Analysis of Data Exchange Among Heterogeneous IoT Systems

Jannik Laval, Nawel Amokrane, Mustapha Derras, and Néjib Moalla

Abstract Data interoperability allows data exchanges among information systems, their sub-systems and their environment. The multiplicity of these exchanges and the increasing amount of exchanged data can generate dysfunctions with negative impact on the overall performance of the communicating systems. Data interoperability should therefore be continuously assessed and improved. We propose a messaging metamodel that aggregates collected information from several pub/sub-communication protocols, and we present a work in progress which utilizes services provided by AMQP, MQTT, CoAP and Kafka to collect information in order to analyze data exchanges. Including these pub/sub-communication protocols and the data analysis platform Moose to achieve monitoring, we propose the pulse framework that provides a tracking of architecture changes in the pub/sub-systems. We analyzed the differences between the protocols to provide a generic metamodel to include all of these pieces of information in the same system. It will allow to extract precise information about the evolution of the system.

Keywords Data interoperability · Data analysis · Monitoring · Message brokers

1 Introduction

The problem of interoperability between heterogenous systems already exists and is amplified by the strong deployment of Internet of Things. To respond to this challenge, enterprises address this problem by emphasizing on the use of open standards for data format as well as communication protocols. Despite these efforts, interoperability is still a real issue that cannot be ignored.

J. Laval (✉) · N. Moalla
University Lumière Lyon 2, DISP lab EA4570, Bron, France
e-mail: Jannik.Laval@univ-lyon2.fr

N. Amokrane · M. Derras
Berger-Levrault, Lyon, France
e-mail: Nawel.amokrane@berger-levrault.com

B. Archimède et al. (eds.), *Enterprise Interoperability IX*, Proceedings of the I-ESA Conferences 10, https://doi.org/10.1007/978-3-030-90387-9_7

Distributed message brokers are typically used to decouple separate stages of a software architecture. They permit communication between these stages asynchronously, by using the publish/subscribe (pub/sub) paradigm. Implementing a message-oriented middleware enables asynchronous communication which allows applications to be more loosely coupled. As a result, available resources can be better utilized and systems performance improved. These message brokers are also finding new applications in the domain of IoT devices and may also be used as a method to implement an event-driven processing architecture.

The increase in the number of exchanges and consequently in the amount of data leads to the need to deploy monitoring and analysis systems. During exchanges in an event-oriented system, monitoring can be carried out at different levels (e.g., at the level of a node, an exchange, messages, etc.) or even on the entire system. Several different tools are needed. For example, RabbitMQ offers a management console to monitor the status of messages and the status of each element of an AMQP system. This console shows the list of resources, their characteristics, and some statistics. This console is suitable when the maintainer focuses his analysis on a particular node and knows the structure of the system. For example, when a queue is accessed, the messages in that queue are displayed. This allows an analysis of the situation at a particular time. However, when you want to do more complex analysis, advanced queries on resources, then these tools are not adapted. They require time-consuming manual work. For example, consumed messages are no longer presented, so it is not possible to follow their evolution. The entities in the structure (exchanges, waiting lines, etc.) also disappear from the management console as soon as they are deleted. Thus, when a consumer disconnects, these elements also disappear and are no longer usable. The management console does not allow you to view the history of a system's resources.

When considering existing open-source and monitoring tools (Nagios, Zabbix) that are great enterprise level software designed to monitor everything from performance, availability of servers, network equipment to web applications and database, we notice that they are capable of monitoring components like network protocols, operating systems, web server, website, middleware and so on, but only focusing on low level monitoring information such as, performance indicators or memory usage. Our approach uses monitoring for the assessment of interoperability, an analysis capable of defining a classification of potential causes by order of importance for a given problem. A monitoring system is defined as a process or a set of distributed processes including collection, interpretation and dynamic processing of information related to an application being run.

The messaging data model presented in Amokrane et al. [1] aggregates data collected related to message exchanges and is created for the Berger Levrault messaging infrastructure. It provides a common control point and facilitates the extraction of interoperability related indicators. The messaging data model describes the messaging structure implemented through message queueing and exchange system. It is used to collect meta-information from log services offered by the exchange infrastructures and keeps track of the exchanged messages.

