Improving data-intensive EDA performance with annotation-driven laziness

Quirino Zagarese, Gerardo Canfora, Eugenio Zimeo, Iyad Alshabani, Laurent Pellegrino et al.

PII: S0167-6423(14)00134-8
DOI: 10.1016/j.scico.2014.03.007
Reference: SCICO 1739

To appear in: *Science of Computer Programming*

Received date: 1 July 2013
Revised date: 18 February 2014
Accepted date: 13 March 2014

Highlights

- A novel architectural model for improving the efficiency of publish/subscribe middleware.
- A framework (WS-Link) that implements the model for SOA-based event-driven interactions.
- Java annotations to program the lazy strategies that guide the framework.
- An intense experimentation analysis of a complete implementation of the framework in a Grid5000-based testbed.

Improving Data-Intensive EDA Performance with Annotation-driven Laziness

Quirino Zagarese, Gerardo Canfora, Eugenio Zimeo

Department of Engineering, University of Sannio, Benevento, ITALY

Iyad Alshabani, Laurent Pellegrino, Amjad Alshabani, Françoise Baude

INRIA, I3S-CNRS, Université de Nice Sophia Antipolis, FRANCE

Abstract

Event-driven programming in large scale environments is becoming a common requirement of modern distributed applications. It introduces some beneficial effects such as real-time state updates and replications, which enable new kinds of applications and efficient architectural solutions. However, these benefits could be compromised if the adopted infrastructure is not designed to ensure efficient delivery of events and related data. This paper presents an architectural model, a middleware (WS-Link) and annotation-based mechanisms to ensure high performance in delivering events carrying large attachments. We transparently decouple event notification from related data to avoid useless data-transfers. This way, while event notifications are routed in a conventional manner through an event service, data are directly and transparently transferred from publishers to subscribers. The theoretical analysis shows that we can reduce the average event delivery time by half, compared to a conventional approach requiring the full mediation of the event service. The experimental analysis confirms that the proposed approach outperforms the conventional one (both for throughput and delivery time) even though the middleware overhead, introduced by the specific adopted model, slightly reduces the expected benefits.

Email addresses: `quirino.zagarese@unisannio.it` (Quirino Zagarese), `gerardo.canfora@unisannio.it` (Gerardo Canfora), `eugenio.zimeo@unisannio.it` (Eugenio Zimeo), `iyad.alshabani@inria.fr` (Iyad Alshabani), `laurent.pellegrino@inria.fr` (Laurent Pellegrino), `amjad.alshabani@inria.fr` (Amjad Alshabani), `francoise.baude@unice.fr` (Françoise Baude)

Keywords: web-services, SOA, EDA, publish-subscribe, data-intensive applications

1. Introduction

Service Oriented Architecture (SOA) supports flexible design of complex and multi-organization applications [5]. Loose coupling and interoperability among software components constitute the main drivers of this architectural model: the former is achieved by spatially decoupling services through specific mediators, such as registries and brokers, the latter by exploiting standard protocols and semantics.

In spite of its flexibility, SOA-based applications are typically implemented by exploiting procedural programming models, which emphasize remote service calls and workflows. Even though this model can be pragmatically applied to a large class of applications, it fails in some domains where events represent first-class concepts to deal with.

Call stack based interaction assumes that one thing happens after another, identifying a single path of execution where the caller's does not continue to run until the called method completes. If invocations are slow or carry a large amount of data, call stack based interactions become inefficient, and if the services to call are not known a priori, even poorly flexible.

Many computer systems, especially embedded ones, are designed to respond to events, e.g. the thermostat signals a low value of the environmental temperature and sends a command to turn on the boiler. On large scale, as computer systems become more interconnected they start to handle an increasing number of events, e.g. an order management system may receive orders from a web site and notify other systems, such as the financial one, to check for example whether a credit card is valid, and the warehouse, to verify that inventory to fulfill the order is present.

In this new scenario, some new properties characterize software systems: event propagation (events are propagated to any interested party that is listening to some events to process); timeliness (systems publish events as they occur instead of storing them locally); asynchrony (the system that fires an event does not wait for the receiving system). These properties significantly change the behavior of SOA-based systems by introducing an important shift of responsibility. It allows components (a) to be decoupled, since the caller is no longer aware of the sequence of functions to execute and the components that will execute them, (b) to keep the state which are interested in, since they do not query other systems for information but instead exploit their own copy of the required data.

