Using Static Analysis to Address Microservice Architecture Reconstruction

Vincent Bushong
Dept. of Computer Science
Baylor University
Waco, Texas, United States
vincent_bushong1@baylor.edu

Dipta Das
Dept. of Computer Science
Baylor University
Waco, Texas, United States
dipta_das1@baylor.edu

Abdullah Al Maruf
Dept. of Computer Science
Baylor University
Waco, Texas, United States
maruf_maruf1@baylor.edu

Tomas Cerny
Dept. of Computer Science
Baylor University
Waco, Texas, United States
tomas_cerny@baylor.edu

***Abstract*—Microservice design offers many advantages for enterprise applications, including increased scalability and faster deployment times. Microservices' independence from one another in development and deployment provides these advantages. This separation, however, results in the absence of a centralized view of the application's functionality, and each microservice's data model is isolated and replicated. As a result, it has the potential to deviate from the architectural design's original intent. To address this, we offer a method for analyzing a microservice mesh and generating a communication diagram, context map, and microservice-specific limited contexts using static code analysis.**

***Index Terms*—Software Architecture Reconstruction, Reverse Engineering, Static Analysis, Microservices, Cloud-computing**

I. Introduction

Microservice Architecture is the emerging best practice in enterprise software development. All of the integration and business logic are implemented in the microservices themselves. No centralized model is agreed upon, except for the interface that each service will expose to the others. This decoupling means each microservice can be developed, deployed, and scaled independently, leading to more rapid development, evolution, and update cycles and improved performance of the entire system. However, with the missing global perspective of the overall system, it becomes challenging to control the holistic and distributed evolution, avoid consistency errors or poor design decisions leading to architectural degradation.

We propose a method of Software Architecture Reconstruction (SAR) to recover the holistic data model and inter-service communication of a microservice system. Our approach uses static code analysis with no need for dynamic analysis. We demonstrate our method on a large microservice testbed [1].

II. Background and Related Work

There are three broad categorizations for SAR methods: dynamic analysis, static analysis, and manual analysis.

Static code analysis has been used to identify calls between microservices to generate security policy automatically [2]. Also, it has been used to analyze monolithic applications to recommend splits for converting to microservices [3]. In generating a service dependency graph, Esparrachiari et al. posit that source code analysis is not sufficient since the deployment environment may impact the actual dependencies a given deployed module has [4]. However, our goal is different from theirs; we do not necessarily target every possible call in a system for dependency detection; rather, we find the calls that are part of the application's business logic, and for this purpose, the source code contains sufficient information.

Besides source code analysis, Ibrahim et al. used a project's Dockerfiles to search for known security vulnerabilities of the container images being used, which they overlay on the system topology extracted from Docker Compose files to generate an attack graph showing how a security breach could be propagated through a microservice mesh [5].

III. Static Analysis Approach to SAR

Our method does not need system runtime data; instead, it uses code analysis to identify microservice' endpoints and calls between individual microservices. In addition, it identifies data entities that individual microservices operate with. These identified calls allow us to extract a view of the system architecture before it is deployed, whereas other methods require a deployed system [4], [6]–[8]. Using this method, developers can get an updated view of the system's service APIs and service interactions as the code changes, rather than waiting for deployments. This API/interaction view was identified by Mayer and Weinreich as one of the most important aspects of a microservice monitoring tool [9]. This view also corresponds to the service viewpoint identified by Rademacher et al. as one of the fundamental viewpoints to be obtained from the SAR process on microservice-based systems [10]. We also extract a bounded context [11] for each individual microservice and combine them into a single *context map* for the entire system. This combines entities shared across microservices into a single combined entity, preserving all properties and relationships the entity is part of across the entire system. This view corresponds to the domain viewpoint also identified by Rademacher et al. [10], and to our knowledge, ours is the only microservice-oriented SAR tool to include this viewpoint. Such views reveal system details, current design state, serve as base for consistency checking, or architecture erosion assessment [12].

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research, https://research.redhat.com.

Creating a *context map* consists of two parts. First, each microservice project's bounded context has to be created, and second, those bounded contexts are combined into a context map for the entire system. Extracting a bounded context from each microservice requires each service to be analyzed. The first goal is to extract a list of all local classes used in the service. Once the classes have been identified, the next goal is to determine which of them are serving as data entities and which are not; for example, classes acting as REST controllers or internal services need to be filtered out. This is where we use enterprise standards to our advantage; development frameworks use standard components and constructs involving annotation descriptors that indicate a class's semantic purpose. For instance, there are multiple standards for persistence, input validation, transaction boundaries, synchronization, layering, and security. We use these descriptors to identify which classes are acting as entities. Even though we reference mostly Java, similar standards are adopted across platforms.

After the entities are identified, the bounded context of the microservice can be built. This is done by identifying the relationships the entities have with each other. These relationships have three different components, which we extract using static analysis: the types involved in the relationship, the multiplicity of the relationship, and the directionality of the relationship. Identifying the types is done on the basis of the type names of the entities' fields, the multiplicity can be determined by whether or not the field is a collection, and its directionality can be determined by whether or not there is a corresponding field in both of the entities involved or in only one entity.

A context map for the entire system is generated by merging the bounded contexts from for all microservices together. Since the mesh services operate on some of the same entities, the entities in each microservice can be merged by detecting if they have the same or similar names. Different bounded contexts may have different purposes for the entities they share and so may retain different fields from each other. Therefore, the next step is to merge the fields of merged entities. Fields with the same or similar names and the same data type are merged into a single field in the merged entity, while non-matching fields from all the source entities can simply be appended to the merged entity. The result is a context map that represents the scope of all entities used in the mesh.

