into HTTP status codes. In summary, the routing layer is an HTTP facade for the business logic.

F. Business Logic Layer

Classes in the business logic layer are provided with the annotation `@Service`. As a result, they are managed by the Spring Framework and can be used in other contexts using Spring's dependency injection mechanism. For example, the `NaturalPersonController` gets an instance of the `NaturalPersonService` injected to communicate with the business logic layer. As the name suggests, the business logic layer implements the business logic. In our implementation, the business logic is limited to some input validation and the communication with the persistence layer. Another responsibility of the layer is to log domain events. In our case, this is done with the SLF4J logging framework provided by Spring. In addition to the `NaturalPersonService`, there is also a `LegalPersonService` and a `PartnerRelationsService` to provide the remaining functionality of the `partner-service`.

G. Persistence Layer

The responsibility of the persistence layer is to commit run-time-objects to the database and convert persistent data to run-time-objects. Again, the dependency injection mechanism of Spring is used to give the layer above access to the functionality. The persistence layer is implemented using Spring Data JPA and is therefore limited to the definition of an interface with the annotation `@Repository`. Managed entity classes must be annotated with `@Entity`. For the search functionality described above and the possibility

to sort search results, the repository is extended with another two interfaces from Spring Data JPA. For clarity, the methods of the persistence layer are not shown in figure 4. The exact description of Spring Data JPA can be found in the documentation of the Spring Framework. In addition to the `NaturalPersonRepository`, there is also a `LegalPersonRepository` and a `PartnerRelationsRepository`. Each repository is responsible for a specific database table.

VI. Challenges of the microservice architecture

Looking at the architecture described in section IV, it looks like the microservice architecture can solve the problems of the currently implemented monolithic system. In particular, the scalability and fault tolerance of individual parts of the system are a crucial advantage compared to the current implementation. 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 is the consistency assurance across the services. This means the synchronization of the `partnerId` and `name`, 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 `partnerId` or `name` 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 not corrupting the other contexts by changing the data model of `Partner`. This approach is known as Customer/Supplier pattern (described by Evans [15]) in the context of DDD. If it is likely that the data model changes, an anticorruption layer should be considered.

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. 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].

The synchronization of `partnerId` and `name` is a critical part of the application, which is why its implementation needs to 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 is not available for the `partner-service`, it will not be notified as well. This is due to the transient characteristic of REST calls.
- **Synchronicity.** After a change of `partnerId` or `name`, 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 response time 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.** Extensibility is a key benefit of the microservices approach. In the current implementation, the `partner-service` holds a static list of services that need to be notified upon a change of `partnerId` or `name`. If a new service is added to the system, which is interested in partner data, the `partner-service` must be redeployed. Additionally, the bigger the number of services to notify gets, the more the response time of the `partner-service` is impaired.

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

VII. Consistency Assurance in the Partner Management System

In order to find a suitable solution for the specific problem of synchronizing `partnerId` and `name`, the general approaches have to be examined.

A. General approaches

The central research question is how a change of master data can be propagated 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 schemas 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 VI, our microservice architecture already follows this approach. To provide loose coupling however, the addresses of services to be notified 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 the information which service is interested in which data. This approach roughly corresponds to the Observer-Pattern of object-oriented software development, where the subject registers at the observer to get synchronously notified when changes are made.
  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 as long as the requests don't fail.
- **Polling.** Another solution is to relocate the responsibility of synchronizing the redundant data to the interested services themselves. A straightforward approach

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 [23]).

- **Publish-Subscribe.** To completely decouple the service containing the master data from the other services, a message queuing approach can be utilized. 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 service failures. This approach is suitable for 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. For example, the SAGA-Pattern uses this approach to distribute information about changes.
- **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 persisted 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 might be an indicator that the determined bounded contexts are not optimal. In some cases, contexts are coupled so tightly that keeping data redundantly is not feasible. In this case, it should be discussed if the contexts and therefore also services can be merged. Furthermore, if this issue occurs in several parts of the architecture, it should be evaluated whether a microservices approach is the right choice for this domain.

B. Best practice for synchronizing partner data

The discussion in the previous section has shown that some approaches tend to guarantee a stronger level of consistency than 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