In this paper, we extend the messaging metamodel to consider a more generic model adapted to messaging paradigms. We consolidate it by analyzing AMQP broker, MQTT broker, KAFKA broker and CoAP server. We also propose the pulse framework that uses this model to collect meta-information to be able to (i) keep track of the exchanged messages, (ii) simplify the visualization of exchanges, (iii) enhance the maintainability by detecting exceptions (ex: problem of transfer of a message), precising of the context and the origin of the problem and providing alerts and notifications. The pulse framework integrates dynamic features, where the lifecycle of different components of the architecture is depicted by including creation and deletion dates as well as timestamps.

In the remaining sections, Sect. 2 presents related work. Section 3 exposes the pulse framework and related tools that enable the evaluation of data interoperability. Section 4 describes its underling metamodel. Implementation is described in Sect. 5. Section 6 concludes this article and opens perspectives.

2 Related Work

Interoperability assessment evaluates the ability of enterprises or systems to undertake common activities or exchange data. Several interoperability assessments approaches have been proposed since the emergence of the concept of interoperability: maturity models (LISI, LCIM, OIM), interoperability score [2] or degree of interoperability [3]. However, these methods do not allow to precisely indicate or locate interoperability problems and mainly focus on general notions. Also, few interoperability assessment methods address the effective (post implementation) evaluation of data interoperability and few are tooled [4]. These methods have nonetheless provided the fundamental concepts that allow formalizing and evaluating interoperability by indicating whether interoperability problems exist or not. Based on these concepts, other approaches [5, 6] have defined a set of interoperability requirements (e.g., “partners provide permissions for data updates,” “received data is conform to required data”) that should be verified to achieve interoperability.

In terms of existing tools allowing monitoring, we can mention: ELK Stack [7] and Qlik Sense. ELK Stack is the combination of three open-source tools Elasticsearch, Logstash and Kibana. Elasticsearch is a No-SQL database with a focus on search and analysis capabilities, Logstash is a log aggregator that gathers data from different sources, transforms, enhances it and sends it to different output destinations and Kiabana is a visualization tool that works on top of Elasticsearch. Qlik Sense is a dashboard solution that enables one to visualize and preform data analytics. It supports interactive and dynamic visualizations; it is flexible and multiplatform.

3 The Pulse Framework

To explore communication rules allowing to operate connected objects and to reach data exchanges monitoring, visualization and adaptation through pub/sub-messaging model AMQP, MQTT, KAFKA and CoAP protocols, the general prototype framework is demonstrated in Fig. 1. It represents the collected data from different protocols related to the exchange of messages and the log of events. The proposed monitoring system is composed of four layers: (i) data importing and model population layer, (ii) time management and model versioning layer, (iii) persistence layer and (iv) visualization and analysis layer.

3.1 Data Importing and Model Population Layer

One of the main challenges is to import data from different sources with different formats. For that, we define a metamodel representing the structure of a distributed exchange system. The metamodel is detailed in the next section. The dedicated importers take data as input and instantiate the model with the information.

3.2 Time Management and Model Versioning Layer

The instantiation of the messaging model (via the different collecting components) revealed the issue of the historization of the versions of the model to take into account