Next, the code is analyzed to find calls among the microservices to create the communication diagram. This consists of two phases: identifying each service's API endpoints and finding where these endpoints are called from other services. With these two pieces of information, we can create a graph showing the paths of communication between services. Regardless of the method of extracting this information, certain metadata must be collected regarding the endpoints and calls; this includes the path, call method (HTTP/RPC), parameters, and return type (or the expected return type for a call).

Our method depends on using the standardized formats that enterprise standards use to encode this information. In enterprise applications, exposing endpoints is most commonly done in code using functions or annotations specific to a framework or library; this means the definitions will appear consistent each time they appear in code, so code analysis can be used to identify the metadata about defined endpoints. For instance, HTTP/RPC requests are made from a particular client, likely part of the same framework used to define the endpoints. Code analysis can also identify the metadata about every request in the system by finding the function call formats appropriate to the known library. Once the requests have been identified, they can be matched with the catalog of known service endpoints and protocols collected earlier; a match means there is a communication path between the two services.

Extracted communication diagrams and context map, reconstruct the system's architecture in how the services communicate among themselves and how the system treats its data.

To demonstrate the proposed approach, we implemented a prototype suited for Java Spring framework projects and applied it on the TrainTicket microservice benchmark [1]. The system consists of 41 microservices; of these, 36 microservices are Java-based, and they contain 27,259 lines of Java code, counted by the CLOC (Count Lines of Code) tool. The benchmark was analyzed using a MacBook Pro with a 2.9 GHz Quad-Core Intel Core i7 processor and 16 GB of RAM. Prophet took 2 minutes and 37 seconds on this device to clone, analyze, and generate the graphs for the repository. To manually analyze the project and enumerate the entities and inter-service calls took approximately 1.5 hours.

For class entities, manual analysis of the source code revealed 64 distinct entities in the project, and we were able to recover all of the entities, their properties, and their relationships, with the exception of one property and one relationship. Prophet also reported five additional classes that were not considered entities during the manual inspection because they were being used as data transfer objects (DTO) instead of entities. The manual analysis showed 153 connections between the services and their endpoints; separate calls. We identified 135 of these, missing 18 calls. The missing calls were all due to those calls choosing from multiple potential URLs, an ambiguity our tool was not designed to detect.

IV. Conclusions

We have presented a method to analyze a microservice mesh and automatically generate an up-to-date data model and communication diagram for it. This novel method provides several contributions: it is much faster than creating these artifacts manually; it can be re-run as often as the project changes; it effectively serves as a centralized source of documentation of a distributed system that does not introduce further coupling between the services. We demonstrated it on a case study with promising results and plan to expand it to other frameworks and languages to face heterogeneity in future work.

Our tool git-repo: https://github.com/cloudhubs/prophet-utils.

References

[1] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering

research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 323–324. [Online]. Available: https://doi.org/10.1145/3183440.3194991

[2] X. Li, Y. Chen, and Z. Lin, "Towards automated inter-service authorization for microservice applications," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 3–5. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1145/3342280.3342288

[3] S. Eski and F. Buzluca, "An automatic extraction approach: Transition to microservices architecture from monolithic application," in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, ser. XP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1145/3234152.3234195

[4] S. Esparrachiari, T. Reilly, and A. Rentz, "Tracking and controlling microservice dependencies," *Queue*, vol. 16, no. 4, pp. 10:44–10:65, Aug. 2018. [Online]. Available: http://doi.acm.org/10.1145/3277539.3277541

[5] A. Ibrahim, S. Bozhinoski, and A. Pretschner, "Attack graph generation for microservice architecture," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1235–1242. [Online]. Available: https://doi-org.ezproxy.baylor.edu/10.1145/3297280.3297401

[6] B. Mayer and R. Weinreich, "An approach to extract the architecture of microservice-based software systems," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2018, pp. 21–30.

[7] G. Granchelli, M. Cardarelli, P. D. Francesco, I. Malavolta, L. Iovino, and A. D. Salle, "Towards recovering the software architecture of microservice-based systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 46–53.

[8] K. A. Torkura, M. I. Sukmana, and C. Meinel, "Integrating continuous security assessments in microservices and cloud native applications," in *Proceedings of The10th International Conference on Utility and Cloud Computing*, ser. UCC '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 171–180. [Online]. Available: https://doi.org/10.1145/3147213.3147229

[9] B. Mayer and R. Weinreich, "A dashboard for microservice monitoring and management," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 66–69.

[10] F. Rademacher, S. Sachweh, and A. Zündorf, "A modeling method for systematic architecture reconstruction of microservice-based software systems," in *Enterprise, Business-Process and Information Systems Modeling*, S. Nurcan, I. Reinhartz-Berger, P. Soffer, and J. Zdravkovic, Eds. Cham: Springer International Publishing, 2020, pp. 311–326.

[11] C. and Trnka, Michal, "Contextual Understanding of Microservice Architecture:Current and Future Directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 29–45, 2018.

[12] A. Baabad, H. B. Zulzalil, S. Hassan, and S. B. Baharom, "Software architecture degradation in open source software: A systematic literature review," *IEEE Access*, vol. 8, pp. 173 681–173 709, 2020.