Charting Microservices to Support Services' Developers: The Anaximander Approach

Sébastien Mosser[1(✉)], Jean-Philippe Caissy[1], Florian Juroszek[1,2], Florian Vouters[1,3], and Naouel Moha[1]

[1] Université du Québec à Montréal (UQAM), Montréal, Canada
{mosser.sebastien,caissy.jean-philippe,juroszek.florian, vouters.florian,moha.naouel}@uqam.ca

[2] Université Côte d'Azur (UCA), Sophia Antipolis, France
florian.juroszek@univ-cotedazur.fr

[3] Centre des Études Supérieures Industrielles (exia.CESI), Toulouse, France
florian.vouters@viacesi.fr

Abstract. Microservice architectures have gained popularity in the last ten years, based on their intrinsic capabilities of implementing scalable software architectures. However, understanding a microservice architecture is still a challenging task for software architects. Current state-of-the-art approaches addressing this challenge focus on exhaustive solutions, working in an all-or-nothing way. These all-or-nothing solutions rely on heuristics to create one *map* of a given architecture, using static and/or dynamic analysis of the existing source code. This is not compatible with the classical approaches used in software comprehension, that relies on the exploration of a program in an incremental way. In this paper, we leverage the *exploration* metaphor and describes the Anaximander approach to support the incremental definition of a map that suits the needs of the architect exploring an architecture. Using probes working at different levels of abstraction and precision, one can incrementally chart a map representing the architecture and leverage the map by querying it. We applied the Anaximander approach to six reference microservice architecture published by major actors from the state-of-practice.

Keywords: Microservice architecture · Software comprehension · Software composition

1 Introduction

Microservices are gaining momentum to support the development of complex service architecture. Relying on the promising principles of domain-driven design [12], microservices architectures provide an excellent answer to tame the challenge of developing scalable service-based systems. Such architectures are

© Springer Nature Switzerland AG 2020
E. Kafeza et al. (Eds.): ICSOC 2020, LNCS 12571, pp. 36–44, 2020.
https://doi.org/10.1007/978-3-030-65310-1_3

decomposed into a set of independent microservices, each of these being dedicated to a given domain. The communication between services is delegated to reliable communication paradigms, such as messages buses [2]. From a software engineering point of view, micro-services triggers several maintainability issues, *e.g.*,, how to maintain and evolve such systems.

Table 1. Size and technology heterogeneity for each reference architecture

Id.	Ref. architecture [1]	Technologies				Size		
		Lang.	DBs	Mess.	Depl.	#Files	#LoCs	#Serv.
S1	HipsterShop	5	1	2	2	163	38,934	10
S2	SockShop	3	4	2	9	222	19,014	8
S3	eShopOnContainers	1	7	5	3	1,585	143,356	8
S4	Vert.x MS Blueprint	1	7	2	1	218	18,881	9
S5	Shopping Cart	1	7	2	1	396	70,045	8
S6	Robot shop	4	3	2	1	120	6,341	7
Total for all arch.		*5*	*5*	*6*	*11*	*2,704*	*296,580*	*50*

In 2020, Assunçãao *et al.* described a variability challenge related to microservice engineering [1], where they identified six references open-source microservice architectures. These reference systems (see Sect. 4) demonstrate the high level of variability related to microservices development (Table 1). This level of heterogeneity is intrinsic to microservices architectures, and it is necessary to support developers and architects who have to maintain such systems. Reverse-engineering approaches typically support this task [9]. However, in the very case of microservices architecture, the quest for a fully-automated tool that can reverse-engineer any microservice architecture is pointless by design. On the one hand, static code analysis approaches will quickly reach a limit considering the flexibility offered to the developers by the existing technologies, and the upcoming frameworks that are not yet invented. On the other hand, dynamic approaches (*e.g.*, analyzing traces of execution) are fragile *w.r.t.* the scenarios used as input to capture the dynamic traces.

Instead of targeting an ultra-high-definition description of the architecture, we propose here to define an incremental and iterative way of creating such a description. The key idea is to consider such a description as a map, and leverage the way cartographers addressed the creation of maps in the early days of our civilization. We named our approach after ANAXIMANDER, a Greek philosopher known to have produced the first map of the world. Based on a source code audit of the reference architectures, we propose in this paper to describe an incremental approach to support developers and architects who maintain microservice architecture.

