R-MOM: A Component-Based Framework for Interoperable and Adaptive Asynchronous Middleware Systems

Jonathan Labéjof, Antoine Léger, Philippe Merle, Lionel Seinturier, Hugues Vincent

R-MOM: A Component-Based Framework for Interoperable and Adaptive Asynchronous Middleware Systems

Jonathan Labéjof*†, Antoine Léger*, Philippe Merle†, Lionel Seinturier†‡ and Hugues Vincent*
*THALES COMMUNICATIONS & SECURITY
Massy, France
Email: firstname.lastname@thalesgroup.com
†Univ. Lille 1 & Inria
LIFL UMR CNRS 8022
Villeneuve d'Ascq, France
Email: firstname.lastname@inria.fr
‡IUF

***Abstract*—Systems of systems (SoS) are composed of sub-systems such as Distributed, Information Technology, Real-Time and Embedded systems. Among distributed systems, Message-Oriented Middleware (MOM) is used by SoS in order to share status information from system elements (component, service, etc.). Often several different MOM technologies are used in one SoS, then interoperability between these MOM is a requirement.**

In this paper, we present R-MOM, a component-based framework for interoperable and adaptive asynchronous middleware systems.

R-MOM provides a reflective component architecture where one MOM functionality is embedded into one component which is modifiable at run-time. Loosely-coupling between reflective components permits to get a fined-personalization of MOM functionalities, such as protocol, encoding rule, Quality of Services (QoS) processing, data production/consumption, description, routing and filtering. Interoperability between integrated protocol functionalities is a consequence of architecture design.

R-MOM interoperates with different kinds of MOM, from distributed message queues (Java Message Service, Advanced Message Queueing Protocol, 0MQ) to content-based publish/subscribe systems (OMG's Data Distribution Service). This paper describes the architectural concepts of the R-MOM framework, discusses its implementation, and evaluates its interoperability capability.

***Keywords*-Adaptability, Distributed systems, Asynchronous communication, Message Oriented Middleware (MOM), Reflective Component Model, Reconfigurability, Interoperability.**

I. Introduction

Asynchronous communication paradigms are widely used by distributed systems as a solution for loosely coupling software entities. Loose coupling brings several interesting properties such as flexibility and ability to take into account new system and application requirements.

Among asynchronous communication paradigms, Message-Oriented Middleware (MOM) is commonly used in distributed systems. It consists to transmit a message (or data with context information such as sending date or routing property) from a producer to one or many consumers. The producer specifies logical (topic) or physical (queue) targets from where consumers are notified when data is available (push mode) or request data (pull mode). This communication model is flexible since producers and consumers are independent from each others.

Actually, a plethora of MOM exists. Differences come from the functionalities which are specific to the application domains they target (e.g. message distribution, event-driven solutions [24], [20], [10]). For example, OMG's Data Distribution Service (DDS) [16] specifies a publish/subscribe distribution model with Topics and is dedicated to Real-Time and Embedded systems (RT-E). Java Message Service (JMS) [7] specifies both logical and physical targets. But the main differences come from API (creation of producers and consumers) and quality of services (QoS). For example, DDS provides twenty one QoS parameters to configure producers and consumers. JMS, for its part, defines ten QoS parameters dedicated to IT systems with six of them associated with messages and four with producers.

The difference between implementations becomes an issue when a system uses different ones in order to exchange common structured data. For example, Systems of systems (SoS) are composed of sub-systems such as IT, RT-E and distributed systems. SoS require MOM platforms from those sub-systems in order to exchange system elements status information. Therefore, SoS have to ensure data value from all used different MOM. Even if most MOM implement same functionalities, their protocols are different, and they are not interoperable amongst each other. That's why interoperability is a requirement in such global systems.

Some MOM solutions address interoperability at API level in describing a specification related to the domain (JMS for IT systems, DDSI/DDS for RT-E systems [18]), with limitations about QoS. AMQP specification [3] provides interoperability at the protocol level, in order to keep safe messages content whatever the APIs are, but does not ensure QoS processing. In addition much MOM properties such as message description

and encoding rules are static. Furthermore, AMQP does not provide an API where MOM functionalities are flexible, and so, is not adaptive to new MOM requirements.

In this paper, we present R-MOM, a component-based framework for interoperable and adaptive asynchronous middleware systems.

Our approach provides the smallest architecture in order to ease its learning and makes interoperability between asynchronous middleware system functionalities. The R-MOM architecture is composed of six families of components corresponding to six MOM functionalities expected by MOM users, such as message production/consumption/sending/reception, data serialization/encoding, routing, description, filtering and QoS processing. Those components are declined to non-functional and binding components. An Envelope inspired from the AMQP Envelope, with lighter and more flexible capabilities, is transported between components and over the network, with information related to messages, encoding and non-functional properties. R-MOM makes interoperability between ten asynchronous communications technologies, and simplifies the configuration and execution phases in focusing on sending and reception of data.

Furthermore, we present a use case using UDP, DDS, JMS and AMQP in a SoS. In order to compare R-MOM with usual architectures, we design the SoS architecture with and without R-MOM. Both SoS architectures are equivalent at the level of components, and show that R-MOM is portable in legacy systems. At run-time, R-MOM is currently the best solution to address adaptation through four points:

1) *R-MOM permits to personalize MOM functionalities independently from each others.*
2) R-MOM saves context information whatever the nature of adaptation tasks to apply on asynchronous communication components.
3) R-MOM avoids system unavailability time during asynchronous communication changes.
4) R-MOM execution time is negligible compared to existing solutions.

This paper is divided into seven sections. Section II describes the R-MOM architecture. Section III deals with R-MOM bindings to three concrete MOM solutions. Section IV presents an implementation of R-MOM with the FraSCAti platform [21]. Section V evaluates a concrete use case scenario in comparing architecture choices and execution times, between R-MOM and existing MOMs. Section VI compares R-MOM with related works about interoperability. Section VII summarizes the contribution and discusses about perspectives.

II. The R-MOM Architecture

R-MOM provides a component-based framework for interoperability between asynchronous distributed system functionalities, i.e. data, functional and non-functional interoperability.

Interoperability has a lot of definitions, depending on application business and concerned abstraction levels.

For example, James A. O'Brien and George M. Marakas give this definition: *"Being able to accomplish end-user applications using different types of computer systems, operating systems, and application software, interconnected by different types of local and wide area"* [14].

In this paper, we focus on interconnection with asynchronous middleware paradigms. Therefore, the interoperability we target consists in translating one communication technology to another one. Simple and smart adapters [25] exist for that, respectively understood as direct and indirect transformations, and both with strengths and weaknesses.