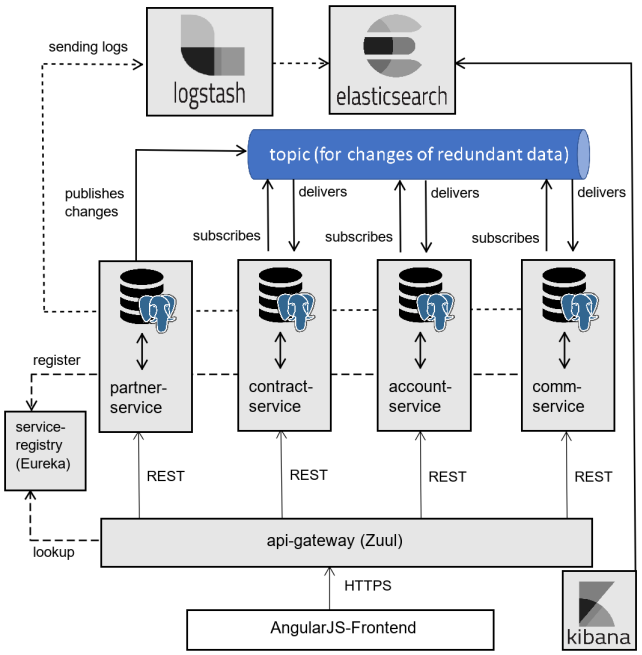

Figure 5. `Partner Management System`: Publish Subscribe Solution

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 partner data, containing the `partnerId` and `name` 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.
- `partnerId` or `name` might not be up-to-date.
- A deleted `Partner` might not yet be deleted everywhere.

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 his or her 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.

 ISBN: 978-1-61208-777-1

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 is acceptable – 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 loose coupling, high scalability and easy monitoring. Especially because of loose coupling, the Publish-Subscribe-Pattern is the most viable solution.

Figure 5 shows how the Publish-Subscribe solution can be used to synchronize `partnerId` and `name`. On every change of the partner data, an event is broadcasted by the `partner-service`. The other services subscribe to this topic and receive those events and update their own data accordingly. Technically, the topic could be implemented by, for example, RabbitMQ and a standardized messaging protocol like AMQP.

VIII. Conclusion and Future Work

This paper has presented a microservice case study from the insurance industry. The challenges of the existing `Partner Management System` were identified and discussed. The paper gave insights into the design process of the new microservice architecture, presented implementation details and discussed further (insurance-specific) topics such as special compliance requirements. In addition, the new challenges arising from the microservice architecture were discussed and the implementation, which was initially developed during our research project, was critically reviewed with regard to consistency. The general solutions for the synchronization of data in distributed systems were pointed out and a (more suitable) alternative to the current implementation was determined.

The next steps will be to complete the implementation of the publish-subscribe approach for the `Partner Management System` with the described technologies. Prior to take the system into operation, it must be comprehensively tested under real-world conditions. In our recent research a test strategy tailored to the requirements of the industry partner was designed and implemented. This strategy comprises several steps, i.e. interface tests, web interface tests, load and performance tests, which are performed within so-called 'layers'. These layers represent different test environments, up to an infrastructure which is similar to the production environment. To exploit the potential of the strategy it is reasonable to integrate the tests into a CI/CD (continuous integration / continuous delivery) pipeline as part of the pipeline's test stage. After completing the implementation of the `Partner Management System`, the tests of the system will be performed based on the test strategy.

To demonstrate the approach 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: 05, 2020].

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

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

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

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

[6] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns," Software: Practice and Experience, vol. 48, no. 11, 2018, pp. 2019–2042.

[7] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Software, vol. 33, no. 3, 2016, pp. 42–52.

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

[9] 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.

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

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

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

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

[14] M. Fowler, "Event Sourcing," https://martinfowler.com/eaaDev/EventSourcing.html, December 2005, [retrieved: 05, 2020].

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

[16] "Versicherungsaufsichtliche Anforderungen an die IT (VAIT) (2019) vom 20.03.2019," https://www.bafin.de/SharedDocs/Downloads/DE/Rundschreiben/dl_rs_1810_vait_va.html, March 2019, [retrieved: 05, 2020].

[17] 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://www.conf-micro.services/2019/papers/Microservices_2019_paper_8.pdf, [retrieved: 05, 2020].

[18] A. Koschel, A. Hausotter, M. Lange, and P. Howeihe, "Consistency for Microservices - A Legacy Insurance Core Application Migration Example," in SERVICE COMPUTATION 2019, The Eleventh International Conference on Advanced Service Computing, Venice, Italy, 2019, https://www.thinkmind.org/index.php?view=article&articleid=service_computation_2019_1_10_18001, [retrieved: 05, 2020].

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

[20] V. Vernon, Implementing domain-driven design. Addison-Wesley, 2013.

[21] "Schutzstufenkonzept des LfD Niedersachsen," https://www.lfd.niedersachsen.de/technik_und_organisation/schutzstufen/schutzstufen-56140.html, October 2018, [retrieved: 05, 2020].

[22] A. Roland, "Secrecy, technology, and war: Greek fire and the defense of byzantium, 678-1204," Technology and Culture, vol. 33, no. 4, 1992, pp. 655–679.