2 Related Work

Haitzer and Zdun [4] present a *Domain-Specific Language* (DSL) to abstract an application's architecture in a semi-autonomous way. This approach emphasizes that working incrementally is essential. Granchelli *et al.* [3] present an approach to recover the architecture of microservice systems called MicroART from a GitHub repository and a reference to the container engine managing the application. This approach differs from ours by using a monitoring tool such as `tcpdump` to capture the communication log between services without taking into account the architecture deployment artifacts. Kleehaus *et al.* [6] provides a tool called *MICROLYZE* to recover the infrastructure in real-time of a microservice architecture. Similar to our approach, *MICROLYZE* uses both automatic and manual processes to gather information. Ma *et al.* [8] propose another approach to generate service dependency graphs automatically. Those graphs are used to analyze and visualize the dependencies between the microservices deployed for the application. Their solutions allow them to select specific test cases in order to run regression tests on the application. Ma *et al.* explore similar monitoring solutions [7] to leverage annotation in Java source code. Those annotations are used to help build service dependency graphs.

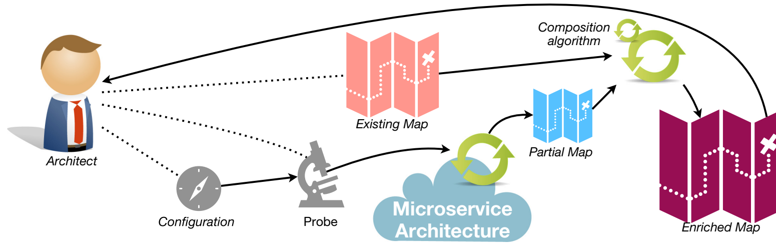


Fig. 1. Overview of the Anaximander approach

Leveraging the cartography metaphor, all the approaches described in this can be seen as exploration campaigns of the architecture, trying to create a complete map out of a single exploration. The maps are dedicated to a single objective (*e.g.*, non-regression testing) and cannot inter-operate with each other. Moreover, the amount of information produced is very detailed, and it might overwhelm an architect, preventing the approach to answer the architect's questions.

3 The Anaximander approach

Taking a different point of view, the key concepts of Anaximander are the definition of *(i) partial maps*, obtained as the result of the execution of *(ii) exploration probes* applied to the system. This approach tackles by design the

heterogeneity of micro-services architectures (see Sect. 4), and is complementary of the approaches already existing in the state-of-the-art that can be considered as *exploration probes*. We describe in Fig. 1 the approach for a software architect, that relies on the classical *extract - abstract - present* paradigm [10] used in reverse engineering. The architect selects a *probe* among the one available off the shelf, and execute it on the architecture. A probe can rely on static analysis, or dynamic traces. As a result of its exploration, a probe returns a partial map, *i.e.*, the information gathered by the probe. The obtained partial map is then composed with the already existing one (if any), to enrich the knowledge (*e.g.*, adding new information, correcting errors).

3.1 Modelling the Map as a Graph

We define an architecture *map* as a typical graph $g = (V, E) \in \mathcal{G}$, where $V = \{v_1, \ldots, v_i\} \in \mathcal{V}^i$ is a set of vertices and $E = \{e_1, \ldots, e_j\} \in \mathcal{E}^j$ a set of edges. A vertex v is defined as an vertex identifier, a type, and a set of associated properties P. An edge e is defined as a pair of source and target vertex identifiers, a type, and a set of properties. A property p is a simple key-value pair. To support the efficient manipulation of the maps, we rely on two constraints that need to hold in a given map: *(i)* vertex uniqueness and *(ii)* edge uniqueness.

To manipulate the map and support its enrichment, we leverage the classical *match and merge* algorithm [5]. Each graph element (*i.e.*, graphs, vertices and edges) defines an *equivalence relation* (denoted as $\equiv$) for matching purpose (*e.g.*, two nodes are considered equivalent when they have the same identifier), and a *merge* function (denoted as $\oplus$) to merge two elements identified as equivalent. Thus, enriching an existing map m with the result of a probe m' is simply to compute $m'' = m \oplus m'$. To correct an error, we rely on the opposite operation *remove* (denoted as $\ominus$), where the following law holds: $m = (m \oplus m') \ominus m'$.

