MDRM Pattern: Relational heterogeneous metadata management patterns in a microservice environment

Dongmin Kim*
Human IT Convergence Research Center
Korea Electronics Technology Institute (KETI)
Gyeonggi-do, Republic of Korea
dmkim@keti.re.kr

Jaegi Son
Human IT Convergence Research Center
Korea Electronics Technology Institute (KETI)
Gyeonggi-do, Republic of Korea
jgson@keti.re.kr

***Abstract*—In microservice applications developed for specific tasks, multiple comprising units may have the same purpose, but the data generated from them may not be related or consistent. This leads to bottlenecks such as low data utilization, making it difficult to implement them in applications. Herein, we propose a microservice data relationship management (MDRM) pattern, which is a heterogeneous metadata management pattern with the ability to maintain relationships with data generated by multiple units via organic relationships. The MDRM pattern is basically a microservice architecture data management pattern that determines data keys shared with each other in metadata generated by a number of units comprising a microservice application. It performs data operations based on keys, including creation, read, update, and deletion, and manages relationships between heterogeneous metadata. This pattern aids in configuring microservice applications where data collected from service units with different metadata must be managed in a single report form, and aggregated or integrated for monitoring. In addition, this pattern can be easily used not only as relational metadata in microservice environments but also in distributed computing applications.**

***Index Terms*—Cloud Native, Microservice, Microservice application, Relational heterogeneous data, Design Pattern**

I. Introduction

Microservice [1] is a new software architecture that can organize multiple functional units or small functions to create one large application, and can be easily changed, combined, and distributed. Unlike the monolithic application structure, which includes all functions in a standalone program, microservice is widely used in the cloud service application area, because it can expand large-scale systems fast, accurately, and easily using cloud-native technology.

In addition, microservices allow multiple functions or small functions to be deployed and updated independently, and can focus only on individual parts, teams, or developers, making it easier to perform application-related tasks without side-effects. Thus, microservices have been adopted by enterprise companies globally, including Amazon, Netflix, Uber, and Etsy. Microservices are well-known for improving business agility and achieving high profits [2].

In microservice structure-based applications, different units may have the same purpose, but the data generated from them may not be related or consistent. This leads to bottlenecks such as low data utilization, making it difficult to implement them in applications.

Fig. 1 compares the data management (creation, reading, deletion, and update) system of monolithic and microservice structures. Fig. 1(a) shows the data management system in a monolithic structure, while Fig. 1(b) shows the data management system in the microservice structure. For microservices that can be deployed and updated independently, their databases are also used independently for each unit when necessary. If data are generated for each unit function comprising an application and a relationship is required with the generated data, the application of the microservice structure is inappropriate.

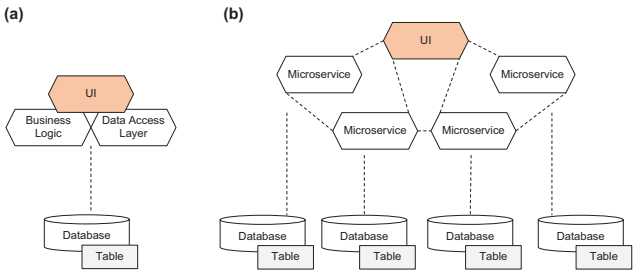

Fig. 1. Data management system according to software structure: (a) monolithic structure, (b) microservice structure.

The use of complex data in microservices is of significant interest to service developers and managers. Rodrigo Laigner's study [3] presented the current status and research direction for data management in microservices through a survey of 120 experienced service developers. The results of the survey solve the problem of heterogeneous data utilization by observing that service developers rely on the temporary design due to the lack of holistic data management solutions for microservices, and demand for database management systems (DBMSs).