[23] 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, [retrieved: 05, 2020]

 ISBN: 978-1-61208-777-1

Towards a Tool-based Approach for Microservice Antipatterns Identification

Rafik Tighilt
Université du Québéc à Montréal
Montréal, Québec, Canada
Email: tighilt.rafik@gmail.com

Manel Abdellatif
Polytechnique Montréal
Montréal, Québec, Canada
Email: manel.abdellatif@polymtl.ca

Naouel Moha
Université du Québéc à Montréal
Montréal, Québec, Canada
Email: moha.naouel@uqam.ca

Yann-Gaël Guéhéneuc
Concordia University
Montréal, Québec, Canada
Email: yann-gael.gueheneuc@concordia.ca

***Abstract*—Microservice architecture has become popular in the last few years because it allows the development of independent, reusable, and fine-grained services. However, a lack of understanding of its core concepts and the absence of reference or consensual definitions of its related concepts may lead to poorly designed solutions called antipatterns. The presence of microservice antipatterns may hinder the future maintenance and evolution of microservice-based systems. Assessing the quality of design of such systems through the detection of microservice antipatterns may ease their maintenance and evolution. Several research works studied patterns and antipatterns in the context of microservice-based systems. However, the automatic identification of these patterns and antipatterns is still at its infancy. We searched for re-engineering tools used to identify antipatterns in microservice-based systems in both academia and industry. The results of our search showed that there is no fully-automated identification approach in the literature. In this paper, we aim to reduce this gap by (1) introducing generic, comprehensive, and consensual definitions of antipatterns in microservice-based systems, and (2) presenting our approach to automatically identify these antipatterns. Currently, this work is still in progress and this paper aims to present the approach and the metamodel used for future implementation.**

Keywords–Microservices; Antipatterns; Identification.

I. INTRODUCTION

A microservice is defined as a small service, with a single responsibility, running on its own process, and communicating through lightweight mechanisms [1], such as representational state transfer application programming interfaces (REST APIs) and message brokers. Each microservice in a microservice-based system fulfills a single business function, manages its own data, runs on its own process, is managed by a single team, and is not tied to the system itself for its evolution or deployment. Microservices are built around business requirements, deployed by a fully automated deployment machinery with a minimum centralized management [2] and are loosely coupled.

Several major actors of the software industry have adopted microservice-based systems, such as Netflix and Amazon. The popularity of this architecture still grows, mainly due to its dynamic and distributed nature, which offers greater agility and operational efficiency and reduces the complexity of handling applications scalability and deployment cycles wrt. monolithic applications [2]. Software maintenance is one of the most important fields in the software industry, whether in expenses or in resources [3].

However, like any other architectural style, microservice-based systems also face challenges with maintainability and evolution due to "poor" solutions to recurring design and implementation problems, called antipatterns [4]. These antipatterns can degrade the overall quality of design and quality of service of the microservices themselves and the system as a whole [5].

The nature of microservice systems makes them very dynamic (multi-language, multi-operating environments, etc.) [2]. This makes the identification of antipatterns difficult, especially because there is a lack of automated approaches in the literature to help fulfil this task.

We contribute to the maintenance and evolution of microservice-based systems with generic, comprehensive, and consensual definitions of antipatterns in microservice-based systems and an automatic tool-based approach for the identification of antipatterns in these systems. Our automatic tool-based approach relies on a meta-model we established and described in this paper. The meta-model covers the needed information to apply our heuristics and identification rules yet can be extended for future work.

However, we must overcome some challenges introduced by microservice-based systems.

1) **Microservices are, by definition, independent [2].** Microservice-based systems are deployed on multiple providers using different tools and configurations.
2) **Microservices can be built using different programming languages [1]**. This makes the identification process more challenging compared to single-language systems.

Thus, for the first step of our work and to validate our approach, we only consider systems built with the Java programming language and using Docker as container technology as they are among the most popular tools to build microservice-based systems.

The remainder of this paper is structured as follows. Section II describes previous work related to microservices antipatterns cataloguing and identification. Section III outlines our methodology for antipatterns identification. It also introduces our catalogue of microservice antipatterns and the metrics and hints to identify these antipatterns. Section IV presents some limitations that we identified in our approach. Finally, Section V concludes this paper and presents the future work.

 ISBN: 978-1-61208-777-1

II. Related Work

In their study, Pahl and Jamdi [6] aim to identify, taxonomically classify and systematically compare the existing research body on microservices and their application in the cloud. They conducted a systematic mapping study of 21 works on microservice design published between 2014 and 2016. They defined a characterization framework and used it to study and classify the works. Their study reports a lack of research tools supporting microservice-based systems and conclude that microservice research is still in a formative stage. The study results in a discussion of the microservice architectural style concerns, positioning it within a continuous development context and moving it closer to cloud and container technology.

