



Analysis of Information Exchanges among Distributed IS, an Industrial Case

Nawel Amokrane, Jannik Laval, Philippe Lanco, Mustapha Derras, Néjib Moalla

► To cite this version:

Nawel Amokrane, Jannik Laval, Philippe Lanco, Mustapha Derras, Néjib Moalla. Analysis of Information Exchanges among Distributed IS, an Industrial Case. International Workshop on Smalltalk Technologies, Sep 2018, Cagliari (Sardaigne), Italy. <hal-04680795>

HAL Id: hal-04680795

<https://hal.science/hal-04680795v1>

Submitted on 29 Aug 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



HAL Authorization

Analysis of Information Exchanges among Distributed IS, an Industrial Case

Nawel Amokrane

Berger-Levrault, France
nawel.amokrane@berger-levrault.com

Jannik Laval

University of Lyon, University
Lumière Lyon 2, DISP lab EA4570
BRON France
jannik.laval@univ-lyon2.fr

Philippe Lanco

Berger-Levrault, France
philippe.lanco@berger-levrault.com

Mustapha Derras

Berger-Levrault, France
mustapha.derras@berger-levrault.com

Nejib Moalla

University of Lyon, University Lumière Lyon 2,
DISP lab EA4570
BRON France
nejib.moalla@univ-lyon2.fr

Abstract

The multiplicity of information exchanges among distributed Information Systems (IS) generates complexity and brings out control and analysis needs that can be handled by establishing monitoring systems. We present in this paper a work in progress where we exploit services provided by RabbitMQ, a messaging based communication mean, in order to collect information about IS architectures and their interactions. We propose a Messaging Monitoring Metamodel that structures and aggregates these collected information. It provides a single point of control and enables depicting high level information about messaging aspects of existing distributed IS. This paper also showcases the use of Moose, a software analysis platform, to implement monitoring services. The main component of this implementation is extending Moose metamodel with the proposed metamodel in order to perform querying and data visualization.

Keywords Data analysis, Data Visualization, Monitoring, Message Brokers

1. Introduction

Once Information Systems (IS) are put into production, they are mostly part of distributed networks that require shar-

ing and exchanging information. These interactions generate complexity and bring out control needs that are beyond the applications business scope, where it is important to supervise the network's interactions in order to anticipate, or react promptly to, potential dysfunctions. In this context, Berger-Levrault (BL)¹ implements data exchange within its IS ecosystems through the use of flexible, scalable and loosely-coupled architectures, such as service oriented (Erl 2005) and event driven (Michelson 2006) architectures (SOA, EDA). These architectures rely on several communication means to convey data among the network, in this article we focus on messaging mechanisms (Hohpe and Woolf 2004). The latter allows sending messages from a source (*publisher*) to one or more recipients (*consumers*) by using specific routings.

Aiming to fully harness its integration architectures, BL-MOM (Message Oriented Middleware), an in-house framework, has been developed by Berger-Levrault as a first step towards building a robust interoperability framework that handles scalable and configurable architectures. It establishes routes between communicating applications following, over a first phase, the AMQP protocol (Vinoski 2006). The exchange of messages is handled according to publish/subscribe or RPC (Remote Procedure Call) patterns (Hohpe and Woolf 2004), allowing consequently the applications to be loosely coupled. BL-MOM uses RabbitMQ², a reliable open source communication mediator, as the underlying message broker and provides helpers to facilitate

¹ Software provider specialized in the fields of education, health, sanitary, social and territorial management

² <https://www.rabbitmq.com/>

creating messages schema, publishers, consumers and messaging operations over this broker.

BL-MOM implements configurations that guarantee service availability, delivery and persistence of messages. However, it lacks of means of control over the undertaken message interactions in order to mitigate dysfunctions and maintain the existing exchange architectures. In this work, we propose to utilize data visualization and querying implemented with Moose (Ducasse et al. 2005), a Smalltalk based open source software and data analysis platform, in order to provide a single point of control by monitoring the architectures configuration and the messaging behavior.

In the remaining sections of this paper, Section 2 defines terms used in this paper. We explain the motivation behind this work in Section 3. Then, we present in Section 4 the architecture of the proposed monitoring system, its underlying metamodel and a subset of the reported queries and indicators. Section 5 describes implementation elements. Section 6 depicts some resulting data visualizations illustrated by an industrial case study. Section 7 concludes this paper and opens perspectives.