A study by Braun and Eric [4] proposed a back-end technology for heterogeneous data management in microservice architectures. By providing a low-combination data service set in a microservice structure in a module unit, they defined a data model that can be reused in a microservice unit and a

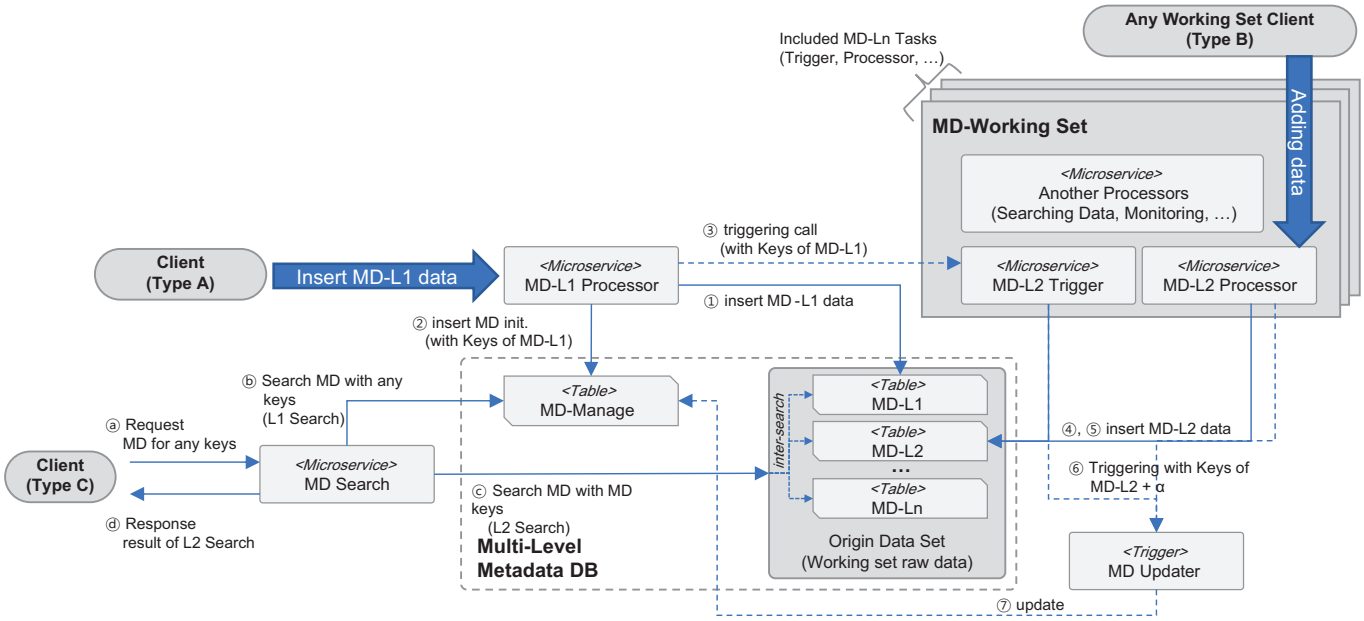

Fig. 2. MDRM pattern structure.

data management layer that can selectively apply a database according to the purpose. This maximizes the benefits of microservices by providing scalability to utilize heterogeneous databases, so that separate services can be defined for different data types and management tasks, and by providing low combinations between different functions. Non-structured microservice applications can support heterogeneous databases; however, they complicate the structure of microservices, making it more difficult to assign links or relationships between data.

A study by Ntentos and Evangelos [5] proposed an architecture design decision (ADD) model that induces the correct selection of microservice data management architectures. The model supports selective decision-making of design patterns that include a total of 35 knowledge sources established in the form of qualitative research for data management in microservices. Although this study provides the most suitable pattern among 35 well-known patterns for service developers, there is no pattern for dealing with heterogeneous data with relationships in the ADD model candidates. In addition, it is difficult to obtain results by applying them directly to the ADD model, even if a new pattern is added to support them.

In addition, previous studies have focused on microservice patterns and practices for data management in microservices. [6, 7, 8, 9]. Unfortunately, these results are organized only in developer blogs, technical documents, private storage within open source sites, and system documents. Thus, there are limitations in defining general architecture concepts, and there are uncertainties and risks inherent in actual data management in microservices owing to biased results from certain domains.