A direct transformation or simple adapter consists in converting one communication operation call to another one. Because the call is specific, request processing time is the fastest. However this solution is not adaptable to other communication framework even if it is based on the same communication paradigm. Therefore, if both technologies have to be changed, all communication paradigm information context will be lost at the operational level.

An indirect transformation or smart adapter consists in using an intermediate common language, based on the model communication paradigm. This approach requires to specify a communication standard able to handle all communication paradigm capabilities, and to perform optionally additional processings as specific quality of services. This smart adapter should not be replaced during the system lifespan, therefore it must be as dynamic and reconfigurable as possible. Resulting request processing time is much longer than the direct transformation, and the memory footprint is more important because it imposes to use the common language as a third technology and simple adapters to communicate with, instead of one simple adapter.

Both simple and smart adapters have strengths and weaknesses, therefore their use depend on system requirements. Real-Time and Embedded systems (RTE [11]) (such as sensors networks [22], [8]) aim to minimize memory consumption and to improve efficiency, whereas IT systems wish to process a large amount of received data. Therefore, simple adapters are commonly used in system nodes solicited only for data sending, and smart adapters are commonly used in system nodes which are communication intersections, i.e. both data sender and receiver (such as in peer-to-peer architectures [19]).

We propose in this paper a smart adapter for interoperability of message content and context (description, QoS). We identify a MOM as the set of three logic parts; *(i)* the architecture based on the MOM paradigm extended with specific features, *(ii)* the message or the data interesting the application (same for all MOMs paradigm) and *(iii)* the message context containing non-functional properties related to the message. These parts help us to define the common features shared between MOMs. The result is the architecture of MOM functionalities but also messages and their contexts.

In the remainder of this section, we present our interoperable architecture with MOMs, in two parts. First, *(i)* with system nodes architecture as a set of components in charge of processing Envelopes (see subsection II-A). Then *(ii)* and *(iii)* with the concept of the Envelope which is responsible for sending message information at the transport level (see subsection II-D).

Figure 1 represents the interfaces related to the R-MOM API, and is described in the remaining of this section.

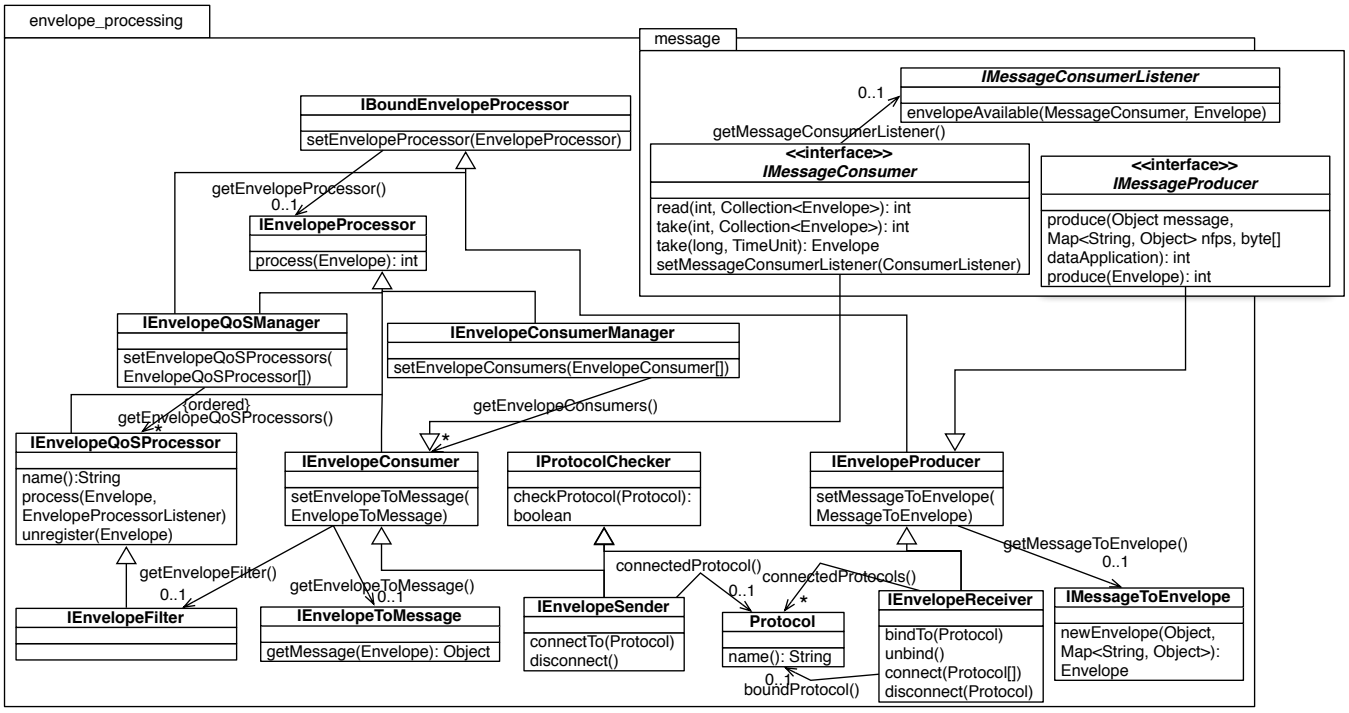

Fig. 1. **API of R-MOM components** – UML class diagram

A. Core processing architecture

An Envelope is a generic container for message value and context, and is processed by a set of components which represent a composition of MOM functionalities, and named a R-MOM core node.

1) R-MOM concepts: We design the R-MOM architecture into analyzing six MOM capabilities and in providing an adaptive version for each one by using one reflective component per functionality. Coupling between functionalities depends on component bindings, and permits to get as much flexibility as possible.

In a view where a system is considered through its functionalities, if all functionalities respect one property P, then the whole system also respects P. Therefore, if R-MOM corresponds to an assembly of adaptive functionalities, then R-MOM is adaptive too. Now, let's see how to have such an assembling of MOM functionalities, and such an adaptive property for all these functionalities.

Figure 2 represents an UML2.x class diagram of R-MOM components with respective MOM capabilities. Their descriptions follow:

- (P) The message production/consumption protocol is in charge of defining a policy about the means to exchange messages over the network or between applications.
- (D) The message description identifies the message structure.
- (C) The message transformation is in charge to (de)serialize a message depending on (D) and (P).
- (Q) The Quality of services processing process all *Envelope* non-functional properties.
- (F) The Message content filters that we assimilate to (Q) in our architecture view but specific to consumption tasks.
- (B) Message distribution is responsible for message routing policy.

A such architecture simplifies (re-)configuration in focusing on functionalities to use. For example, several specialists can apply their expertise on a R-MOM system without impacting other specialists works. For example, message transformation does not impact message distribution.