2. Terms and Vocabulary

In this paper, we use some specific terms related to IS Messaging and RabbitMQ implementation of the AMQP protocol (Vinoski 2006). We define them in this section.

- **Connection:** a TCP network connection between an application and the RabbitMQ broker
- **Channel:** a stream of communications between two AMQP peers
- **Message:** a message is composed of a header and a body. The header contains the properties of the message presented in a specific type of format. The body or payload is the transiting application data also presented in a specific type of format
- **Exchange:** a named entity that receives messages from producers and routes them to queues
- **Queue:** a named entity that holds messages and delivers them to consumers
- **Routing key:** a virtual address that an exchange may use to route messages towards queues
- **Publisher:** a client application that publishes messages to exchanges
- **Consumer:** a client application that requests messages from queues

The links between some of these elements are depicted in Figure 1.

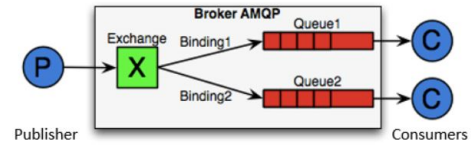


Figure 1. AMQP exchange and queuing system (Vinoski 2006)

3. Motivation

Existing open source and commercial integration frameworks (Apache Camel ^{3,4}, NServiceBus ⁵) do provide monitoring consoles. However they mostly focus on low level monitoring information such as, frequency of messages, performance indicators or memory usage. These frameworks only provide partial solutions considering Berger-Levrault monitoring and analysis needs.

RabbitMQ offers a management console with information related to the structure of the messaging system and the status of messages (Dossot 2014). It presents lists of existing resources (channels, exchanges, queues...), their content, characteristics and a set of statistics. It is, for example, possible to access queues and check the pending messages.

Based on our experience, RabbitMQ console can be used for real time monitoring and is suitable for specific queries where the maintainer knows the queues or exchanges that must be tracked. However, it does not allow advanced querying and filtering over the resources and the transiting messages, especially needed in case of multiple exchanges between IS with thousands of messages. Keeping track of in-transit messages is for instance not permitted as the consumed messages are no longer presented in the management console. Besides, 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.

We advocate that the behavior of data exchange among communicating applications gives an indication of their data interoperability level. A monitoring system should provide elements to help maintain a good level of interoperability based on interoperability requirements (Mallek et al. 2010). Though the lack of the above mentioned control elements complicates the analysis and diagnosis of interoperability dysfunctions such as the inactivity of publishers and consumers, the invalidity of exchange formats or the unavailability of data. Hence it is difficult for maintainers to identify the context and the origin of the problem, based only on RabbitMQ management console.

³<http://sksamuel.github.io/camelwatch/>

⁴<http://rhq-project.github.io/rhq/>

⁵<https://particular.net/nservicebus>

We therefore propose to take advantage of other RabbitMQ services such as messages and events tracing, and combine those with information provided by BL-MOM, in order to perform advanced monitoring and querying. This would provide indicators allowing the determination of maintenance actions.

4. Messaging Monitoring System

We supervise the existing messaging interactions by interrogating RabbitMQ, the underlying message broker and its provided log services. And in order to provide high level indicators that facilitate interactions maintenance, we aggregate the collected information into a common metamodel, the Messaging Monitoring Metamodel. In the following, we present the metamodel elements and detail how these elements are collected and what monitoring information can be reported.

4.1 Messaging Monitoring Metamodel

The Messaging Monitoring Metamodel (depicted in a simplified version in Figure 2) illustrates the messaging structure implemented through message queuing and exchange system. Each message carries application data within the payload where the data is formatted according to an exchange format. It is published into an exchange then routed to none or several queues according to routing keys that are defined via bindings between the exchange and the queues. The architecture components represent publishers or consumers, they are linked to resources (exchanges and queues) through connection channels. The publisher and consumer clients connect to the broker with user credentials, where every user has specific permissions.

A node is a RabbitMQ server, establishing several nodes can be used to handle large scale, geographically distributed architectures. RabbitMQ can also function in a cluster mode, load balancing one broker instance over several nodes. Within a node, RabbitMQ separates groups of resources with virtual hosts. The latter provides logical grouping and isolation of resources that also share a common authentication.