In this study, we propose microservice data relationship management (MDRM), a heterogeneous metadata management pattern that has the function of maintaining relationships with metadata generated by multiple units with organic relationships in a microservice environment. The basic concept of the MDRM pattern is to define a data key shared by the metadata generated by a number of units comprising a microservice application, and to relate heterogeneous metadata by triggers such as creation, read, update, and deletion. This pattern helps to configure microservice applications where data collected from service units with different metadata must be managed in a single report form, and aggregated or integrated for monitoring. In addition, this pattern can be used not only in microservice environments that continue to expand because of low combination of units, but also in distributed computing applications.

II. MDRM Pattern

A. Architecture overview

The MDRM pattern is a microservice design pattern that admits metadata in the form of json. Metadata generated in the microservice unit are managed at multi-level to maintain relationships between data, and data operations such as creation, reading, updating, and deletion. Fig. 2 shows the structure of the MDRM pattern, and the essential components of the MDRM pattern are summarized in Table I.

B. Scenario

The MDRM pattern scenario is divided into a microservice unit relationship definition stage, a metadata generation stage,

TABLE I
COMPONENTS OF MDRM PATTERNS

Components	Descriptions
Client	The client is a subject that performs data operations in a microservice application developed in the MDRM pattern and can be a component (such as a user) outside the application or a microservice unit in the application (Fig. 2 illustrates Type A, B for data generation and Type C for data access).
MD-Manage	MD-Manage is an intra-database table that manages metadata and manages key data for the relationship of heterogeneous metadata.
Origin Data Set	The Origin Data Set is an in-database table for storing metadata generated in microservice units and is created/managed in microservice units that generate metadata.
MD-L1 Processor	The MD-L1 Processor is a metadata level 1 processor, a microservice unit that processes initial data for relational metadata.
MD-Working Set	MD-Working Set is a set of microservice units for relational metadata; depending on the number, it can be extended to MD-L2, L3, ..., Ln, and each set comprises a Trigger and a Processor.
MD-Ln Processor	The MD-Ln Processor is a unit within the Nth microservice set that stores data generated by the microservice in the corresponding microservice data table within the Origin Data Set.
MD-Ln Trigger	The MD-Ln Trigger is a unit within the Nth microservice set. This is a trigger called through key data stored by the MD-L1 processor if related to the metadata Level N microservice and the data covered by the MD-L1 processor.
MD Updater	MD Updater is a microservice unit called from the MD-Ln Processor and the MD-Ln Trigger that performs the function of referencing or updating data in MD-Manage if necessary.
MD Search	MD Search is a microservice unit that performs the function of querying previously-stored relational metadata.

an additional metadata generation and data relationship formation stage, and a data inquiry stage. Each stage is described as follows.

1) **Step 1: Define microservice unit relationships**
   This step defines the microservice units that generate relational metadata, which is described in detail with examples in the next subsection.
2) **Step 2: Create metadata**
   This step is a data generation step for relational metadata in a microservice application, initiated by the MD-L1 Processor. The data delivered to the MD-L1 Processor is stored in the corresponding microservice data table within the Origin Data Set, of which key data is also stored in MD-Manage.
3) **Step 3: Create additional metadata and build data relationships**
   Additional metadata generation proceeds within the MD-Working Set. Depending on the relationship defined in step 1, MD-LnTrigger may be called by the MD-L1 Processor, or metadata may be generated by another client (or microservice) to call the MD-Ln Processor. If the data to be stored is related to the existing data stored by the MD-L1 Processor, the key data in MD-Manage may be referenced or updated.
4) **Step 4: Look up data**
   In microservice applications, inquiry of relational metadata is performed by MD Search, and requires key data stored in MD-Manage. Key data stored in MD-Manage is required to obtain table information, in which relational data in the Origin Data Set are stored. MD Search, which acquires table information, inquires and collects data for each table, and delivers the inquiry result to the data inquiry requestor (client or microservice).