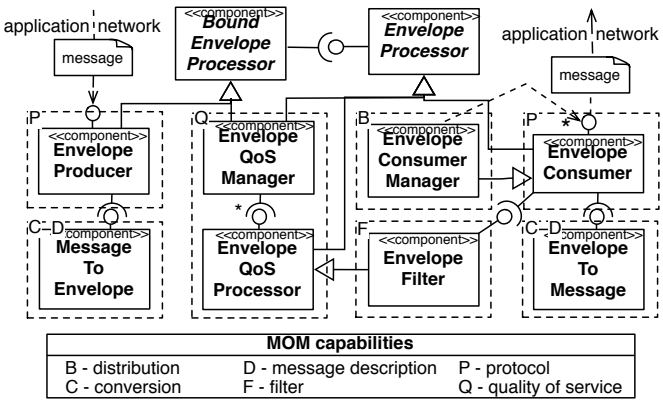

Fig. 2. **R-MOM concepts** - UML component diagram

In Figure 2, we represent a R-MOM core node. All plain line boxes are components with only one provided interface in order to simplify the understanding of the architecture. Interfaces are defined in Figure 1. Interface names start with *I*

and are implemented by the corresponding component (for example, an *EnvelopeProducer* component implements the *IEnvelopeProducer* interface). Relationships have the same semantics, than in a UML2.x component diagram except for dotted line relationships which denote a message reception from an application or the network. The main idea is to process an Envelope into a R-MOM node, which is a composition of R-MOM components. Two main parts concern *BoundEnvelopeProcessor* and *EnvelopeProcessor* components. Both components are the minimum required by R-MOM to process data.

Additional components are for binding component related to the communication technology the system needs to interoperate with, or the quality of service, which needs to be enforced.

We now detail the way *Envelope* production, QoS processing and consumption work.

2) Envelope production: An *EnvelopeProducer* gets messages from an application (thanks to a *IMessageProducer* interface, see subsection II-B) or from the network (thanks to a binding component, see section III). It requires at most one *MessageToEnvelope* in charge to get an *Envelope* related to message information (value, QoS and application data), and requires an *EnvelopeProcessor*, among one *EnvelopeQoSManager* or one *EnvelopeConsumer*.

3) QoS processing: An *EnvelopeQoSManager* implements both *IBoundEnvelopeProcessor* and *IEnvelopeProcessor* interfaces. It is in charge to process message QoS thanks to many referenced *EnvelopeQoSProcessor*s bound via a plug-in design pattern. When it receives an *Envelope*, it resolves the QoS to process. In the most dynamical case (if QoS discovery is required) *EnvelopeQoSProcessor* names are used to find which components are able to process QoS.

An *EnvelopeQoSProcessor* called to process a QoS has to get the value, deserializes it if necessary, then updates the value and serializes it before put it in the *Envelope* for future *EnvelopeQoSProcessor* calls and make envelope serialization easier.

4) Envelope Consumption: An *EnvelopeConsumer* can filter an *Envelope* and gets an embedded message thanks respectively to *EnvelopeFilter* and *EnvelopeToMessage* components. *Envelope*s are sent to an application (thanks to push and pull methods, see subsection II-B) or to the network (thanks to an integrated communication technology which like to interoperate with other ones, see section III). As a specialization of the *EnvelopeConsumer*, the *EnvelopeConsumerManager* is in charge of routing locally a message to *EnvelopeConsumer*s with specific distribution policies[1]. For efficiency reasons, it is able to filter an *Envelope* and to deliver messages to many *EnvelopeConsumers*.

5) R-MOM node interfaces: Package *envelope_processing* illustrated in Figure 1 represents all interfaces in charge of processing an Envelope.

Interfaces are *IEnvelopeSender*, *IEnvelopeReceiver*, *Protocol* and *IProtocolChecker*. Both *IEnvelopeSender* and *IEnvelopeReceiver* are used respectively to send and to receive serialized *Envelope*s from the network. An *IEnvelopeReceiver* inherits from *IEnvelopeProducer*, so its task is to get an *Envelope* buffer from the network, to convert it to an *Envelope* and to give the result to its *IEnvelopeProcessor*. *IEnvelopeSender* inherits from *IEnvelopeConsumer*, so, its business is to get *Envelope*, convert it into an array of bytes, and send it to the network. The *Protocol* interface enables to specialize the network protocol used by both *IEnvelopeSender* and *IEnvelopeReceiver*[2]. Therefore, both inherit from the *IProtocolChecker* interface which can check if a *Protocol* can be processed or not. An *EnvelopeReceiver* component can be bound to one *Protocol*, and connected to many *Protocols*, in order to receive messages from many sources. An *EnvelopeSender* component can be connected to a target using only one *Protocol*.

*Protocol*s are used to identify system node exchanges, and to ease adaptation from design-time to run-time. Architects and final users can use a *BindingFactory* component in order to register at run-time new R-MOM bindings and create related *EnvelopeSender*, *EnvelopeReceiver* and *Protocol*, from an URI. Parameterizing and to evolving a R-MOM node becomes as simple as to use URIs, whatever complex architecture provided by integrated asynchronous communication technologies (see section III). For example, 0MQ allows to specify a binding through the TCP transport layer with the simple URI value ”tcp:127.0.0.21:8080” which indicates that a message consumer or a message producer aims to be bound to the IP address ”127.0.0.21” and the port 8080. In the case of R-MOM, the value ”tcp:127.0.0.21:8080” creates an *EnvelopeSender* with the same URI as given in the 0MQ example, and the text ”jms:topic:my_topic” indicates to R-MOM to create a component bound to the JMS topic named ”my_topic”.

B. Message production/consumption generic interfaces

Message production and consumption respect push and pull modes, thanks to *IMessageProducer* and *IMessageConsumer* interfaces (see Figure 1) which inherit respectively from *IEnvelopeProducer* and *IEnvelopeConsumer* interfaces. The *IMessageProducer* produces message with context information (nfps is non-functional properties) or directly *Envelope*. The *IMessageConsumer* permits to use both pull and push modes, i.e., to be notified about Envelope reception with the *IMessageConsumerListener*, or to take and read manually a set of received Envelopes. *MessageProducer* and *MessageConsumer* are components which implement respectively *IMessageProducer* and *IMessageConsumer*.

C. Interoperability and bindings

In order to interoperate with existing MOM platforms, we provide *Protocol* and *Interface* bindings as a way to bind a MOM platform to R-MOM components. An example with 3 existing MOM platforms is given in Section III.