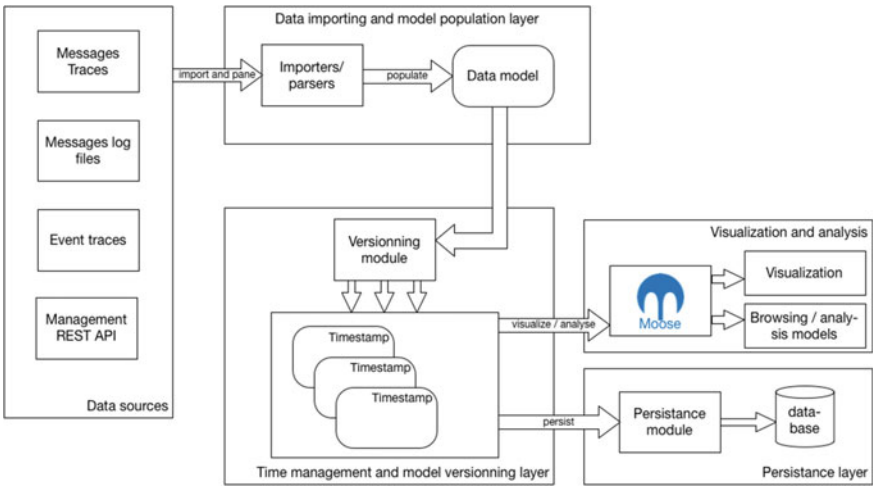

Fig. 1 Pulse framework architecture

the dynamic aspects of the system. We need a kind of historization to be able to understand the events in earlier versions of the architecture and to favor a better analysis for maintenance needs.

A trivial method would be to integrate a timestamp for creation, deletion and update to each entity of the model. The problem with this approach is the strategy to build a specific status at a specific time. Another method can be the creation of a new model each time there is an update, which takes big space at each new data coming from the importers.

We consider another solution based on Orion [9]. Orion is a model that enables creating different versions of a data model considering the tracking of changes in this model. The principle behind Orion is that each change triggers an Orion action which is responsible for adding the change to the data model. Each change can result in an updated version of the data model. Orion optimizes the persistence of different versions of the model, where Orion handles deltas and pointers to earlier versions. Orion copies the sole entities that have been impacted by a change. Figure 2 illustrates our versioning strategy. This version management of the data model allows us to follow the evolution of the messaging architecture over time, where each version represents an image of the architecture at a given moment.

To resume, an Orion version includes the latest changes and information about the action that was preformed to create this version. For the user, each version represents

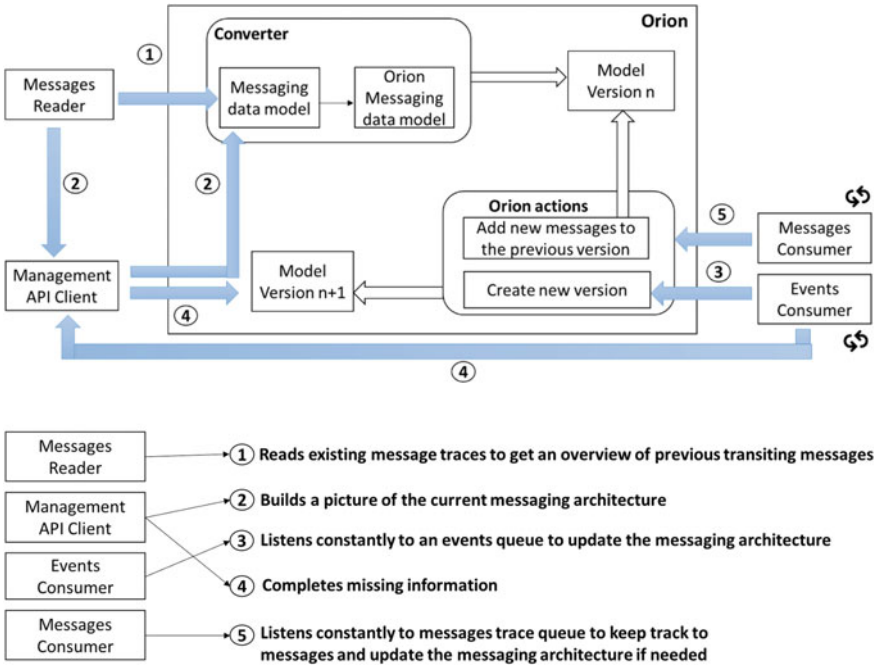

Fig. 2 Orion and the model versioning process

a screenshot of the monitored system in a specific time. In other words, instead of having an overcrowded unusable model, Orion provides multiple small models, each of them describing a change to the system at a certain time.

We defined a strategy to create a version each time it is necessary. In the case of a message exchange system, we define two kinds of events.