Events represent state transitions and are commonly modelled as messages comprising a header, that contains message-specific information like priority or expiration time, and a payload, that contains user-specific information [7].

In SOA, event-driven messaging [6] enables the exchange of messages driven by events and subscriptions, thus avoiding inefficient polling interactions. This kind of event driven interactions led to a new type of architecture known as *Event Driven Architecture* (EDA)[31]. Typically, these messages can be used to carry documents (Document Message pattern) or files (File Transfer pattern) when state transitions occur [11]; for these reasons, messages typically include attachments to carry large amount of data, as proposed by modern Enterprise Service Bus (ESB) like JBossESB [1].

On the other hand, the bigger is the size of such attachments, the more effort is delegated to the event-delivery service, especially when the amount of events grows. Moreover, the event subscriber may not be able to handle the attachment, thus wasting the resources employed to transfer it.

The paper presents an architectural model, a middleware (WS-Link) and annotation-based mechanisms to program and execute efficient event-driven applications. The proposed middleware allows the data attached to an event to be directly transferrd from the publisher to the subscriber, thus reducing the traffic to and from the event-delivery system. The data are moved only if needed by the subscriber, but this one is not aware of the "lazy" nature of communication and attachments transfers do not imply any further coding effort. The solution we propose is illustrated by an implementation of a pub/sub infrastructure which handles semantically described events.

The remainder of the paper, an extended version of [34], is organized as follows. Section 2 describes the context that originated this work. Section 3 details our architecture and presents a pub/sub scenario. Section 4 focuses on data-transfers and explains how we avoid the useless ones to increase system efficiency. Additionally to the material published in [34], Sections 5 and Section 6 are providing evidences regarding the situations where

[1]`http://www.jboss.org/jbossesb`

the proposed solution provides performance improvements. More precisely Sections 5 characterizes three pub/sub communication patterns in terms of number of exchanged messages and attachments transfers. Section 6 analyses the performances in terms of event delivery time and throughtput and shows the improvements that can be achieved by employing our data-transfer technique. Section 7 discusses some relevant related work. Finally, section 8 concludes the paper and introduces future work.

2. Motivating context: EventCloud

Delivering events in large scale contexts is becoming a recurrent requirement of many distributed applications, such as those that exploit social networks and pervasiveness to communicate real-time state changes that characterize the behavior of user agents. Moreover, events used by these applications typically carry large attachments that represent a copy of the state changes that trigger the events. In this new scenario, existing solutions for event delivery based on the publisher/subscriber (pub/sub) architectural pattern could exhibit some limitations when the amount of events (carrying attachments) generated by the interacting peers is huge.

One possible solution to address the problem is to make the event service subsystem more scalable by replicating it as a whole or by designing its internals with alternative paths, replicas and caching to connect publishers and subscribers [30]. Even though this kind of scalability could be achieved by working on the deployment configuration or on the design, we argue that by changing the conventional pub/sub model, so that subscribers receive attachments directly from publishers, the architecture could become self-scalable and efficient.

Self-scalable, since different subscribers on average receive messages from different publishers, thus automatically creating a distributed event service. Efficient, since the communication of attachments is direct from the publisher to the subscriber. This way, the store and forward delay characterizing the indirect communication of conventional pub/sub architectures is avoided.

However, supporting transparent event decoupling, into notification plus attachments, and differentiated routing paths requires significant middleware interventions. In the paper, we propose a framework and an architectural approach that allow existing infrastructures, implementing the conventional pub/sub model, to be reused as it and without suffering of performance degradation when events carry large attachments. The framework was initially designed and developed to allow attachments delivery in EventCloud (EC) [21], a pub/sub infrastructure for asynchronous invocation of services developed in the context of PLAY [2] project, even though EventCloud could be replaced by any equivalent pub/sub infrastructure.