Protocol binding permits to produce or to consume messages with bound technologies, in keeping a reference to a related entity. This binding is a specialization of R-MOM

[1]”one to all”, ”one to one”, etc...

[2]For example, a ”socket” protocol is used to exchange bytes over UDP/TCP transport layers, containing an URI. A ”JMS” protocol will be used to realize JMS exchanges thanks to a destination name and a type (Topic or Queue).

EnvelopeReceiver and *EnvelopeSender* components (see sub-section II-A5). It can be configured with an URI in order to ease its use with existing technologies.

Interface binding permits to ease integration of R-MOM in legacy systems. This binding is a component which provides the same interface as proposed by the bound technology, but calls are redirected to R-MOM *MessageConsumer* and *MessageProducer* components (see the sub-section II-B "message production and consumption"). The used bridge design pattern with bound technology interfaces permits to delegate intermediate processing to R-MOM components, and does not imply that we use the bound technology to consume or to produce related messages.

Finally, the bound technology model can be respected thanks to an optional component binding from an *Interface* binding to a *Protocol* binding. Thus, a reference to a bound technology element can be accessible from an *Interface* binding.

D. Envelope structure and message context

This sub-section deals with interoperability and adaptation at the transport level and describes the structure of the envelope, such as a generic container for data value and data context information. The envelope is inspired from AMQP [3] which is a specification for interoperability between MOM at the transport level. The AMQP envelope is inspirited from the SOAP envelope, but it provides a binary format instead of the SOAP envelope XML format. Our envelope provides a structure more flexible and where data encoding size is smaller than the one from AMQP.

The Envelope structure is composed of three parts: a Dictionary of couples of quality of services name and value, the message value with information related to its serialization, and a last buffer which can be used by the application in order to extend Envelope data.

The package *envelope* from Figure 3 represents interfaces related to *Envelope* and *MessageDescription*, once the *Envelope* has been deserialized by the transport layer. The *Envelope* contains one message and one message buffer in order to be deserialized by other deserializers.

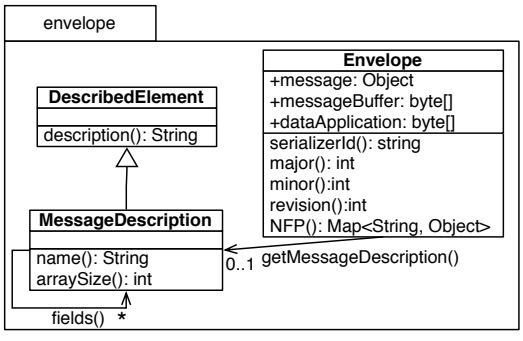

Fig. 3. **Envelope API** - UML class diagram

Even if the R-MOM serialization mechanism depends on envelope senders, generic informations exist in order to identify envelope parts from a serialized envelope.

Table I shows the structure of the R-MOM envelope. It is divided into three parts, header, body and footer, which contain respectively quality of services, message value and application data.

TABLE I
SERIALIZED ENVELOPE CONTENT

PART	PROPERTY	SIZE (Bytes)
HEADER	body position	≥ 1
	serializer Id	≥ 0
	QoS	≥ 0
BODY	foot position	≥ 1
	serializer Id	≥ 4
	Message	≥ 0
FOOTER	Application Data	≥ 0

In all parts described below, integer parameterized values are stored using the ProtoBuf [1] *varint* method to serialize integer values. This method allows to vary the buffer length depending on the integer value. It is not restricted to 32 or 64 bytes, and lesser or equals to the default serialization length, and independent from infrastructures. Therefore it is a good choice in order to save as much bytes as possible from the bandwidth consumption related to envelope size (instead of AMQP which constraints integer values to be coded on 32 or 64 bytes). All arrays of bytes are stored starting with a *varint* corresponding to their length. In the remainder of this section, we call *varint* the serialized type for an integer, *varintX* the serialized type for an integer coded on X bits and *varbytes* the serialized type for an array of bytes starting with a *varint* which is equals to the buffer length. Finally, if writing the message length, the envelope length is equals to 6 bytes, which is less than an int64 serialized by default serializers used by all existing MOM solutions, and less than the 8 header bytes from AMQP frames where the size of non-functional properties is restricted to at most $255 * 4 - 8 = 1012$ bytes, and non-functional properties and message encoding are imposed (and not evolvable).

1) Header: The head part contains two properties. First, the *body position* which is an absolute *varint* index location corresponding to the body beginning location in the buffer used to read the message content without parsing the header, and *serializer Id* for QoS. The serializer Id permits to deserialize QoS, respecting a logic based on dynamic, or static processing. A dynamic solution allows property types discovery, but costs in process execution time and buffer length:

- 0: most dynamic solution, the buffer contains couples of $(varbytes, varbytes)$, where key and value correspond respectively to property name and value.
- 1: between dynamic and static solutions, the buffer contains couples of $(varint, varbytes)$, where keys are stored as integer values, and have to be established between envelope sender and receiver.
- ≥ 2: full static solution, a list of $varbytes$, used with predefined and static QoS.

2) Body: The body part contains two properties, *foot position* and *serializer Id*, which correspond respectively to the

foot offset in the buffer, and to the message serializer identifier coded on at least 4 bytes for 1 *varbytes* and 3 *varint*s. The first one corresponds to a string identifier, for example "PBF" for ProtoBuf. The next 3 ones are related to the serializer version, with three major, minor and revision identifiers.

3) Footer: Footer content is application specific. Therefore, there is not R-MOM rules on this buffer space.

III. Integration of existing MOM platforms through bindings

In this section, we describe how R-MOM interoperates with JMS, RabbitMQ/AMQP and DDS. The interoperability is ensured with the binding approach (see Subsection II-C), an implementation of the *Protocol* interface (see Subsection II-A5) and data to *Envelope* transformation.

A. JMS - Java Message Service

JMS [7] is a MOM specification for Java programming. A common API exists in order to make possible the portability of a JMS-based application on top of different JMS implementation. The entry point provided by the ConnectionFactory is the only feature specific to each JMS engine.

We provide an abstract component model for using JMS in R-MOM. We rely on a *DestinationFactory* component in order to run this model. This component embeds a JMS Connection, and is able to create JMS destinations (Topic and Queue), JMS MessageConsumers, JMS MessageProducers, and JMS Messages. We validate our JMS binding model with the integration of JORAM[3], ActiveMQ[4] and OpenJMS[5], which are three existing JMS implementations.

JMS Interface binding components provide JMS MessageProducer and MessageConsumer interfaces. JMS protocol binding components use a JMS BytesMessage[6] in order to save or get a serialized Envelope. All JMS quality of services (QoS) from JMS messages are converted to R-MOM QoS values and saved in the Envelope, but only Envelope QoS which are JMS QoS too, are stored into JMS messages during JMS message distribution.