In his overview and vision paper, Zimmerman [7] reviews popular introductions to microservices to identify microservices tenets. It then compares two microservices definitions and contrasts them with SOA principles and patterns. this paper compiles practitioner questions and derives research topics from the differences between SOA and microservices architectural style. The author conclude his paper with the opinion that microservices are one special implementation of the SOA paradigm.

Garriga [8] defines a preliminary analysis framework in the form of a taxonomy of concepts including the whole microservices lifecycle, as well as organizational aspects. The author claims that this framework is necessary to enable effective exploration, understanding, assessing, comparing, and selecting microservice-based models, languages, techniques, platforms, and tools. He then analyzed state of the art approaches related to microservices using this taxonomy to provide a holistic perspective of available solutions. Additionnaly, the paper identified open challenges for future research from the results of litterature analysis.

Soldani et al. [9] identified and compared benefits and limitations of microservices by studying the industrial *grey* literature. They also studied the design and development practices of microservices to bridge academia and industry in terms of research focus. Marquez and Astudillo [10] provided (1) a catalog of microservice architectural patterns from academia and industry, (2) a correlation between quality attributes and these patterns, (3) a list of technologies used to build microservice-based systems with these patterns and (4) a comparative analysis of SOA and microservice architectural patterns. They did that to determine whether architectural patterns are used in the development of microservice-based systems. This work extended their previous work with Osses [11].

Taibi et al. [12] introduced a catalog and taxonomy of the most common microservices anti-patterns to identify common problems resulting from the migration of monolithic applications to microservice-based systems. Their catalog is based on the experience of 27 interviewed practitioners. The authors identified a taxonomny of 20 anti-patterns including organizational and technical anti-patterns and estimated their level of harmfulness through a survey. They conclude that splitting a monolith is the most critical issue.

Borges and Khan [13] selected 5 well known anti-patterns in microservice-based systems and proposed an algorithm to automatically detect them. The authors claim that their solution can avoid common mistakes when deploying microservices-based projects and can help project managers to get an overview of the system as a whole. They tested their algorithm on a well known open source microservice-based project and revealed possible improvements.

Microservice antipatterns have been discussed in the literature, but very little work has been done in the field of their automatic identification. To the best of our knowledge, only Borges and Khan [13] proposed an algorithm to automatically identify antipatterns in microservice-based systems. However, they only identify 5 antipatterns. Our approach do not focus on the same antipatterns eventhough we may share some.

III. Study Design

This section presents the design of our study. First, we explain the approach we used to construct our research. Then, we detail the metamodel used to automatically identify antipatterns in microservice-based systems. Finally, we list the detection rules for each antipattern.

A. Approach

This section presents our approach to fulfill our objectives. First, we reviewed the literature and studied 67 open-source projects to build a catalogue of microservice antipatterns. Second, we study each antipattern to provide a concise description and extract hints of its presence in microservice-based systems using source-code, configuration files, deployment files and git repositories. The catalogue and the description of the microservice antipatterns have been presented in our previous work [14]. Finally, we build an automated tool-based approach for the identification of microservice antipatterns.

1) Step 1: Catalogue of Microservice Antipatterns:

a) Literature review: To build our catalog, we reviewed the literature following the procedures proposed by Kitchenham et al. [15] for performing systematic literature reviews. We excluded papers not written in English and papers not related to microservices antipatterns. We obtained a total of 27 papers describing microservice antipatterns.

We grouped antipatterns having similar definitions under a single name and excluded antipatterns that are only related to the organizational structure of the company or too specific and that cannot be generalized (e.g., the Frankenstein antipattern that is related to switching from waterfall to agile development).

b) Open-source systems review: After reviewing the literature, we manually analyzed 67 open source systems [16] to assess the concrete presence of the identified antipatterns in these microservice-based systems. Table I shows examples of identified antipatterns inside microservice-based systems.

After reviewing the litterature and the open source microservice-based systems, we obtained a total of 16 antipatterns described below.

1) **Wrong Cuts:** This antipattern consists of microservices organized around technical layers (Business layer, Presentation layer, Data layer) instead of functional capabilities, which causes strong coupling of the microservices and impedes the delivery of new business functions.
2) **Cyclic Dependencies:** This antipattern occurs when multiple services are co-dependent circularly and, thus, no longer independent, which goes against the very definition of microservices.