- A change in the architecture or to the configurations/settings of the monitored system (queue creation, queue deletion, user permissions changed, etc.). The status of the system before and after the change must be kept. So, for each of these changes, an Orion version is created.
- A new trace (new message published, message received, new connections, etc.). In this case, the framework instantiates a dedicated entity in the current version of the model. It is not necessary to create an updated version.

3.3 Persistence Layer

Orion keeps different versions of the model in the Pharo[1] image of Moose, and due to the way that Orion versioning system works and the fact that entities that do not change are not copied from version to version but rather a reference to unchanged entities in the previous version is copied, storing different versions in memory will not pose a space problem, but for the long run, we needed a way to store our models in a persistent way. With this feature, we enable external systems like Grafana[2] to acquire metrics and use them to display certain visualizations.

When our persistence module is called, it stores all versions of the Orion model to a json file. It can be extended to output other kind of structured data.

3.4 Visualization and Analysis Layer

Implementing model versioning enabled us to preform two things: analyze and visualize changes to the monitored system in real time as they occur and to go back in time to a previous state of the monitored system to visualize and analyze changes and their impact at a giving time.

These two features allow us not only to detect interoperability issues as they occur but also to identify the potential source of a certain problem by going back to previous states.

[1] https://pharo.org/.

[2] https://grafana.com/.

4 Pulse Metamodel

The goal of the pulse metamodel (Fig. 3) is to represent three aspects of the messaging structure:

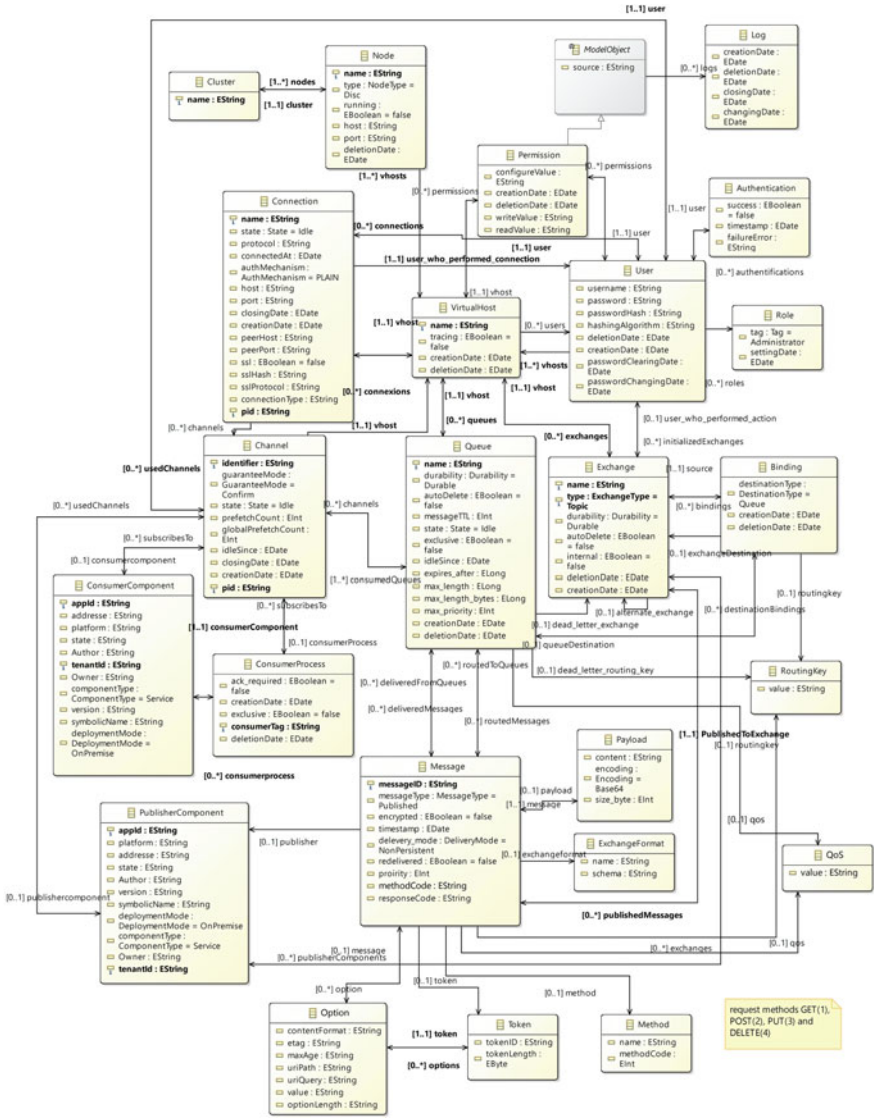

Fig. 3 Pulse metamodel

Table 1 Comparing concepts with the AMQP-based model

Protocols	Similar concepts	Different concepts
MQTT	Cluster, V-host, user, connection, channel, exchange, binding, queue, routing keys, message, security	QoS, persistent session
Kafka	Cluster, user, connection, channel, message, security	Topic, partition, QoS
CoAP	Cluster, V-host, user, connection, channel, exchange, binding, queue, routing keys, message, security	QoS, token, options, request methods

- A static representation: the messaging structure implemented through message queuing and exchange system.
- A dynamic representation, where the messages flow from publishers to consumers is represented.
- The lifecycle representation of architecture, where components (e.g., queue, exchange, …) are created, modified, deleted.

This metamodel is aimed to be generic enough to consider different protocols, as these protocols can be used to set up data exchanges within information systems and with their environment. We extend a previously presented metamodel [1] mainly based on AMQP with other protocols: AMQP, MQTT, Kafka and CoAP protocols. The analysis of these protocols allowed as to determine similarities and differences comparing to AMQP (Table 1).

- AMQP is a standard protocol for exchanging messages between applications. It is message-oriented and allows asynchronous communication using queues. It makes applications interoperable by facilitating communication and offers message encryption. The protocol is used in client/server communications and in the management of IoT devices. The structure of the system is message-based: The message is created by a message producer, passes through an exchange and transits through queues according to routing keys to finally reach the message consumer. Each message contains a header and a content. The header is formatted according to an exchange format and contains a set of metadata useful for routing the message. The content contains the message to be processed by the message consumer. The message broker manages all these elements. Clients connect using identifiers.
- MQTT [9, 10] is a protocol that enables specialized message exchanges for lightweight machine to machine and IoT communications. Like AMQP, it is a publish/subscribe protocol. It works in a similar way to AMQP. Some differences are: It has a payload containing the message, and a minimalist header. A session indicator shows whether a persistent session has previously been created with the client. The real difference with AMQP is the presence of a quality-of-service parameter. The broker allows one to choose one of three levels of service. This level of service ensures that a message has reached its destination.

- Kafka [11, 12] is a communication protocol optimized for rapidly distributing messages between applications in a scalable manner. It is a publish/subscribe protocol with a message storage system designed as a distributed transaction log. This system allows for continuous data processing. Thus, a record in the transaction log contains a key, a timestamp and the content of the message. A Kafka cluster stores these transactions in categories called topics, using the same principle as AMQP. Topics are organized into partitions. Each broker can have several partitions. A partition is an ordered sequence of records.
- CoAP [13] is a communication protocol specialized in constrained systems. It has been defined in RFC 7252. CoAP's main objective is to address the constraints of specific environments, e.g., energy management in building automation. CoAP is based on the UDP protocol with an overlay for sending messages and managing responses. A message can be of four different types: confirmable, unconfirmable, acknowledgment and reset. Thus, in the header of CoAP messages, a message code or a response code is included.

Based on the analysis of these four protocols, we propose a metamodel that can be instantiated independently on each of them. The metamodel is presented in Fig. 3. In this figure, each item is represented. To include topic and partition items of Kafka, we choose to use the same concepts as the other protocols: Queue is used for partition concept. Exchange is used for topic concept.

5 Implementation

The pulse metamodel is implemented in the Moose framework [4]. This framework is a data analysis platform. It contains data import tools, modeling tools, a domain specific language allowing queries to be made on these data and a language for creating visualizations.