Respecting the JMS specification, the implementation of the *Protocol* interface uses two fields, a destination type (Queue or Topic), and a destination name.

An example of the interoperability between JMS and ActiveMQ is detailed in the Subsection IV-B.

B. DDS - Data Distribution Service

DDS [16] is an OMG specification, for a publish/subscribe data-oriented model. Data description is possible thanks to an IDL[7] file. Six concepts are required to publish or subscribe data: a *DomainParticipant*, a *Publisher*, a *Subscriber*, a *Topic*, a *DataWriter* and a *DataReader*. A *DomainParticipant* belongs to a domain in order to notify a DDS system that we want to participate over a domain. A *Topic* is the information related to a data type. A *DataWriter* depends on both *Publisher* and *Topic*, and is in charge to write data to its *Publisher*. A *DataReader* depends on both *Subscriber* and *Topic*, and reads data from its *Subscriber* (common quality of services to *DataWriter* and *DataReader* must match in order to perform data send).

DDS interface binding components provide *DataWriter* and *DataReader* interfaces. DDS protocol binding components contains optional references to *DomainParticipant*, *Publisher*, *Subscriber* and *Topic* in order to respect the DDS entity model. We respect data description in defining the *Envelope* with IDL files. The message content type is of CORBA type "any". A default topic is known over the system, able to be interested by all of these untyped *Envelope*. Users have to specialize the type of the embedded message into the *Envelope* in order to not be subscribed to all sent *Envelope*s.

The DDS *Protocol* interface implementation contains a domain name, a list of partition names, a topic name and (optionally) quality of services related to publication or subscription operations, and optionally, is able to create locally related entities (*DomainParticipant*, *Publisher*, *Subscriber* and *Topic*). Therefore, with the same result and control over the DDS API, our solution simplifies its use and configuration, in focusing on data distribution, and in using the simple DDS *Protocol* with three fields instead of manage the lifespan of five entities.

C. RabbitMQ/AMQP

AMQP is a specification of the Advanced Message Queuing Protocol [3]. It focuses only on message exchange protocol, and does not provide any API, contrary to JMS and DDS.

RabbitMQ is an AMQP compliant solution. In AMQP, only a *Channel* and a *ConsumerListener* are required to exchange messages. A *Channel* declares or deletes "Exchange" and "Queue" brokers between message consumers and producers. A *Channel* also sends arrays of bytes (no message serialization here) to an "Exchange", or to a "Queue", and a *ConsumerListener* receives data only from a "Queue".

RabbitMQ interface binding components implement *Channel* interface for data producing, and require *ConsumerListener* interface in order to receive data. RabbitMQ protocol binding components contain a *Channel*. RabbitMQ *Protocol* interface implementation contains only one field which corresponds to the channel name, as an "Exchange" for an *EnvelopeSender* and a "Queue" for an *EnvelopeReceiver*.

IV. Implementation

This section presents our R-MOM implementation, written in Java, and using FraSCAti [21] [2]. FraSCAti is a reflective implementation of the Service Component Architecture (SCA) specification [13]. Therefore, R-MOM is configurable thanks to platform independent model configuration files (SCA composite files), and uses reconfigurable interface bindings. The R-MOM implementation is available in the OW2 SVN repository at the address: *http://tinyurl.com/6v6hsqa*.

[3] http://joram.ow2.org/
[4] http://activemq.apache.org/
[5] http://openjms.sourceforge.net/
[6] JMS message containing an array of bytes
[7] Interface Description Language.

All interfaces presented in Subsection II-A5 such as the JMS API and component implementations are defined in a project independent from FraSCAti. Component are deployed thanks to Injection of Control (IoC) [6] mechanisms provided by FraSCAti coupled with SCA composite configuration files, in preserving R-MOM source codes from component model specificities.

A. Supported MOM technologies

Ten asynchronous communication technologies are supported by our implementation: JGroups[8], JBossMQ[9], JORAM[10], ActiveMQ[11], OpenJMS[12], RabbitMQ[13], OpenSplice[14], 0MQ[15], KryoNet[16], and Esper[17]. UDP and TCP are also supported for fast and simple exchanges. Even if KryoNet and Esper are not stricto sensus MOM, we reuse their send/receive API and develop bindings (see section III) in order to exchange data with all of them.

Three serialization libraries are reused through *EnvelopeToMessage* and *MessageToEnvelope* components: Java Serialization, ProtoBuf[18] (hosted by the Protostuff[19] library) and Kryo[20].

B. ActiveMQ/JMS example

Figure 4 is an example of an SCA configuration for a R-MOM core node with ActiveMQ/JMS binding components.

In this SCA composite, there are two JMS binding components (”producer” in lines 8-13 and ”sender” in lines 18-23), one JMS DestinationFactory component (”destinationFactory” in lines 4-7, see Subsection III-A) and two convertor components (”jmsMessageToEnvelope” in lines 14-17 and ”envelopeToBuffer” in lines 24-27, see Subsections II-A2 and II-A4) are defined. The ”producer” component implements the JMS message producer interface (line 9), and references the ”jmsMessageToEnvelope” (line 12) and the ”sender” components (line 11). ”jmsMessageToEnvelope” converts a JMS Message to an Envelope. ”sender” uses ”envelopeToBuffer” to serialize input Envelopes and send them through ActiveMQ protocol, where JMS destinations are initialized thanks to the ”destinationFactory” component (lines 4-7) (which is the only one component related to ActiveMQ, other ones use JMS API in order to initialize their properties). Finally, only the ”producer” component is promoted by the composite (line 3).