 ISBN: 978-1-61208-777-1

TABLE I. EXAMPLES OF IDENTIFIED ANTIPATTERNS IN MICROSERVICE-BASED SYSTEMS

System name	Identified antipatterns
ACME Air	Manual Configuration, Shared Persistence, Hardcoded Endpoints, No Healthcheck
Cinema microservice	Manual Configuration, Hardcoded Endpoints, No Healthcheck
Delivery system	Hardcoded endpoints, Local logging, Insufficient monitoring, No Healthcheck
E-commerce microservices sample	Manual Configuration, Hardcoded Endpoints, No API gateway, Local Logging
Microservices demo	Hardcoded Endpoints, No API Gateway, No API versioning
Beer Catalog	Hardcoded Endpoints, Shared Libraries, Multiple Service Instances Per Host
Springboot microservices example	Manual Configuration, Hardcoded Endpoints, No Healthcheck

3) **Mega Service:** This antipattern appears when a microservice serves multiple business functions. A microservice should be manageable by a single team and bounded to a single business function.
4) **Nano Service:** This antipattern results from a too fine-grained decomposition of a system in which multiple microservices work together to fulfill a single business function.
5) **Shared Libraries:** This antipattern consists of libraries and files (ex. binaries) used by multiple microservices, which breaks the microservices independence as they rely on a single source to fulfill their business function.
6) **Hardcoded endpoints:** This antipattern relates to URLs, IP addresses, ports and other endpoints being hardcoded in the microservice source code including configuration files. This may interfere with the load balancing and the deployment of the microservices.
7) **Manual Configuration:** This antipattern happens with configurations that must be manually pushed to each microservice of a system. Microservice systems evolve rapidly and their management should be automated, including their configuration.
8) **No Continuous Integration (CI) / Continuous Delivery (CD):** Continuous integration and delivery are important for microservices to automate repetitive steps during testing and deployment. Not using CI/CD undermines microservices, which encourages automation wherever possible.
9) **No API Gateway:** This antipattern occurs when consumer applications (front ends, mobile applications, etc.) communicate directly with microservices. Each application must know how the whole system is decomposed and must then manage endpoints and URLs for each microservice.
10) **Timeouts:** This antipattern happens when timeout values are set and hardcoded in HTTP requests, which leads to spurious timeouts or unnecessary delays.
11) **Multiple Service Instances Per Host:** This antipattern happens when multiple microservices are deployed on a single host, which prevents their independent scaling and may cause technological conflicts inside the host.
12) **Shared Persistence:** This antipattern happens when multiple microservices share a single database: they no longer own their data and cannot use the most suitable database technology for it.
13) **No API Versioning:** This antipattern happens when no information is available about a microservice version, which can break changes and force backward compatibility when deploying updates.
14) **No Health Check:** This antipattern occurs when microservices are not periodically health checked. Unavailable microservices may not be noticed and cause timeouts and errors.
15) **Local Logging:** This antipattern occurs when microservices have their own logging mechanism, which prevents the aggregation and analyses of their logs and may slow down the monitoring and recovery of a system.
16) **Insufficient Monitoring:** This antipattern relates to microservice systems performances/failures, which are not tracked and cannot help maintain the functions of the systems.

2) Step 2: Detection of the Microservice Antipatterns: We present an approach to detect the antipatterns catalogued in Section III-A1. Figure 1 shows that our approach takes as input a microservice-based project or a list of microservices (both either as Git repositories or local source code folders). Then, from each microservice, it extracts the relevant files by excluding binaries (e.g., .jar, .exe, .bin files) and vendor files (e.g., node_modules, composer vendor etc.). Then, our approach splits the extracted files into four categories based on their extension, content, and programming language:

1) **Code:** These are source-code files of the microservice. We split these files into programming files (Java, PHP, Go, etc.), configuration files (XML, JSON, YAML, etc.), markup files (HTML, CSS, EJS, etc.), and data files (CSV, GitAttributes, Properties, etc.).
2) **Environment:** If available, these files store environment variables for the microservices. Usually in key value pairs.
3) **Deployment:** These are deployment scripts for the microservices (Dockerfiles, docker-compose, etc.). We do not consider configuration files in this category. We only save files directly related to deployment (Docker files, Docker-compose, etc.).
4) **Configuration:** These are configuration files for the microservice (JSON files, XML files, etc.). Source code files that only contain configuration are also added to this category. That means that a source code file "xxx-config.java" will be considered in both the source code category and the configuration category.