4.2 Messaging Monitoring Queries and Indicators

The Metamodel aggregates information from several sources. We present in the following each source and the information it conveys:

- Message traces provided by RabbitMQ tracing plug-in⁶, this allows to identify for each message:
 - The RabbitMQ node it transits through, along with the connection and virtual host information
 - The exchange it was published in or consumed from, the queues it is routed to or the queue it is consumed from and the related routing keys

- The user publishing or consuming the message
- Its timestamp, type (published / received) and delivery mode (persistent or not)
- Current configuration of the broker audited through the use of RabbitMQ REST management API⁷.
- History of events of creation and deletion of resources, virtual hosts, users and permissions; creation and closing of connections and channels and user authentication attempts. This is provided by the RabbitMQ Event Exchange plug-in⁸.
- Contextual elements about the communicating applications characteristics provided by BL-MOM.

This conjunction of sources allows us to have a single point of control and perform advanced monitoring to facilitate maintenance actions. Here is a subset of the queries and indicators that can now be reported:

Business level queries: BL-MOM overrides message traces providing high level information i.e. information with identifiable business signification, such as: publishers and consumers application identifiers, tenants, topics and exchange formats. This enables, for instance, to consider the tenant identifier of the application that publishes or consumes messages in case of a multi-tenant application. Similarly, topics to which messages have been published when using a publish/subscribe messaging pattern can be specified. Such information help maintainers specify the context of the interactions.

Messages filtering: messages can be filtered according to, or combining, several of their characteristics such as: identifier, timestamp, exchange, queue, publisher, consumer, related user, state, size, encoding or exchange format. This allows to have more precision while searching in messages traces. For example, filtering messages that transit within a time slot between a publisher and a consumer allows to inspect the behavior of the interaction during a period where a break down occurs. We can also check if a message is duplicated, redelivered or rejected.

Security checks: supervising the security of message exchanges can be improved by checking several elements:

- User authentication timestamps and the success or failure of the authentication. This contributes to the detection of violation attempts.
- The set of users, and their corresponding consumers which have similar permissions on a set of resources, in order to prevent potential data leaks.
- Payload content, when it is not confidential, to check whether secret elements are encrypted or not.