[8]http://www.jgroups.org/
[9]http://www.jboss.org/
[10]http://joram.ow2.org/
[11]http://activemq.apache.org/
[12]http://openjms.sourceforge.net/
[13]http://www.rabbitmq.com/
[14]http://www.opensplice.com
[15]http://www.zeromq.org/
[16]http://code.google.com/p/kryonet/
[17]http://esper.codehaus.org/
[18]http://code.google.com/p/protobuf/
[19]http://code.google.com/p/protostuff/.
[20]http://code.google.com/p/kryo/

```
1. <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2. <sca:composite
    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
    name="Producer">
3.  <sca:service name="SProducer" promote="producer/SProducer"/>
4.  <sca:component name="destinationFactory">
5.      <sca:implementation.java
            class="rmom.frascati.activemq.DestinationFactory"/>
6.      <sca:service name="SDestinationFactory"/>
7.  </sca:component>
8.  <sca:component name="producer">
9.      <sca:implementation.java
            class="rmom.frascati.jms.message.producer.Producer"/>
10.     <sca:service name="SProducer"/>
11.     <sca:reference name="REnvelopeProcessor"
            target="sender/SSender"/>
12.     <sca:reference name="RMessageToEnvelope"
            target="jmsMessageToEnvelope/SMessageToEnvelope"/>
13. </sca:component>
14. <sca:component name="jmsMessageToEnvelope">
15.     <sca:implementation.java
            class="rmom.frascati.jms.envelope.convertor.MessageToEnvelope"/>
16.     <sca:service name="SMessageToEnvelope"/>
17. </sca:component>
18. <sca:component name="sender">
19.     <sca:implementation.java
            class="rmom.frascati.jms.envelope.sender.EnvelopeSender"/>
20.     <sca:service name="SSender"/>
21.     <sca:reference name="RDestinationFactory"
            target="destinationFactory/SDestinationFactory"/>
22.     <sca:reference name="REnvelopeToMessage"
            target="envelopeToBuffer/SEnvelopeToBuffer"/>
23. </sca:component>
24. <sca:component name="envelopeToBuffer">
25.     <sca:implementation.java
            class="rmom.frascati.envelope.convertor.EnvelopeToBuffer"/>
26.     <sca:service name="SEnvelopeToBuffer"/>
27. </sca:component>
28.</sca:composite>
```

Fig. 4. **SCA R-MOM core node composite** - JMS interface/protocol binding message producer without QoS processing

V. Evaluation

We highlight in this section the interoperability capability efficiency provided by R-MOM compared to existing MOM. First we present the description of a SoS architecture, then we compare the design and the execution of this SoS with *(i)* and without *(ii)* R-MOM.

A. Description

Figures 5 presents the SoS architecture. The scenario consists to check activity in a room. A movement sensor *(m)* and a video camera *(v)* send respectively events via UDP and images via DDS, to an alarm terminal *(a)*. *(a)* verifies if *(m)* checks out movement from the room, and if true, sends information from sensors with the current date to a Control Terminal *(c)* via JMS or AMQP. Finally, *(c)* processes received information with a *MessageProcessing* component in order to fire a critical or minor alert.

The whole system is deployed on four Linux Ubuntu 11.10 virtual machines, one per sub-system node, over a MacBook Pro with the MacOSX 10.7.2 operating system installed on the following architecture; Processor: 3.06 GHz Intel Core 2 Duo, Memory: 4 GB 1067 MHz DDR3. System clock of virtual machines is synchronized with the hosted machine. MOM technologies and R-MOM bindings used are OpenSplice/DDS, ActiveMQ/JMS and RabbitMQ/AMQP.

B. Comparison

During the design phase, only adapter component implementations *(A)* are different in *(i)* and *(ii)*. R-MOM is used in

(i) such as a smart adapter, instead of in *(ii)* which uses simple adapters. Therefore, in our architecture view (see Figure 5), there is no concrete difference between *(i)* and *(ii)*. In this use case, R-MOM is portable to legacy systems. At run-time, *(i)* provides much possibilities than *(ii)* detailed in this subsection.

Figure 6 shows the timeline scenario related to our evaluation case, during 1 minute. The goal is to realize different adaptation tasks. That is to say to modify *(D)* dynamic and *(S)* static properties for all data transmission protocol (UDP, DDS, JMS and AMQP), and *(P)* to change of data transmission protocol (change JMS by AMQP) between *(a)* and *(c)*. *(D)*, *(S)* and *(P)* are usually done for scalable reasons. For example, they consist to modify quality of services. *(S)* is a redeployment task.

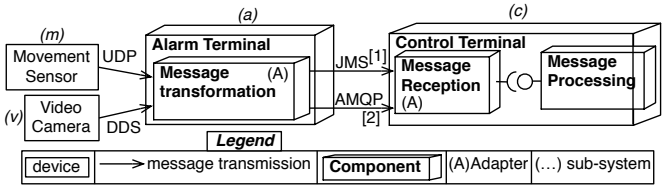

Fig. 5. **Architecture of the SoS use case** - Security Device processing and change of MOM at run-time from JMS [1] to AMQP [2]

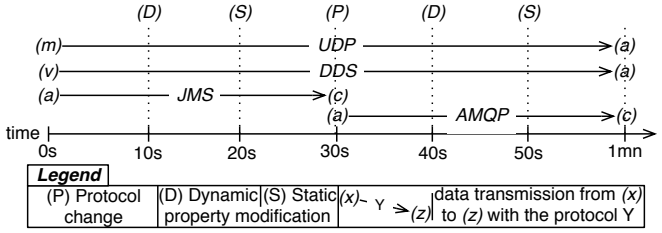

Fig. 6. **SoS Timeline** - during 1 minute

Before analyzing the micro-benchmarks related to the execution time of the adaptation tasks and data transmission, it is important to understand that *(P)* brings a consequence about system availability time. In *(i)*, *(P)* consists to add AMQP bindings into adapter components, and so, to avoid to loose data thanks to JMS bindings. In *(ii)*, *(P)* consists to change adapter components, and the consequence is an system unavailability time which corresponds to component change time and AMQP starting time.

Let's see what is the overhead on data transmission introduced by R-MOM, and *(P)* time in *(ii)*.

Figures 7, 8, 9 and 10 show micro-benchmarks related to data sending and reception time for each one of data transmission protocols. Results are an average on 10 sessions of 10,000 exchanged data, with data size variation: 8B, 512B, 1KB and 32KB.

One general remark is that the R-MOM reconfigurable component model layer takes from 5 to 15 μs more time to process a message than direct call to communication layers. Therefore, in the case of a very fast transport layer such as UDP which takes about the same time to process sending

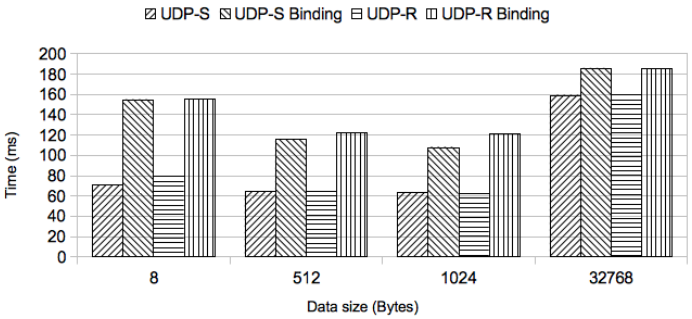

Fig. 7. **Average time about 10 times 10.000 data sending (-S) and reception (-R) with UDP and related R-MOM bindings** - data of size 8B, 512B, 1KB and 32KB

and reception tasks than R-MOM, UDP is half as long for data lesser than 1KB (see Figure 7). Otherwise, processing times depend on data packets serialization size, communication technology layer complexity and constant component service call duration. Micro-benchmarks about DDS, JMS and AMQP (see Figures 8, 9 and 10) ensure this remark. More data packets serialization size and communication technology layer complexity increases, more *(i)* and *(ii)* processing time difference decreases. For example, with DDS, the fastest evaluated MOM solution, time process becomes negligible, i.e. bindings use time average is sometime lesser than in *(ii)* for the DDS micro-benchmark about reception processing time for data size of 1KB.