3.2 Modelling Probes as Functions

Exploration probes are the software artifacts used to produce the partial maps. According to the heterogeneity of the technologies involved in microservices architectures, it is unrealistic to develop a polyglot framework supporting the state-of-practice as well as anticipating any upcoming technological trends. As a consequence, we decided to model a probe as a black-box function $p : conf \rightarrow \mathcal{G}$, taking as input its configuration, and producing as output a map, in a textual format. Adding or removing information to the map relies on the $\oplus$ and $\ominus$ operators previously described, *e.g.*, $m_{t+1} = m_t \oplus p(configuration)$.

The immediate advantage of this black-box representation is that it unifies the outcome of each exploration while supporting the designers of probes to use the most appropriate technologies for their very own probes. For example, a static analysis of Go source code will leverage the compiler capabilities directly embedded inside the Go language, where a probe dedicated to analyzing Spring Boot Java services will leverage the reflexivity API available in Java to analyze

the developed artifacts. Dynamic analysis can leverage classical formal models such as the *Knowledge Discovery Metamodel* [11], an international standard promoted by the OMG to support software modernization. To tame this heterogeneity and consider all the probes as equals from the architect point of view, it is possible to wrap each probe into an image (*e.g.*, using Docker or Singularity container technologies). The image will contain all the necessary software dependencies (*e.g.*, executable, compiler, libraries, frameworks) for a given probe, and hide this complexity to the architect into a black-box approach. It emphasizes the idea of probes' black-box representation, where the internal implementation details are hidden inside the container. The probes library available off-the-shelf is then a set of turn-key images ready to be used by the architect, and creating a new probe is as simple as publishing a new image inside the library.

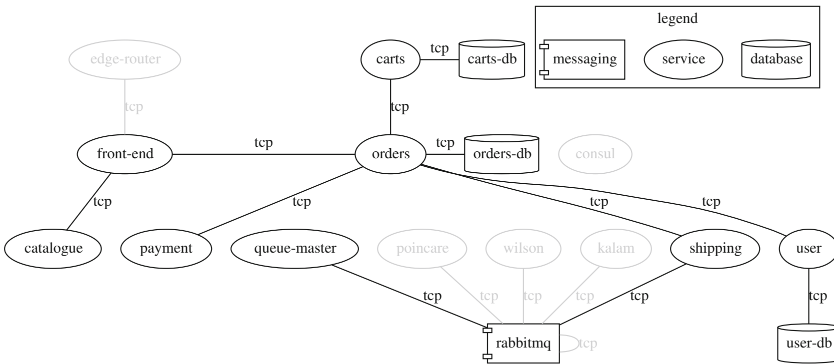


Fig. 2. ANAXIMANDER map obtained dynamically using WeaveScope (m_i)

4 Exploring a Reference Architecture

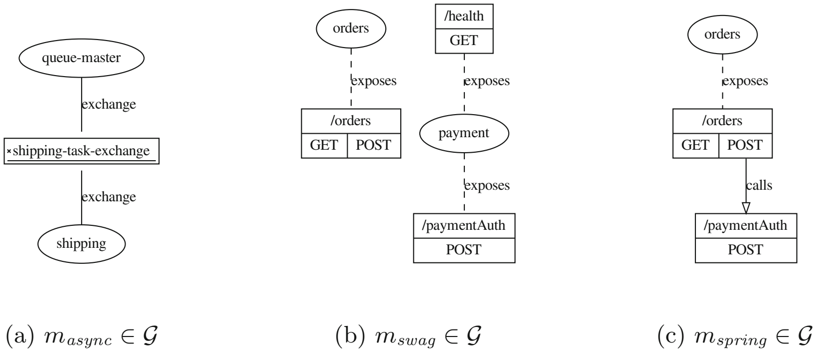
In this section, we validate the ANAXIMANDER approach based on the reference architectures used to express the requirements. The source code of the probes is available on the project repository[1]. For the sake of concision, it is not possible to provide here an in-depth analysis of each of the reference architecture. Instead, we focus on a single one (S2, *SockShop*[13]), as it is built as a demonstration showcase by a tool vendor (WeaveWorks), medium-sized concerning the five others, a representative in terms of heterogeneity (three languages for service development, three databases technologies, two messaging framework and nine deployment technologies), and involves 8 services.