⁶<https://www.rabbitmq.com/firehose.html>

⁷<https://pulse.mozilla.org/api/>

⁸<https://www.rabbitmq.com/event-exchange.html>

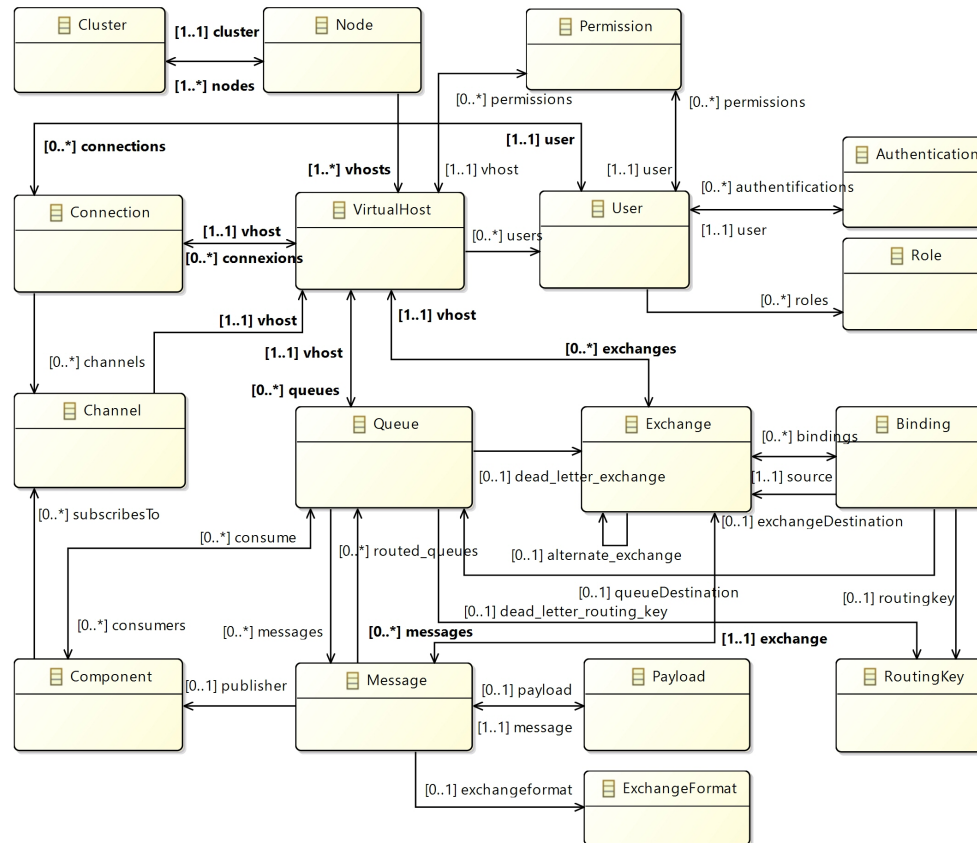


Figure 2. Messaging Monitoring Metamodel

Architecture evolution: the ability to track resources creation and deletion allows to depict current and past RabbitMQ messaging configurations in terms of exchanges, queues and their bindings. Furthermore, relying on business information provided by BL-MOM, we can identify the communicating applications even if their technical identifiers have changed at the broker level. We can accordingly provide a larger scope visualization of the architecture components and their interactions along with their evolution over time.

Interoperability indicators: the behavior of data exchange among communicating applications gives an indication of their data interoperability level. If, for example, a resource has been inactive over a certain period of time, this can indicate the unnecessary or the obsolescence of the interaction. It can furthermore indicate a configuration change (consumption on another channel) or a process change at the application level. We present in Table 1 how we can highlight some data interoperability problems by contextualized indicators or possible queries over the proposed metamodel. We also indicate potential causes and point out existing correlations between interoperability problems.

5. Implementation

We implemented a prototype tool as a first step towards developing the proposed monitoring system. It allows to visualize some of the behaviors of Information System interactions. This prototype aims to allow maintainers to understand the status of each message and gives indications to help analyze dysfunctions that may occur in the messaging system. The prototype is implemented on top of Moose, a software analysis platform (Ducasse et al. 2005), and is based on an extension of its metamodel. The latter is independent from source code (Ducasse et al. 2011). Moose provides importers for several languages, such as Java, C#, C++ and Smalltalk. We experimented our approach on data generated by a RabbitMQ implementation.

The implementation of the prototype is composed of two parts: the implementation of the previously presented Messaging Monitoring Metamodel and the development of two visualizations to show the feasibility and the interest of the proposed system.

5.1 Implementation of the Metamodel

The implementation of the metamodel is an extension of Moose. The latter also includes a family of metamodels that can be customized for various aspects of code representation

Interoperability problem	Indicators	Potential causes
Inactivity of publisher or consumers	<ul style="list-style-type: none"> - Date of last published message on the concerned topic exchange - Date of last consumed messages on the concerned queues - Intervals between publications for inactivity periods of publishers - Intervals between connections for inactivity periods of consumers 	<ul style="list-style-type: none"> - Unnecessity / obsolescence of the interaction due to architecture configuration change or a process change - Deleted user or not communicated credentials change - Application shutdown
Invalid exchange format	<ul style="list-style-type: none"> - Rejected messages on the concerned queues - Exchange format of last accepted message on the concerned queues 	<ul style="list-style-type: none"> - Incompatible exchange formats - Evolution of the exchange format not communicated
Missing interaction	<ul style="list-style-type: none"> - Presence of the concerned topic exchange - Presence of the consumer - Date of last connection of the consumer 	<ul style="list-style-type: none"> - Error in publishing or consumption properties attribution - Problem of inactivity of a publisher - Problem of inactivity of a consumer - Problem of invalid exchange format
Unavailable expected data	<ul style="list-style-type: none"> - Lost messages on the concerned canals (filtering according to message known elements) 	<ul style="list-style-type: none"> - Problem of missing interaction
Data leak	<ul style="list-style-type: none"> - Presence of illegal consumers - Authentication attempts 	<ul style="list-style-type: none"> - Credential leak or error in credential attribution to consumers - Error in consumption properties attribution

Table 1. Interoperability problems their indicators and related causes

(static, dynamic, history,...). Moose core describes the static structure of software systems, particularly object-oriented software systems⁹. Extending Moose is a major asset for our implementation approach, as it allows us to rely on and reuse powerful tools and analysis developed within Moose (Ducasse et al. 2009).

To complete the implementation of the metamodel, we need to develop importers to populate the model. For that, the goal is to create 2 types of importers that catch information from different sources (as explained in Section 4.2): parsers for messages traces files and events log files and REST requests to interrogate RabbitMQ management API. Currently, only the traces file parser and importer are developed. Consequently, the case study focuses on those files only.