From each category, we can extract some information to build

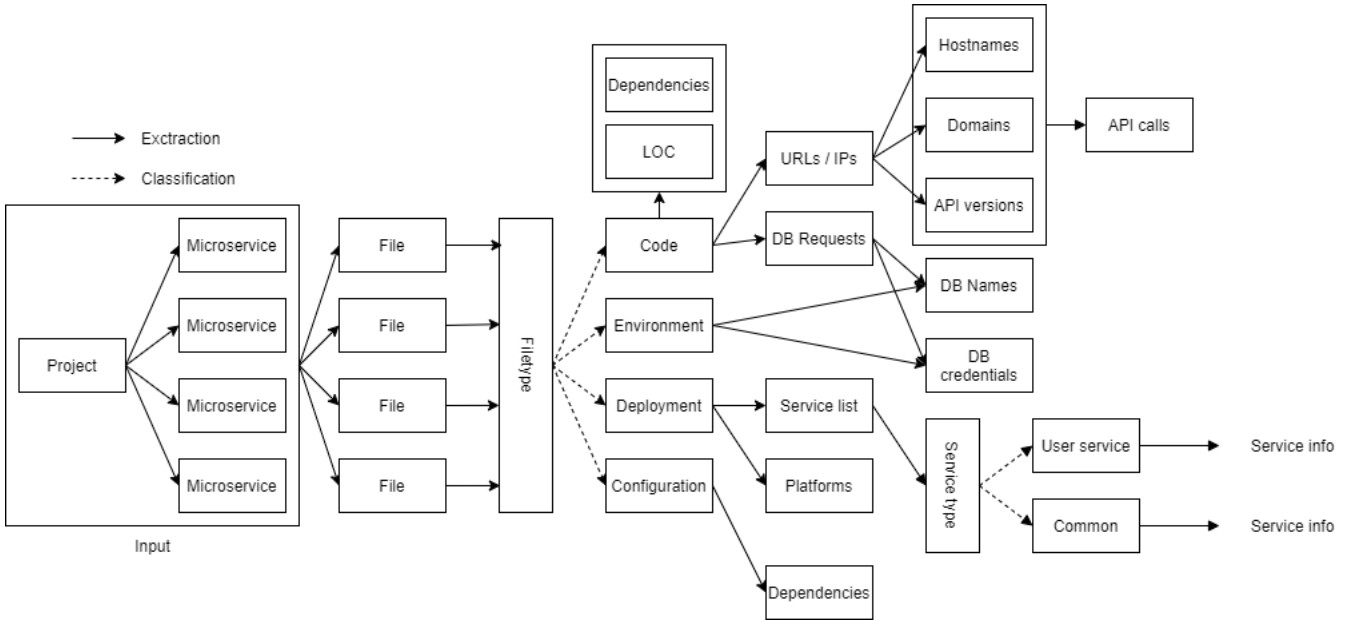

Figure 1. Antipatterns Identification Process Architecture

our model of the microservices, which contains information needed to apply our detection heuristics to identify antipatterns.

B. Metamodel Definition

This section describes the meta-model that we use to encapsulate the needed information to identify antipatterns. The metamodel is divided in 13 components, each one containing some information related to the systems and the microservices. The components are as follow:

1) System: This component holds information about the system itself.

- **isGitRepository**: True if the provided system to analyze is a git repository.
- **importedAt**: The timestamp when the analysis was performed.

2) GitRepository: If the provided system is in a Git repository, this component stores information about it.

- **url**: Repository URL.
- **owner**: The owner of the Git repository.
- **nbContributors**: The number of different developers that contributed to the repository.
- **nbCommits**: The number of commits to the repository.
- **branch**: Current branch of the repository.
- **buildStatus**: If available, the build status of the repository.

3) Microservice: This component stores information about a single microservice.

- **languages**: List of programming languages used in a microservice.
- **loc**: The number of lines of code for a microservice.

4) Dependency: This component holds information about a single dependency.

- **name**: Dependency name.
- **source**: The source from where the dependency is installed.
- **category**: The dependency category (e.g., ORM, Logging, Monitoring, etc.).
- **type**: The type of the dependency (binary, framework, library, etc.).

5) Deploy: This component contains generic deployment information.

- **area**: If available, the area of the deployment (development, staging, production, etc.).
- **instructions**: List of deployment instructions (e.g., Dockerfile commands).

6) Config: This component contains configuration information.

- **type**: The configuration file type.
- **path**: The path of the configuration file.
- **values**: Actual configuration key/value pairs.

7) Env: If available, this component contains environment variables information.

- **type**: The environment file type.
- **values**: Actual environment variables key/value pairs.

8) Code: This component holds information about a given source code file.

- **languages**: List of programming languages of the current file.
- **mainLanguage**: Main programming language used in the file.
- **loc**: Lines of code of the file.

 ISBN: 978-1-61208-777-1

9) Image: If available, this component holds information about container images of the system.

- **name**: Image name.
- **type**: If available, the image type (e.g., database system, monitoring,etc.).

10)Server: This component is related to deployment server information.

- **address**: Server address.
- **port**: Deployment port number.

11)HTTP: This component stores information about HTTP requests.

- **sourceFile**: File from where the HTTP request was performed.
- **endpointURL**: HTTP requests destination endpoint.
- **port**: HTTP requests port.
- **type**: The type of the HTTP request.
- **parameters**: HTTP requests parameters.

12) Database: This component stores information about database queries.

- **dbQuery**: The database query string.
- **queryType**: The type of the database query.
- **dbName**: Database name.
- **dbLocation**: Database location address.
- **dbUsername**: If available, the database user name.
- **dbPassword**: If available, the database user password.

