# Analysis of Data Exchange among Heterogeneous IoT Systems

Jannik Laval, Nawel Amokrane, Mustapha Derras, Néjib Moalla

# Analysis of Data Exchange among Heterogeneous IoT Systems

Jannik LAVAL[1] and Nawel AMOKRANE[2] and Mustapha DERRAS[2] and Néjib MOALLA[1]

[1] University Lumière Lyon 2, DISP lab EA4570, Bron, France
surname.name@univ-lyon2.fr
[2] Berger-Levrault, Lyon, France
surname.name@berger-levrault.com

**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 can't be ignored.

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 multiplicity of these data exchanges generates complexity and brings out control needs that can be addressed by establishing monitoring and analysis systems. These systems operate on several different tools to generate the pub/sub module communication messaging monitoring. For example, RabbitMQ offers a management console with information related to the structure of the messaging system and the status of messages. It is suitable for specific queries where the maintainer knows the elements that must be tracked. It presents lists of existing resources (channels, exchanges, queues, etc.), their content, characteristics and a set of statistics. It is, for example, possible to access queues and check the pending messages. However, it does not allow advanced querying and filtering over the resources and the transiting messages. Keeping track of in-transit messages is for instance not permitted as the consumed messages are no longer presented in the management console. Also, messaging canals such as exchanges, queues and their bindings are volatile and can be deleted when the consumer disconnects, as the console does not provide a visualization of the history of existing resources. It is for example not possible to identify all the consumers that a resource has had.

To supervise and monitor the underlying message broker that is used as a communication canal among the interoperable systems. 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, section 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 Section 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 Kibana 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 Figure 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.

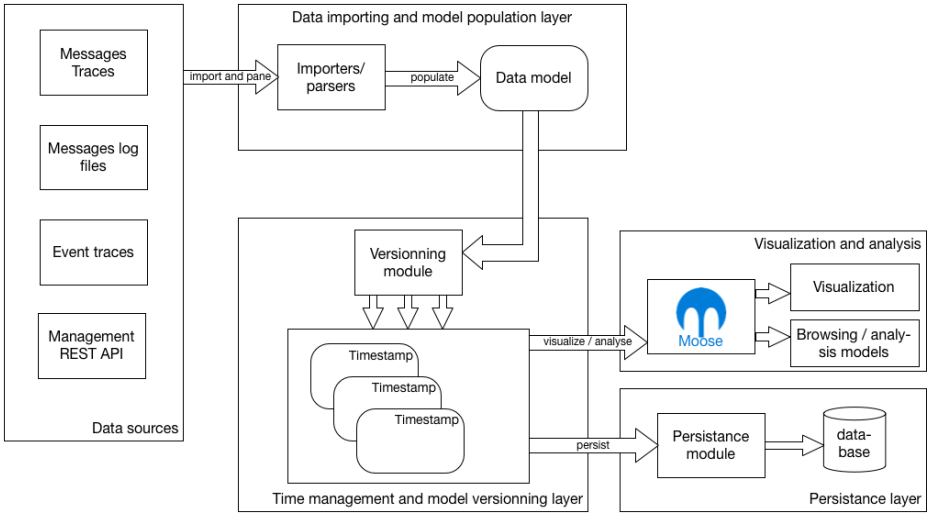


**Fig. 1.** Pulse Framework architecture

### 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 meta-model representing the structure of a distributed exchange system. The meta-model 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 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 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.

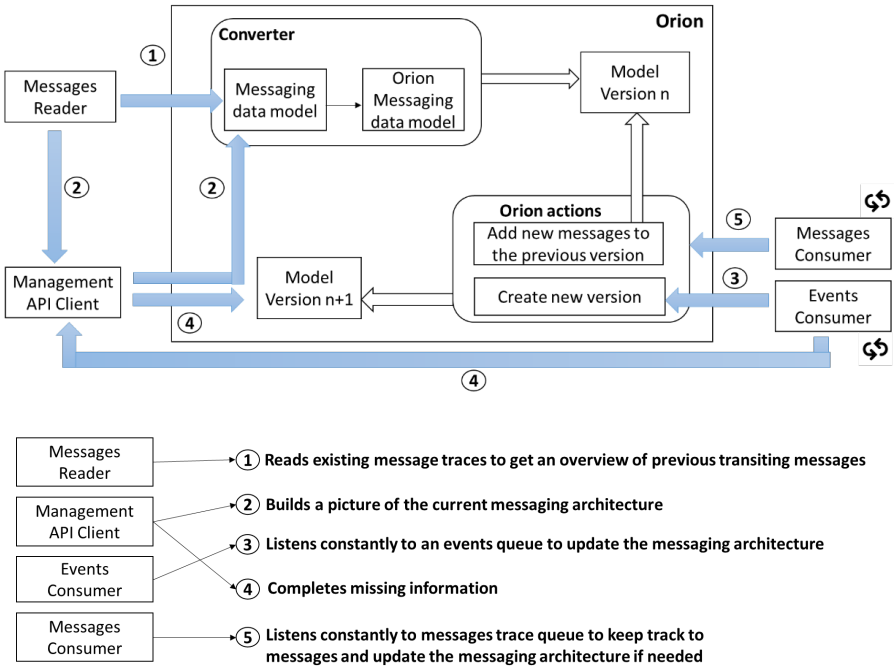


**Fig. 2.** Orion and the model versioning process

### 3.3 Persistence layer

Orion keeps different versions of the model in the Pharo1 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 Grafana2 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.