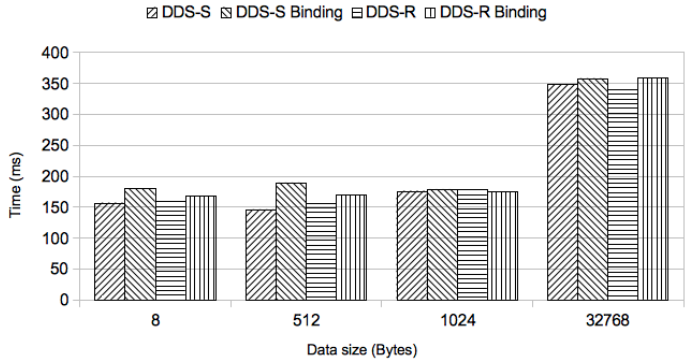

Fig. 8. **Average time about 10 times 10.000 data sending (-S) and reception (-R) with DDS and related R-MOM bindings** - data of size 8B, 512B, 1KB and 32KB

Table II shows adaptation task durations for *(i)* and *(ii)*. *(i)* is 5 to 300 μs more longer than *(ii)*, because it consists also to initialize all component layers. As said previously, *(P)* imposes system unavailability time only in *(ii)*, i.e. component change processing time plus deployment time of AMQP components (about 500 μs for a sender and 2 ms for a receiver).

C. Conclusion

Instead of usual solutions, R-MOM adaptive capabilities permit to avoid system unavailability time, and to exchange messages with a negligible overhead inducted from the interoperability. Therefore R-MOM is an interoperable and useable solution for MOM communications in distributed systems.

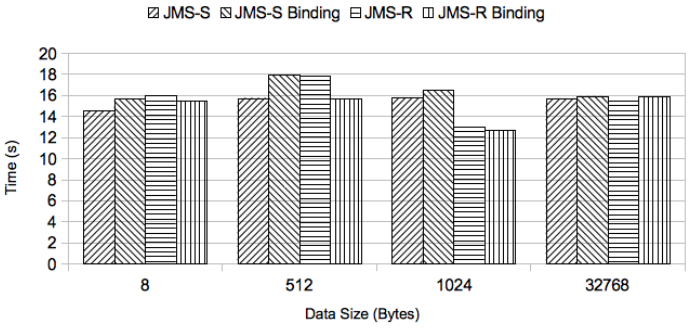

Fig. 9. **Average time about 10 times 10.000 data sending (-S) and reception (-R) with JMS and related R-MOM bindings** - data of size 8B, 512B, 1KB and 32KB

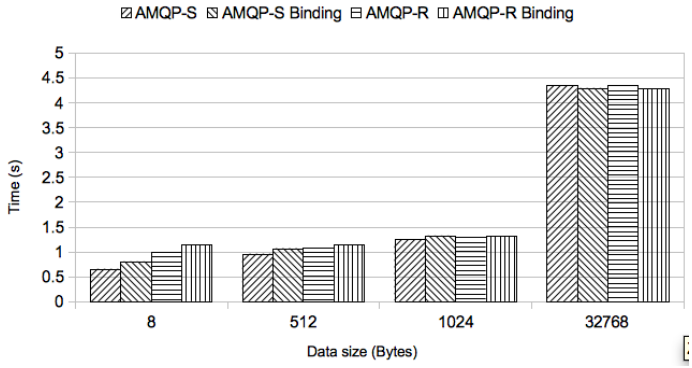

Fig. 10. **Average time about 10 times 10.000 data sending (-S) and reception (-R) with AMQP and related R-MOM bindings** - data of size 8B, 512B, 1KB and 32KB

VI. Related work

This section compares R-MOM with related work about interoperability for MOM solutions. Interoperability comparison is made over three high to low levels: architecture, business/operational and protocol.

A. At the architecture level

Since 2007, the OSOA Service Component Architecture (SCA) specification [5] provides an architecture and programming models for distributed systems which have been endorsed by the OASIS consortium [13]. Even if an OSOA SCA event specification [4] exists, no implementation is known at the time of the writing of this paper. However the specification imposes to use channels which correspond to our couple of *EnvelopeQoSManager* and *EnvelopeConsumerManager* components in order to configure local distribution and message filtering. Adaptive concerns are ensured by binding such as *EnvelopeProducer* and *EnvelopeConsumer* components and their specialization.

B. At the business/operational level

The OMG's CORBA Component Model (CCM) specification [15] provides communication through Remote Procedure Call (RPC - facet and receptacle ports) and Event (sources and sinks). In order to reuse legacy systems or use new MOM features as realized with large sets of Quality of

TABLE II
Average time in μs for 10 deployment[1] operation and 20 dynamic[2] and static[3] property modifications for MOM solutions and R-MOM bindings

Kind	NODE TYPE	DEP[1]	DYN[2]	STAT[3]
UDP	Sender	110	5	15
	EnvelopeSender	140	20	30
	Receiver	220	5	20
	EnvelopeReceiver	260	10	25
JMS	MessageProducer	2212	942	994
	EnvelopeSender	2198	970	908
	MessageConsumer	3148	1051	960
	EnvelopeReceiver	3347	1049	1004
AMQP	MessageProducer	524	16	29
	EnvelopeSender	719	17	33
	MessageConsumer	2117	1	12
	EnvelopeReceiver	2331	20	30
DDS	DataWriter	50	2	45
	EnvelopeSender	76	20	62
	DataReader	123	5	101
	EnvelopeReceiver	155	17	135

Services (QoS) from the OMG's Data Distribution Service (DDS), system developers have to develop connectors [17]. They become a new communication means, between CORBA and the other middleware solutions, but not integrated to the CORBA event based layer. Even if the business is close, a gap is inducted from the architecture.

In 2004 [12] presents three interesting approaches about the use of interoperability in MOM/Event models thanks to Java CORBA and IIOP as an interoperable communication protocol, but the system consists to interoperate with event channels, and not at the level of message consumer and producer, and neither for quality of services.