C. Example of Killer application with MDRM pattern: cargo container management program structure

Here, we illustrate the microservice data management pattern as an example of the structure of the cargo container management program to which the MDRM pattern is applied. Cargo container management is managed with container information (including container number and import/export date), certificate files and metadata containing inspections of dangerous goods, such as drugs and explosives (in the example below, certificate files are managed as object data encoded or decoded in base64 format).

Fig. 3 is a relationship definition table for step 1 of the MDRM pattern. This table stores the arbitrary code number corresponding to the microservice, the name of the MD-Ln Trigger called through the MD-L1 Processor, and the table name in the Origin Data Set.

<Table> working_set_info			
code	**name**	**trigger_name**	**db_table**
00000	Container_RegInfo	-	container_reg
DA001	Danger Manager	danger_trigger	danger_table
EE331	Drug Manager	drug_trigger	drug_trable

Fig. 3. Example of microservice metadata relationship definition table for cargo container management.

Fig. 4 shows the order in which container information is first entered. The Container Register corresponding to the MD-L1 Processor stores metadata in the table (container_reg) and MD-Manage table within the Origin Working Set. In this case, the key of the relational metadata is the reg_no field value of the container_reg table. In addition, the metadata associated with this data is dangerous goods data with the 'DA001' code.

Figure 5 shows an example in which the MD-Working Set is used for relational metadata management after initial data input. The Container Register looks up the 'DA001' code number and calls the corresponding trigger, danger_trigger. In this case, the reg_no value, which is a relational metadata key, is transmitted together. The danger_trigger calls MD Updater after adding data to its table in the Origin Working Set with the received key data. MD Updater updates the relational data code number in the MD-Manage table.

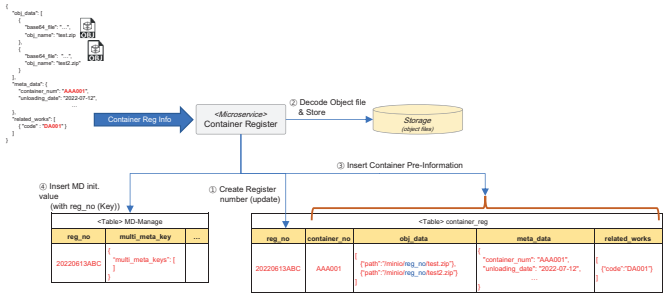

Fig. 4. Example of initial input of relational metadata for container management.

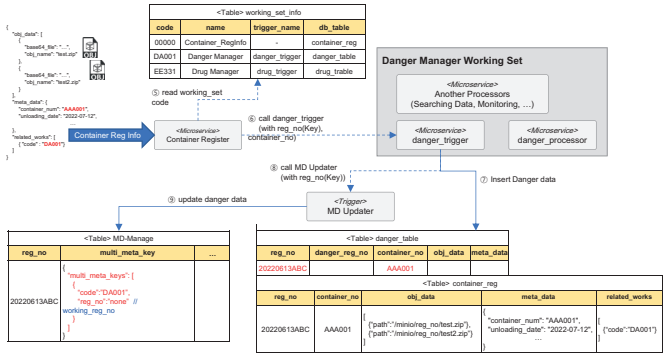

Fig. 5. Examples of entering and updating relational codes for relational metadata for container management.

Fig. 6 shows an additional input example of metadata by the client of the MD-Working Set. The danger_processor corresponding to the MD-Ln Processor looks up the relational metadata key (reg_no) for inserting metadata into its working-set table, 'danger_table', and inserts metadata into the field where the key data is entered (actually performing an update on that field). After metadata insertion, the danger_processor calls MD Updater to update the field data along with the registration number of the metadata additionally inserted into the field, where the relational metadata key exists in the MD-Manage table, and performs all the data insertion steps (Steps 1 to 3 are completed).

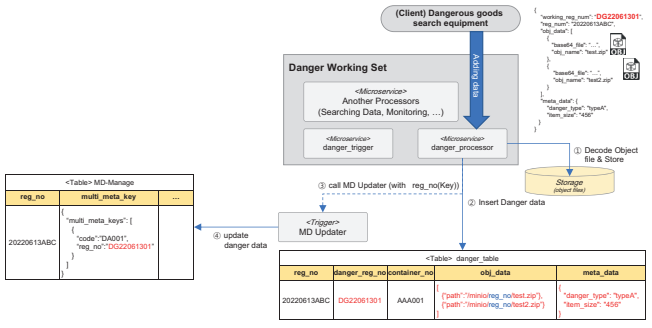

Fig. 6. Example of creating relational metadata by other clients.