13) Import: This component stores information about imported packages in the source code.

- **package**: The imported package.
- **path**: The imported path.
- **fileType**: Imported file type.

Figure 2 illustrates the relations between each component of our metamodel.

C. Detection rules

We now describe the detection rules of our approach for each antipattern.

1) Wrong Cuts: Microservices have one file type in the source code and connect to multiple microservices having also one file type. An example would be a microservice containing only presentation related code connecting to a microservice containing only business logic code. We rely on the files extensions, contents, and programming languages to identify this antipattern.

2) Cyclic dependencies: Microservices performing API call to other microservices circularly. We detect this antipattern using the API calls, endpoints, and dependencies extracted in our model.

3) Mega Service: Such a microservice has more lines of code, connects to multiple databases, has a high fan in and fan out, and has a lot of dependencies compared to other microservices.

4) Nano Service: Such a microservice has less lines of code, connects to zero or one database, has a low fan in and fan out, and has no or a few dependencies compared to other microservices.

5) Shared Libraries: Multiple microservice source files, dependency binaries, and libraries are shared between multiple microservices.

6) Hardcoded Endpoints: REST API calls inside microservices source code, deployment files, configuration files, or environment files contain hard-coded IP addresses, port numbers, and URLs. There is no service discovery present in the system.

7) Manual Configuration: Microservices have their own configuration files. No configuration management tools are present in the dependencies of the system and no microservice is responsible of configuration management.

8) No CI/CD: Configuration files and version control repositories do not contain continuous integration/delivery-related information. We rely on an extensible list of CI/CD tools to perform our analysis.

9) No API Gateway: Microservice source code does not contain signatures of common API gateway implementations (e.g., Netflix Zuul). No frameworks or related tools are present in the dependencies of the microservice. API calls are direct calls to microservices.

10) Timeouts: Timeout values are present in REST API calls. No signatures of common circuit breaker implementations (e.g., Hystrix) are present in the source code. No circuit breaker is present in the dependencies of the microservice.

11) Multiple Service Instances Per Host: We analyze and compare deployment scripts of all microservices to find the ones that share the same hosts.

12) Shared persistence: We extract the databases used by the microservices and then assess if any database is used by more than one microservice.

13)No API Versioning: Endpoints and URLs do not contain version numbers. No version information present in the headers when performing HTTP requests.

14) No health check: No "healthcheck" or "health" endpoint in microservices. No common implementation of health checks present in the source code (e.g., Springboot actuator).

15) Local Logging: No distributed logging present in the dependencies. No common logging microservice. Each microservice has its own log file paths.

16) Insufficient Monitoring: No monitoring framework or library in the microservices dependencies (e.g., Prometheus).

IV. Approach Limitations

In this section, we discuss the limitations of our approach and the measures that we took to reduce them.

A. Internal limitations

Although we intensively reviewed the literature to find the most common antipatterns in microservice-based systems, they are potentially other antipatterns that we did not include in our study. Yet, with the antipatterns described in our catalogue, we aim to establish a foundation for future work. Other researchers should perform similar reviews to confirm/infirm ours.

The detection rules we established to identify antipatterns are subject to our interpretation of antipatterns. Mega service for example is subjective, and can be discussed. However, we tried to minimize this limitation by considering every microservice as a part of the system instead of a stand-alone application. This way, we can say that a Mega service is *relative* to the system.

 ISBN: 978-1-61208-777-1

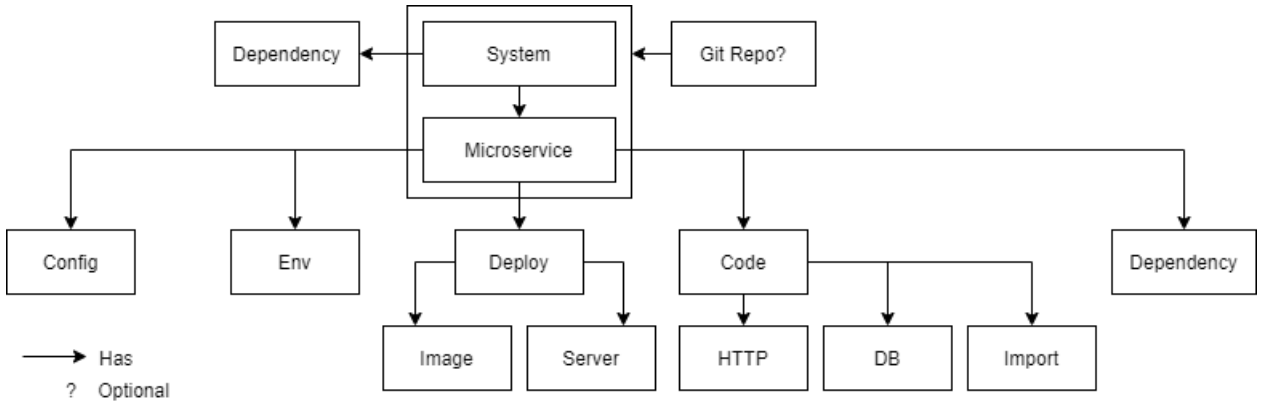

Figure 2. Relations between all metamodel components

B. External limitations

Microservice-based systems are volatile. They can be built using multiple technologies and deployed to multiple providers. Even though we tried to identify the most common technologies in the field of microservices, we might have omitted some. We will minimize this limitation by building a tool that can be extended by providing more parsers and deployment environments.