## 4 Pulse Metamodel

The goal of the Pulse Metamodel (Figure 3) is to represent three aspects of the messaging structure:

- 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**).

[1] https://pharo.org/

[2] https://grafana.com/

**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 |

- AMQP is an asynchronous message queuing protocol, aiming to create an open standard for passing messages between applications and systems regardless of internal design. AMQP enables encrypted and interoperable messaging between organizations and applications. The protocol is used in client/server messaging and in IoT device management. A message is published into an exchange by a message producer then routed to none or several queues according to routing keys that are defined via bindings. The message eventually reaches the consumers for consumption. Each message carries application data within the payload where the data is formatted according to an exchange format. The architecture publisher components and consumer components are linked to resources (exchanges and queues) through connection channels. The clients connect to the broker with user credentials, where every user has specific permissions. A node is a RabbitMQ server.
- MQTT [9, 10] is a Machine to Machine and IoT connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It can work in an equivalent way as AMQP. Each message has a payload which contains the data to transmit. The session flag tells the client whether the broker already has a persistent session available from previous interactions with the client. The pub/sub model decouples the client that sends a message (the publisher) from the client or clients that receive the messages (the subscribers). QoS gives the client the possibility to choose a level of service. There are three levels of QoS: 0,1 and 2. The service level determines what kind of guarantee a message has for reaching the intended recipient.
- Kafka [11, 12] has been optimized to stream data between systems and applications as fast as possible in a scalable manner. Its storage layer is essentially a pub/sub messaging pattern like AMQP message queue designed as a distributed transaction log, making it highly valuable for enterprise infrastructures to process streaming data. The Kafka cluster stores streams of records in categories called topics. Each record consists of a key, a value, and a timestamp. Data is divided into topics, which resemble streams of messages. Topics are multi-subscriber, are divided into partitions and each broker can possess one or more of such partitions. For each topic, the Kafka cluster maintains partitions. Each partition is an ordered, immutable sequence of records that is continually appended to a structured commit log. Producers publish

to topics and brokers store messages received from the producers. Messages are payloads of bytes that consumers use iterators to consume.

- CoAP [13] is a specialized Internet Application Protocol for constrained devices, as defined in RFC 7252. One of the main goals of CoAP is to design a generic web protocol for the specific requirements of this constrained environment, especially considering energy, building automation, and other machine-to-machine applications. There are two layers in CoAP architecture. One layer is communication exchange over UDP, and the second layer is request and response. Request and response semantics are carried in CoAP messages, which include either a Method Code or Response Code, respectively. CoAP defines four types of messages: confirmable, non-confirmable, acknowledgement, reset.

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 Figure 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 using Moose. Moose [4] is a data analysis platform that includes tools to import and parse data, model it, analyze it and visualize it all in one tool.

The implementation provides all the modules presented in the framework presented in Figure 1. To complete the implementation of the metamodel, we developed importers to populate the model. We supervise and monitor RabbitMQ by interrogating its services to collect information from several sources:

- Message traces provided by RabbitMQ tracing plug-in; they are overwritten by contextual business elements about the communicating applications characteristics provided by BL-MOM. To read these traces, two modules have been developed: a message log interpreter (*Messages Reader*) and an AMQP consumer subscribed to a message trace exchange (*Consumer Messages*).
- History of events of creation and deletion of RabbitMQ resources, provided by RabbitMQ Event Exchange plug-in. These events are collected via an AMQP consumer module subscribed to an events exchange (*Events Consumer*)
- Current configuration of the broker audited using REST client (*Management API Client*) interrogating RabbitMQ management API

## 6 Conclusion

We presented in this work a framework to analyze the data exchange based on collected messages from a network of communicating systems. The proposed system exploits services from publisher/subscriber systems as AMQP, CoAP, MQTT and Kafka. We

proposed a generic messaging metamodel used to gather and aggregate information collected from several sources. An effective framework for highlighting data exchanges helps to address their complexity. We implemented the metamodel and analysis functions by extending the metamodel of Moose taking advantage of its inherent services. We elaborated importers and parsers to collect data from different sources and populate the metamodel. The log events are first collected and parsed according to several pub/sub messaging communication protocols allowing consider the dynamic aspects of messaging configuration.

Further developments are planned: data visualizations and queries can be developed to help indicate interoperability failures. For instance, architectural evolution over time can be used to indicate idle interoperability interactions.