5.2 Using Moose as a Platform

Moose is used in our implementation as a complete platform. It provides the possibility to build parsers, data queries, data browsers and data visualizations by extending inherent services.

This way, Moose allows us to bring an agile answer to Berger-Levrault industrial needs by completely integrating the monitoring metamodel into its environment. We seamlessly benefit from the provided services for data browsing

and visualization. Once our metamodel is implemented and declared, we can for instance browse data with the generic browser. Further developments are needed to implement the required parsers, queries and data visualization.

Figures 3 and 4 presented in the following section show two examples of visualizations built with Moose.

6. Case Study

We use here a case study of exiting interactions among BL applications to showcase some monitoring services of the implemented prototype. Berger-Levrault provides its clients with a software as a service Console (SAAS-Console) to allow a secure access to its cloud deployed software via SSO (Single-Sign-On) mechanism. The SAAS-Console is also an administration console that allows the clients to autonomously manage the accounts and access rights related to the software they use.

The console exchanges data with several applications (a Customer Relationship Management application, an authentication module and business applications). We consider here the one that uses RabbitMQ for messages transmission: interaction with BL-Socle for the automatization of provisioning, to which the SAAS-Console sends information regarding the packages of software to be deployed for the clients and their assigned user access accounts.

The interactions are set up with BL-MOM and the exchanged messages transit through a RabbitMQ node.

⁹ see (Demeyer et al. 2001) and <http://www.moosetechnology.org/docs/famix>

6.1 Goals

In this primary case study, we provide monitoring elements in response to some of the needs expressed by Berger-Levrault:

- Having a clear visualization of the traces of transiting messages in each queue and their characteristics. This is particularly needed considering that transiting messages are volatile in RabbitMQ console.
- Having a global vision of the structure of messaging architectures and the undertaken message paths. This is to ensure the presence and activity of publishers and consumer and the expected interactions between them.
- Ensuring the correctness of the subscriptions and that there are no data leaks.

6.2 Used Data

Berger-Levrault provided traces files from the RabbitMQ node used for the above explained data exchange between the SAAS-Console and BL-Socle. BL-MOM controls the messaging actions allowing to have logs with business level information. These log files are activated for elements that we want to trace on the broker node. The log files are JSON encoded, containing a JSON entity for each action on messages. In this case study, we exploit only these log files without extracting information from the other sources (e.g. REST API). For the sake of confidentiality, we do not trace the payload of the messages which contains the transiting application data.

6.3 Results

For the first goal we propose to represent the messages, collected from the traces, in a browsable, histogram typed, visualization. It depicts the messages routed to a specified queue, during a chosen period and grouped by equal slots of time. Figure 3 shows the resulting visualization for published messages into one of the queues that the SAAS-Console messages are routed to and that is consumed by BL-Socle. The queue is bound to a topic exchange that handles messages related to user access accounts and the visualization shows the rate of published messages traced during working hours. We can see that there have been more activity on the SAAS-Console regarding the creation and updating of user accounts and access rights during the end of the day. Moreover, In order to inspect the messages, we upgraded this visualization (that is also provided by RabbitMQ console but in a static way), by offering the possibility to browse into each entry to view the list of messages and deploy accordingly each message enabling, this way, to check its definition and characteristics.

In answer to the other expressed needs, we provide, based on traces files, a cartography that offers an overall view of the current messaging architecture components. The visualization exposes the internal broker architecture, in terms of

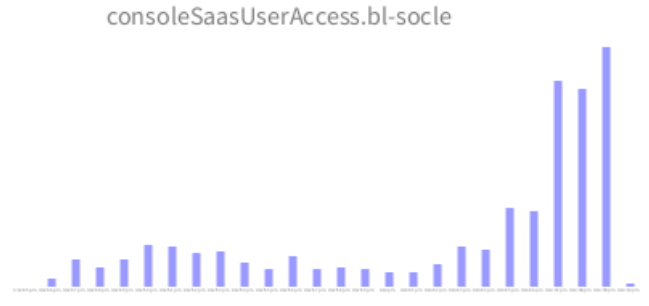


Figure 3. Visualization of transiting messages in a queue

exchanges to which the messages are published and routed queues, along with business information by identifying the producers and consumers of the messages. An example is presented in Figure 4. The visualization succeeds in showing the exiting interaction between the SAAS-Console as the publisher and BL-Socle as the consumer. It displays the underlying RabbitMQ architecture elements:

- The use of three topic exchanges (consoleSaasApplicationRole, consoleSaasClientContract and consoleSaasUserAccess)
- The queues to which the exchanges are bound (SaasApplicationRole.bl-socle, SaasClientContract.bl-socle, SaasUserAccess.bl-socle, SaasUserAccess.logger)

We notice that user access related exchange (consoleSaasUserAccess) is also bound to a queue consumed by a logger application.