Fig. 7 shows the step of inquiring about the previously stored relational metadata. The Container MD Searcher, which corresponds to MD Search, looks up data from the MD-Manage table by a relational metadata key. The Container MD Searcher then queries all related tables in the Origin Data Set based on the inquired data, configures the return data, and returns it to the service requester as a result of the request (Step 4 Completed).

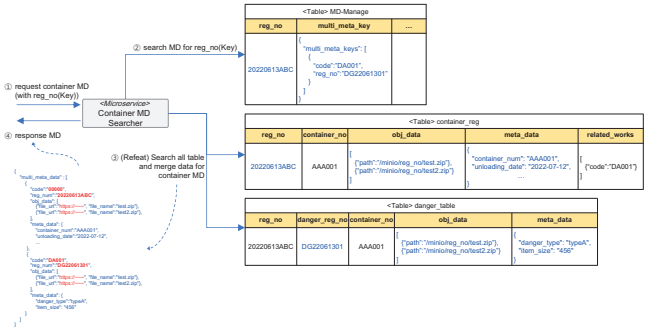

Fig. 7. Example of relational metadata lookup

III. Conclusion

In this study, the MDRM pattern, which is a microservice architecture data management pattern for maintaining/managing the relationship between metadata generated in microservice units, was proposed. The management of relational metadata based on microservices can also make metadata management easier by upgrading the management system (such as DBMS) and table design of existing databases. However, since the MDRM pattern proposed in this study is a design pattern for microservices, there is a difference in that it can support microservice application design more easily and quickly.

In addition to the structure of the cargo container management program described as an example of this study, the proposed MDRM pattern can be used in all application structures where the relationship between metadata created in each microservice unit exists, such as application structure for heterogeneous sensor data collection/management or employee history management structure within a company.

Acknowledgment

This work was supported by Korea Institute of Marine Science & Technology(KIMST) grant funded by the Korea government(Ministry of Oceans and Fisheries) (No.PJT201289, Development of a platform for sharing and providing private cloud based container search information)

References

[1] Gary Olliffe. "Microservices : Building Services with the Guts on the Outside", (2015). [Online]. Available: "https://blogs.gartner.com/gary-olliffe/2015/01/30/microservices-guts-on-the-outside/"

[2] Jeremy H. "4 Microservices Examples: Amazon, Netflix, Uber, and Etsy", (2022). [Online]. Available: "https://blog.dreamfactory.com/microservices-examples/"

[3] Rodrigo Laigner, Yongluan Zhou, et al., "Data management in microservices: State of the practice, challenges, and research directions", arXiv preprint arXiv:2103.00170, 2021.

[4] Braun, E., Schlachter, T., Düpmeier, et al., ”A Generic Microservice Architecture for Environmental Data Management.”, Environmental Software Systems. Computer Science for Environmental Protection., ISESS 2017. IFIP Advances in Information and Communication Technology, 2018, pp. 383–394.
[5] NTENTOS, Evangelos, et al. “Supporting architectural decision making on data management in microservice architectures”, European Conference on Software Architecture. Springer, Cham, 2019. pp. 20-36.
[6] Gupta, A. “Microservice design patterns”, (2017). [Online]. Available: “http://blog.arungupta.me/ microservice-design-patterns/”
[7] Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. (2014). [Online]. Available: “http://martinfowler.com/articles/microservices.html”
[8] Pahl, C., Jamshidi, P, “Microservices: A systematic mapping study”, 6th International Conference on Cloud Computing and Services Science. 2016, pp. 137–146
[9] Richardson, C. “A pattern language for microservices”, (2017). [Online]. Available: “http://microservices.io/ patterns/index.html”