As a starting point, we transformed the dynamic map provided by the tool vendor into an ANAXIMANDER artifact (Fig. 2). This first map m_i is the composition of three different information: *(i)* the server that host the services, *(ii)* the TCP connections that exist between the services and *(iii)* the kind of service

[1] https://github.com/ace-design/anaximander-microservices.

(*i.e.*, database, messaging, service). For the sake of readability, we only kept the two last ones in the paper version of the map. As the map is obtained by listening to a runtime infrastructure, it contains noise, *i.e.*, existing containers in the deployment infrastructure that are not related to the business logic (*e.g.*, `edge-router`, `consul`).

To remove the noise, we use a probe dedicated to extracting services from a Kubernetes descriptor. This probe extracts from the deployment descriptors the services into a map m_k, but cannot infer their interconnection. This is where the composition of multiple probes provided by ANAXIMANDER is helpful: to date, our most useful map is $m_0 = m_i \ominus (m_i \ominus m_k)$, *i.e.*,, the map containing all the discovered interconnection in m_i, without the infrastructure noise ($m_i \ominus m_k$).



(a) $m_{async} \in \mathcal{G}$ (b) $m_{swag} \in \mathcal{G}$ (c) $m_{spring} \in \mathcal{G}$

Fig. 3. Partial maps used to explore S2 with probes (RabbitMQ, Swagger, Spring)

Based on this initial map, we can start the incremental exploration of the infrastructure. First, we want to understand the interconnection that uses asynchronous messages (*e.g.*, RabbitMQ exchange topics) in this architecture. A query to m_0 to know all the services exchanging data with RabbitMQ returns two services: `queue-master` and `shipping`. It means that if the message bus suffers an outage, only the shipping infrastructure will be impacted. To improve the precision of the map concerning asynchronous communications, we use a source code analysis probe to identify the exchange topics from the source code, obtaining a map m_{async} (Fig. 3a).

A critical part of the architecture is the payment of orders, so we decide to explore the interconnection that exists between the `payment` and `order` services. Without more information, we assume that both services communicate using an HTTP REST protocol. We first use a probe dedicated to Swagger contracts identification, identifying the routes exposed by each service (m_{swag}, Fig. 3b). Then, we use a probe that performs a static analysis of the `order` service to identify the control-flow of its Spring implementation (m_{spring}, Fig. 3c). As there is no other connection between `order` and `payment`, we can use this information

to correct our initial map, and erase the technical `tcp` link that exists between the two services and use the proper control-flow instead.

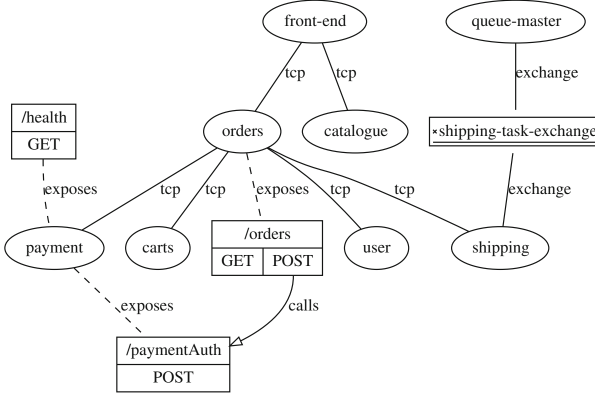


Fig. 4. Final map for S2, composing m_i, m_k, m_{async}, m_{swag} & m_{spring}

We describe in Fig. 4 the map obtained after these preliminary explorations. We used a query to identify the databases and remove them from the map, and then compose all the partial maps with the initial one to obtain a more precise picture of the architecture. The map is still shadowed for some services, but the amount of information inside it was sufficient to answer the questions we were asking about the architecture.