We rely on lists of dependencies and frameworks to identify some antipatterns. We pre-define these lists by taking the most widely used technologies in that area, but we do not pretend to have exhaustive lists. However, we will build the system in a way these lists can be extended easily to cover more tools and frameworks.

Even though configuration files are widely written in JSON, XML, or YAML file formats, they can also be written in the programming language itself. This may lead us to misconsider a file and not include it in the configuration category. We reduce this limitation by not only relying on the file extension, but also on the file name and its content to do the classification.

V. Conclusion and Future Work

We describe in this paper our automated approach for the identification of antipatterns in microservice-based systems. We provide a list of previously identified antipatterns from the literature. We detail the meta-model we use in our approach and we finally define the heuristics and detection rules for each of the identified antipatterns.

We believe that our approach is robust enough to identify the described antipatterns yet still extensible and flexible to evolve with the evolution of programming languages and antipatterns themselves.

Future work includes first implementing our detection rules to identify antipatterns in Java microservices to validate and refine our approach. We will validate our approach by manually analysing the microservice-based systems and calculate precision and recall for each of the identified antipatterns. Then, we want to extend our approach to consider multi-language microservice-based systems. Finally, we aim to empirically study the effect of these antipatterns on the quality of systems.

References

[1] "Microservices: a definition of this new architectural term," 2019, URL: https://martinfowler.com/articles/microservices.html [retrieved: August, 2020].

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., Feb. 2015, ISBN: 978-1491950357.

[3] M. Hanna, "Maintenance burden begging for remedy," *Software Magazine*, vol. 13, pp. 53–53, Apr. 1993.

[4] D. Taibi and V. Lenarduzzi, "On the Definition of Microservice Bad Smells," *IEEE Software*, vol. 35, pp. 56–62, May 2018.

[5] F. Palma, "Detection of SOA Antipatterns," in *Service-Oriented Computing - ICSOC 2012 Workshops*. Springer Berlin Heidelberg, Jan. 2013, pp. 412–418.

[6] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and and Technology Publications, Apr. 2016, pp. 137–146.

[7] O. Zimmermann, "Microservices tenets: : Agile approach to service development and deployment," *Computer Science - Research and Development*, vol. 21, pp. 301–310, Nov. 2016.

[8] M. Garriga, "Towards a Taxonomy of Microservices Architectures," in *Software Engineering and Formal Methods*. Springer International Publishing, Feb. 2018, pp. 203–218.

[9] J. Soldani, D. A. Tamburri, and W.-J. V. D. Heuvel, "The pains and gains of microservices: A Systematic grey literature review," *Journal of Systems and Software*, vol. 146, pp. 215–232, Dec. 2018.

[10] G. Marquez and H. Astudillo, "Actual Use of Architectural Patterns in Microservices-Based Open Source Projects," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec. 2018, pp. 31–40.

[11] F. Osses, G. Marquez, and H. Astudillo, "Exploration of academic and industrial evidence about architectural tactics and patterns in microservices," in *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE*. ACM Press, May 2018, pp. 256–257.

[12] D. Taibi, V. Lenarduzzi, and C. Pahl, *Microservices Anti-patterns: A Taxonomy*. Springer International Publishing, Jan. 2020, chapter 5, pp. 111–128, in Microservices: Science and Engineering, ISBN: 978-3-030-31646-4.

[13] R. Borges and T. Khan, "Algorithm for detecting antipatterns in microservices projects," in *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. CEUR-WS, Sep. 2019, pp. 21–29.

[14] R. Tighilt, M. Abdellatif, N. Moha, H. Mili, G. E. Boussaidi, J. Privat, and Y.-G. Guéhéneuc, "On the Study of Microservices Antipatterns: a Catalog Proposal," in *Proceedings of the 25th European Conference on Pattern Languages of Programs*, 2020, p. To appear.

[15] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele University*, vol. 33, pp. 1–26, Jul. 2004.

[16] M. I. Rahman, S. Panichella, and D. Taibi, "A curated Dataset of Microservices-Based Systems," in *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*. CEUR-WS, Sep. 2019, pp. 1–9.

 ISBN: 978-1-61208-777-1