Such visualization offers a structured representation of a technical architecture that can be limited by business context. It favors the inspection of the existing interactions among publishers and consumers and whether the latter correspond to the intended designed ones. It also allows to detect unauthorized consumers.

The activity of publishers and consumers can be inspected by checking their last publication and consumption. Furthermore, in order to show the evolution of the architecture we need to collect and integrate data from event logs to reconstruct the architecture at a selected point in time to help prevent data leak.

7. Conclusion and Future Work

We presented in this paper the development of data visualization and the querying over RabbitMQ based message exchanges. The aim is to provide analysis and monitoring services to control interactions among Berger-Levrault communicating information systems. The proposed monitoring system relies on the definition of a common metamodel that combines information collected from different RabbitMQ services. The metamodel also takes into account business level information allowing to provide high level monitoring indicators.

We implemented the metamodel by extending Moose metamodel taking advantage of the inherent agility. We elab-

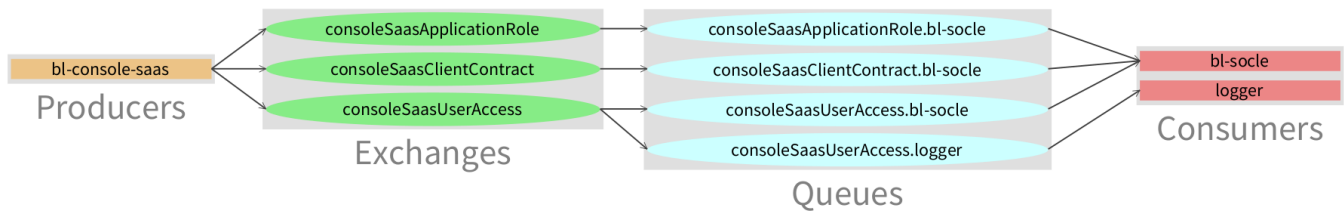


Figure 4. Cartography of publishers and subscribers of traced interactions

orated an importer and a parser for traces data to populate the metamodel. Once structured by the metamodel, we used the data to build two data visualizations in answer to some of Berger-Levrault motoring needs. Further developments are planned: (i) collecting and parsing events logs in order to consider dynamic aspects of the messaging configurations, for this we propose to extend the AMQP Smalltalk client library to consume event messages provided by the event exchange plug-in, (ii) REST requests to interrogate RabbitMQ management API in order to fully populate the proposed metamodel. In addition, other data visualizations and queries can be developed to help indicate messaging dysfunctions. A representation of messages by type can be proposed to visualize rejected messages. The evolution of the architecture over time can as well be used to highlight idle architecture components. Indicators can also be utilized to alert administrators with notifications about failed operations or about rates that exceed accepted thresholds.

References

- S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.
- D. Dossot. *RabbitMQ essentials*. Packt Publishing Ltd, 2014.
- S. Ducasse, T. Gırba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005. ISBN 88-464-6396-X. URL <http://scg.unibe.ch/archive/papers/Duca05aMooseBookChapter.pdf>.
- S. Ducasse, T. Gırba, A. Kuhn, and L. Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 8(1):5–19, Feb. 2009. doi: 10.1007/s10270-008-0081-4. URL <http://scg.unibe.ch/archive/drafts/Duca08a-Sosym-ExecutableMetaLanguage.pdf>.
- S. Ducasse, N. Anquetil, M. U. Bhatti, A. Cavalcante Hora, J. Laval, and T. Girba. MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family. Research report, Nov. 2011. URL <https://hal.inria.fr/hal-00646884>.
- T. Erl. *Service-oriented architecture*, volume 8. Prentice hall New York, 2005.
- G. Hohpe and B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.
- S. Mallek, N. Daclin, and V. Chapurlat. Towards a conceptualisation of interoperability requirements. In *Enterprise Interoperability IV*, pages 439–448. Springer, 2010.
- B. M. Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2, 2006.
- S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6), 2006.