## References


1. N. Amokrane, J. Laval, P. Lanco, M. Derras and N. Moala, "Analysis of Data Exchanges, Contribution to Data Interoperability Assessment", International Conference on Intelligent Systems (IS), Madeira Island, Portugal, 2018.
2. Thomas Ford, John Colombi, Scott Graham, and David Jacques. The interoperability score. Technical report, AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH, 2007.
3. Nicolas Daclin, David Chen, and Bruno Vallespir. Methodology for enterprise interoperability. IFAC Proceedings Volumes, 41(2):12873–12878, 2008.
4. Gabriel da Silva Serapiao Leal, Wided Guédria, and Hervé Panetto. Interoperability assessment: A systematic literature review. Computers in Industry, 106:111–132, 2019.
5. Gabriel da Silva Serapiao Leal. Support à la décision pour l'analyse de l'interopérabilité des systèmes dans un contexte d'entreprises en réseau. PhD thesis, Université de Lorraine, 2019.
6. Sihem Mallek, Nicolas Daclin, and Vincent Chapurlat. The application of interoperability requirement specification and verification to collaborative processes in industry. Computers in industry, 63(7):643–658, 2012.
7. Daniel Berman. 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. Jannik Laval, Simon Denier, Stéphane Ducasse, Jean-Rémy Falleri. Supporting simultaneous versions for software evolution assessment, Science of Computer Programming, Volume 76, Issue 12, 2011, Pages 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. O. Message, Q. Telemetry, and T. Mqtt, "MQTT Version 3.1.1," no. October, 2014.
11. J. Freiknecht, S. Papp, J. Freiknecht, and S. Papp, "Apache Kafka," in *Big Data in der Praxis*, 2018.
12. "Apache Kafka." [Online]. Available: https://kafka.apache.org/.
13. Z.Shelby K.Hartke C.Bormann, "The Constrained Application Protocol (CoAP)," *Internet Eng. Task Force*, pp. 1–112.
14. X. J. Hong, H. Sik Yang, and Y. H. Kim, "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.

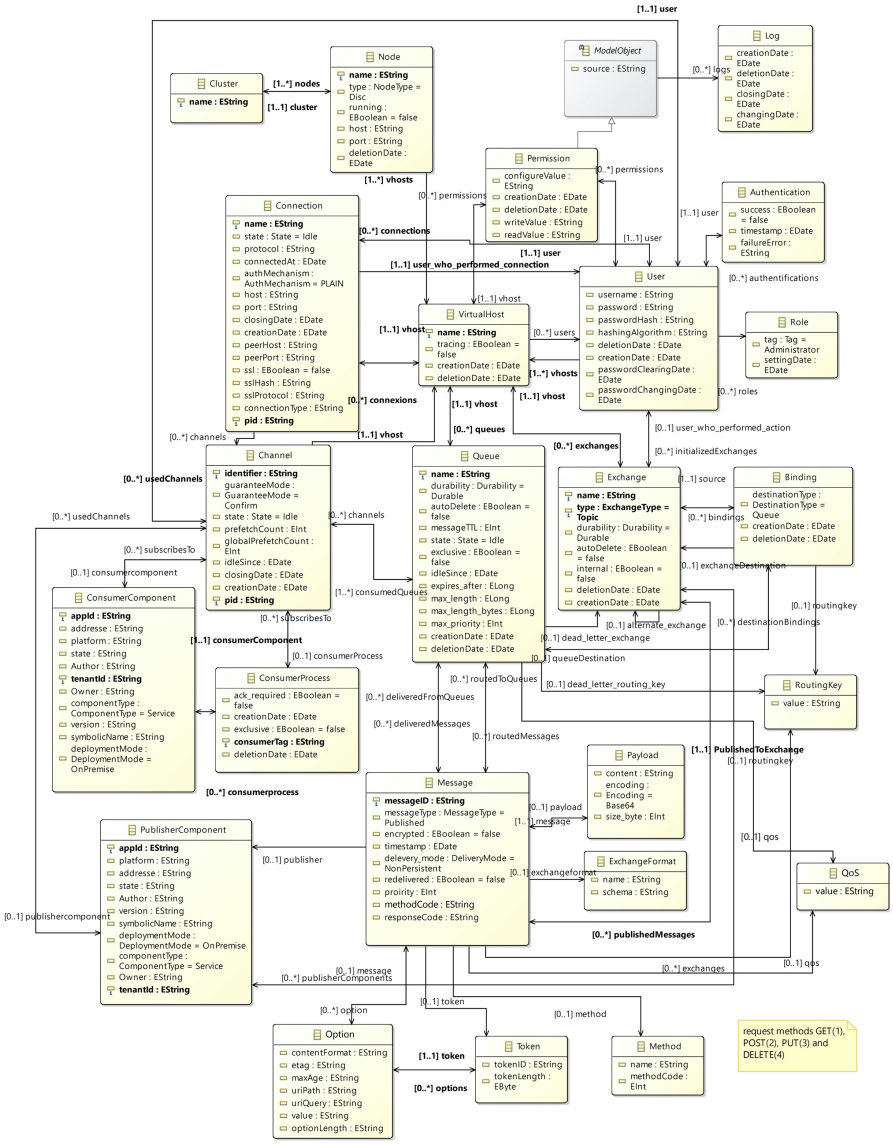


**Fig. 3.** The Pulse Metamodel