An immediate threat to validity is related to the lack of validation outside of the six reference architectures used to defined Anaximander. This is emphasized by the difficulty of collecting open-source microservice architecture, as this paradigm is used to implement business-driven logic. However, we mitigate this threat by the fact that the six architectures were highly heterogeneous, using different coding styles and technologies, and therefore representative of microservice development. Moreover, the representativity of these architectures is emphasized by their selection for a variability study by Asunçãao *et al.*

5 Conclusions and Perspectives

In this paper, we described a novel approach named Anaximander to support microservice architecture maintenance, leveraging the idea of gathering incomplete information about the architecture and composing this incomplete information with the existing ones to enrich the knowledge of the architect incrementally. This approach complements the state-of-the-art ones, which try to create ultra-precise maps by focusing on particular technological choices, where Anaximander support a more flexible way of creating such maps. The need for Anaximander emerged after a careful audit of six references architectures.

This work opens an interesting perspective concerning uncertainty. As the map created by Anaximander is imprecise by design and aims to be refined iteratively, finding a way to model such imprecision (*e.g.*, with goal modelling from the requirements engineering community) will help the architect during the exploration of the system.

Acknowledgments. This research has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), *Université du Québec à Montréal* (UQAM), and the Inria - *Équipe Associée* program.

References

1. Assunção, W.K.G., Krüger, J., Mendonça, W.D.F.: Variability management meets microservices: six challenges of re-engineering microservice-based webshops. In: 24rd International Systems and Software Product Line Conference, SPLC, Montreal, Canada. ACM (2020). https://variability-challenges.github.io/
2. Garg, N.: Apache Kafka. Packt Publishing Ltd., Birmingham (2013)
3. Granchelli, G., Cardarelli, M., Francesco, P.D., Malavolta, I., Iovino, L., Salle, A.D.: Towards recovering the software architecture of microservice-based systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 46–53, April 2017. https://doi.org/10.1109/ICSAW.2017.48
4. Haitzer, T., Zdun, U.: DSL-based support for semi-automated architectural component model abstraction throughout the software lifecycle. In: Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA 2012, pp. 61–70.Association for Computing Machinery, New York (2012). https://doi.org/10.1145/2304696.2304709
5. Kienzle, J., Mussbacher, G., Combemale, B., DeAntoni, J.: A unifying framework for homogeneous model composition. Softw. Syst. Model. **18**(5), 3005–3023 (2019). https://doi.org/10.1007/s10270-018-00707-8
6. Kleehaus, M., Uludağ, Ö., Schäfer, P., Matthes, F.: MICROLYZE: a framework for recovering the software architecture in microservice-based environments. In: Mendling, J., Mouratidis, H. (eds.) CAiSE 2018. LNBIP, vol. 317, pp. 148–162. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92901-9_14
7. Ma, S., Liu, I., Chen, C., Lin, J., Hsueh, N.: Version-based microservice analysis, monitoring, and visualization. In: 2019 26th Asia-Pacific Software Engineering Conference (APSEC), pp. 165–172, December 2019. https://doi.org/10.1109/APSEC48747.2019.00031. iSSN 2640-0715
8. Ma, S.P., Fan, C.Y., Chuang, Y., Liu, I.H., Lan, C.W.: Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Gener. Comput. Syst. **100**, 724–735 (2019). http://www.sciencedirect.com/science/article/pii/S0167739X19302614
9. Müller, H.A., Jahnke, J.H., Smith, D.B., Storey, M.A., Tilley, S.R., Wong, K.: Reverse engineering: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ICSE 2000, pp. 47–60. Association for Computing Machinery, New York (2000)
10. Müller, H.A., Tilley, S.R., Wong, K.: Understanding software systems using reverse engineering technology perspectives from the Rigi project. In: Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering, CASCON 1993, vol. 1, pp. 217–226 (1993)

11. OMG: Knowledge Discovery Metamodel 1.4. Technical report, OMG (2016)
12. Vernon, V.: Implementing Domain-Driven Design. Addison-Wesley Professional, Boston (2013)
13. Weaveworks: SockShop, a generic microservices application (2020). https://github.com/microservices-demo. Accessed 27 May 2020