DREAM [9] is a component framework for the construction of reconfigurable MOMs. DREAM and R-MOM are based on a reflective component model, but DREAM imposes a complex API in order to manage messages (in recycling messages, and adding chunks and messages to one message) which is not required by all MOM platforms. Even if DREAM can use bindings in order to resolve interoperability, it does not provide an adaptive structure as the R-MOM Envelope, and the simple Envelope processing API eases the integration of technology and adaptation tasks related to specific behaviours.

PolyORB [23], the schizophrenic middleware with multiple applicative personalities, describes solutions and implementations able to interoperate with other RPC, MOM or Distributed Shared Memory models, and keeps safe personality whatever the used technology. Proposed as a smart adapter, its "neutral layer" corresponds to our couple of *EnvelopeQoSManager* and *EnvelopeConsumerManager* components. But request processing is the same for all communication paradigms. Therefore, this is its strength and weakness, because it does not support dynamic quality of services processing, message description, filtering as R-MOM where the corresponding "neutral layer" is configurable and reconfigurable at run-time. Data serialization is not as configurable as in R-MOM, i.e. not concerned by system requirements in terms of encryption or size content.

C. At the protocol level

Unlike previous MOM, the open Internet protocol for business messaging AMQP [3] aims to describe a specification about an interoperable protocol used to exchange messages, with its contents and additional features. Unfortunately, the AMQP does not provide evolution for message structure, as proposed by Google ProtoBuf [1], or about quality of services, and imposes to use some of them (as persistency for example), or a maximum which has to be serialized on at most 1012 bytes (see sub-section II-D).

VII. Conclusion

This paper presents R-MOM an interoperable and adaptive component-based framework for asynchronous communications in distributed systems, from the architecture level to the transport layer (see Section II). It simplifies complex architecture imposed by asynchronous communications in focusing only on sending and reception data (see Section III) and in using *Protocols* (see Subsection II-A5). It makes interoperability between ten different asynchronous communication solutions (see Section IV). Thanks to memory and efficiency concerns, R-MOM aims to be applied on several different systems, from embedded to IT systems. R-MOM is flexible enough to make enable at run-time with negligible processing time and avoid system unavailability time (see Section V). Finally, even if some existing solutions propose interoperability, there is no solution as complete as R-MOM in terms of interoperability and adaptive concerns (see Section VI).

In order to extend this work on adaptive distributed systems, we propose three perspectives:

- add a local event layer in order to be notified about envelope processing errors and keep safe R-MOM node from inconsistency states related to bound technologies or QoS processing.
- use R-MOM to provide an introspection mechanism for distributed systems using reflective components. Also, all components use R-MOM to notify the whole system about its status information.
- add connection components in charge to create a solid communication adapter between R-MOM nodes for security reasons.

Acknowledgments

This work is partially funded by the French Ministry of Higher Education and Research, Nord–Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region Campus Intelligence Ambiante (CPER–CIA) 2007-2013 and the grant agreement no. ANR–08–SEGI–010–01 (ITEmIS).

References

[1] Google Inc. ProtoBuf - Protocol Buffers - Google's data interchange format - http://code.google.com/p/protobuf/. WebSite, February 2011.

[2] ADAM. Inria/FraSCAti home page - http://frascati.ow2.org/.

[3] AMQP Working Group. *AMQP Recommendation*, August 2010.

[4] Michael Beisiegel, Vladislav Bezrukhov, Dave Booz, Martin Chapman, Mike Edwards, Anish Karmarkar, Ashok Malhotra, Peter Niblett, Sanjay Patil, and Scott Vorthmann. *SCA - Assembly Model Specification Extensions for Event Processing and Pub/Sub*. OSOA Community, April 2009.

[5] Michael Beisiegel, Henning Blohm, Dave Booz, Mike Edwards, Oisin Hurley, Sabin Ielceanu, Alex Miller, Anish Karmarkar, Ashok Malhotra, Jim Marino, Martin Nally, Eric Newcomer, Sanjay Patil, Greg Pavlik, Martin Raepple, Michael Rowley, Ken Tam, Scott Vorthmann, Peter Walker, and Lance Waterman. *SCA - Assembly Model Specification*. OSOA Community, March 2007.

[6] Dearle, Alan. Software Deployment, Past, Present and Future. In *2007 Future of Software Engineering*, FOSE '07, pages 269–284, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. *Java Message Service Specification*. Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303 U.S.A., April 2002.

[8] Mohammad Hassan, Biao Song, and Eui-Nam Huh. A dynamic and fast event matching algorithm for a content-based publish/subscribe information dissemination system in Sensor-Grid. *The Journal of Supercomputing*, 54:330–365, 2010. 10.1007/s11227-009-0327-0.

[9] Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. DREAM: a component framework for the construction of resource-aware, reconfigurable MOMs. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, ARM'04, pages 250–255, New York, NY, USA, 2004. ACM.

[10] Shen Lin, François Taïani, and Gordon S. Blair. Facilitating Gossip Programming with the Gossipkit Framework. In René Meier and Sotirios Terzis, editors, *Distributed Applications and Interoperable Systems 8th IFIP WG 6.1 International Conference, DAIS 2008, Oslo, Norway, June 4-6, 2008.*, volume 5053/2008. Springer Berlin / Heidelberg, May 2008.

[11] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[12] Gil Seong Na and Sang Ho Lee. Interoperability of Event Service in Java ORB Environment. In *Proceedings of Tenth International Workshop on Database and Expert Systems Applications*, pages 29–33. IEEE, 2002.

[13] OASIS. *Service Component Architecture Assembly Model Specification*. OASIS, January 2010.

[14] James A. O'Brien and George Marakas. Introduction to Information Systems. New York, NY, USA, 2007. McGraw-Hill, Inc.

[15] OMG. *Corba Component Model Specification*, 4 edition, June 2001.

[16] OMG. *Data Distribution Service for Real-time Systems*. OMG, January 2007.

[17] OMG. *DDS for Lightweight CCM*. OMG, February 2009.

[18] OMG. *The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification*. OMG, January 2009.

[19] Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[20] Mike Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16:389–415, June 2007. 10.1007/s00778-007-0044-3.

[21] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 42, 2012.

[22] Amirhosein Taherkordi, Frédéric Loiret, Azadeh Abdolrazaghi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen. Programming Sensor Networks Using REMORA Component Model. In LNCS 6131 (Springer), editor, *6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS'10)*, volume 6, pages 45–62, Santa Barbara, California, USA, June 2010.

[23] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. Polyorb: a schizophrenic middleware to build versatile reliable distributed applications. *Reliable Software Technologies-Ada-Europe 2004*, pages 106–119, 2004.

[24] W.M.P. and van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, 1999.

[25] P. Wolfgang. *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.