The implementation of pulse contains all the elements presented in the framework previously shown in Fig. 1. Four importers retrieving data from RabbitsMQ have been developed: two message importers and two others allowing the model to be synchronized with the architecture of the RabbitMQ application.

Thus, the tool is able to collect:

- Message traces from the RabbitMQ log files. These traces are read thanks to two modules: a message trace interpreter intervening on the log file a posteriori and a consumer subscribed to a message trace exchange. The latter is more reactive, but intrusive because it adds a listening node in the system.
- The history of events in the life cycle of the system's components, from the RabbitMQ Event Exchange plug-in. This information is collected through the instantiation of a consumer subscribed to a dedicated event exchange.
- The configuration of RabbitMQ, from a REST client that queries the RabbitMQ management API.

6 Conclusion

This article presents a framework for analyzing the exchanges of a distributed information system. This framework is composed of importers, allowing the collection of data from EDA systems such as AMQP, CoAP, MQTT and Kafka. It uses a generic metamodel implemented in Moose and offers analysis tools to identify information exchanges in the system. It allows to analyze these exchanges to identify potential problems or to visualize the active and inactive nodes of an IoT system. The model takes into account the dynamic aspects of the system by considering the messages and by modeling the changes in the architecture over time.

The next step is to consider the whole tooling: useful visualizations, queries and importers. Then, we work on the detection of system failures from the analysis of messages and the treatment of these failures. Another issue is to process the mass of data and to use known big data strategies for this.

References

1. Amokrane, N., Laval, J., Lanco, P., Derras, M., & Moala, N. (2018). Analysis of data exchanges, contribution to data interoperability assessment. In *International Conference on Intelligent Systems* (IS), Madeira Island, Portugal.
2. Ford, T., Colombi, J., Graham, S., & Jacques, D. (2007). The interoperability score. Technical Report, Air Force Inst of Tech Wright-Patterson AFB OH.
3. Daclin, N., Chen, D., & Vallespir, B. (2008). Methodology for enterprise interoperability. *IFAC Proceedings Volumes, 41*(2), 12873–12878.
4. da Silva, G., Leal, S., Guédria, W., & Panetto, H. (2019). Interoperability assessment: A systematic literature review. *Computers in Industry, 106*, 111–132.
5. da Silva Serapiao Leal, G. (2019). Support à la décision pour l'analyse de l'interopérabilité des systèmes dans un contexte d'entreprises en réseau. PhD thesis, Université de Lorraine
6. Mallek, S., Daclin, N., & Chapurlat, V. (2012). The application of interoperability requirement specification and verification to collaborative processes in industry. *Computers in Industry, 63*(7), 643–658.
7. Berman, D. (2019) The complete guide to the ELK Stack. Logz.io. Available at: https://logz.io/learn/complete-guide-elk-stack/. [Consulted on 07, June 2019].
8. Laval, J., Denier, S., Ducasse, S. &, Falleri, J.-R. (2011) Supporting simultaneous versions for software evolution assessment. *Science of Computer Programming, 76*(12), 1177–1193. ISSN 0167–6423, https://doi.org/10.1016/j.scico.2010.11.014.
9. MQTT—OASIS standard. [Online]. Available: http://mqtt.org/.
10. Message, O., Telemetry, Q., & Mqtt, T. (2014). "MQTT Version 3.1.1," no. October, 2014.
11. Freiknecht, J., Papp, S., Freiknecht, J., & Papp, S. (2018). Apache Kafka. In *Big Data in der Praxis*.
12. "Apache Kafka." [Online]. Available: https://kafka.apache.org/.
13. Shelby, Z., Hartke, K., & Bormann, C. (2014). The constrained application protocol (CoAP). *Internet Eng. Task Force*, 1–112
14. Hong, X. J., Sik Yang, H., & Kim, Y. H. (2018). Performance analysis of RESTful API and RabbitMQ for Microservice web application. In *9th International Conference on Information and Communication Technology Convergence: ICT Convergence Powered by Smart Intelligence, ICTC 2018*, 2018, pp. 257–259.