PLAY is a platform that allows for "event-driven interaction in large highly distributed and heterogeneous service systems". Subscribers register their interest in some type of events in order to asynchronously receive the ones that are matching their concerns. Events descriptions, inside the EC, are represented as sets of quadruples. Quadruples are in the form of (context, subject, predicate, object) where each element is a dubbed RDF term in the RDF [14] terminology, a standard for data description in the semantic web. The context value identifies the data source; the subject of a quadruple denotes the resource, the statement is about; the predicate defines a property or a characteristic of the subject; finally, the object is the value of the property.

Since the EC supports content-based subscriptions, formulated as SPARQL [22] queries, subscribers can specify fine-grained constraints on each RDF term of quadruples. Each published event is stored on the EC to be retrieved later, by means of a standard RDF datastore. Storing events can be useful to create a knowledge-base that may be used at any time to provide statistics or to correlate events (e.g. by employing a Complex Event Processing engine).

3. Decorating existing pub/sub Event Services with WS-Link

The high level of expressiveness offered by the EC event-format makes it ideal for open environments. On the other hand, the EC has not been designed for the delivery of attachments. In order to keep such expressiveness and to enable attachments delivery, we have extended its features by applying the *Decorator* pattern [26].

The resulting architecture is depicted in Figure 1. Publishers and subscribers interact with the EC by means of the Cloud Proxy Layer, and employ an event-format that enables attachments. An event transporting data, named *Data-Event* hereafter, contains a semantic description, which is the

[2] `http://www.play-project.eu`

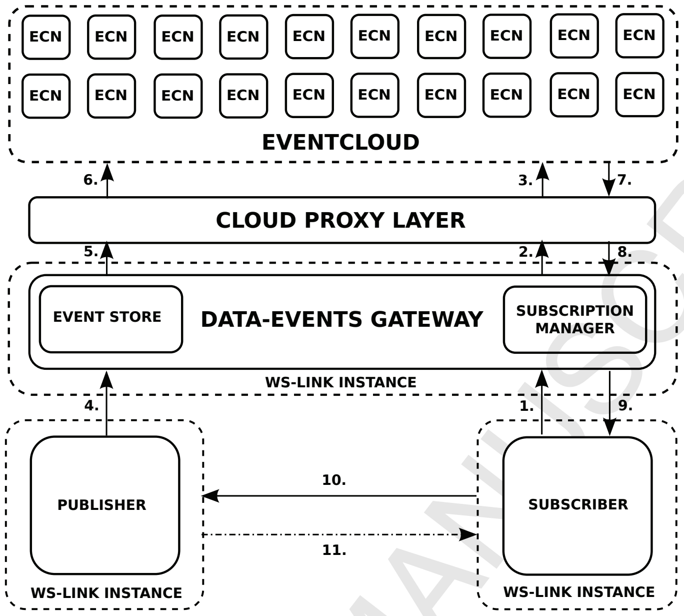


Figure 1: Proposed middleware architecture.

EC event itself, a list of attachment descriptors and the attachments themselves. An attachment can be either a resource (e.g. a file) or an XML document representing a serialized object. In the former case, the corresponding descriptor contains information like the name and the MIME type. In the latter case, the descriptor may provide the language-specific type (i.e. the Java fully qualified name of the class, in case the event should be handled only by Java-based subscribers) or a URI to a WSDL document describing the structure of the serialized object. The URI can be employed to enable dynamic instantiation on the subscriber-side.

Publisher and subscribers now interact with *Data-Events Gateways*, that are responsible for decoupling the semantic description of an event, containing RDF quadruples, from its attachments, that is not used for subscription matching.

Hereafter, the specific event service we target is the EC, where the pub/-sub matching occurs. However, our architecture can be applied to any event service as soon as the event format allows attachments to be decorrelated from event description.

A simple publish-subscribe scenario can better explain the role of each component inside the architecture. When a service wants to subscribe for a specific kind of event, it interacts with the gateway that exposes a *WS-Notification* interface [3]. The gateway prescribes a specific subscription structure. A subscription can be either topic or content-based: in the first case, the content is a simple string representing the topic; in the second case, a query language can be specified and the content is a query written in this language (currently SPARQL).

The subscription is performed by invoking the *subscribe* operation exposed by the gateway (interaction 1 in Figure 1). The gateway is responsible for collecting all the subscriptions and saving them to a *Subscription Manager* (SM). Then, it subscribes itself to the *EC*, by interacting with the *Cloud Proxy Layer* (CPL) (interactions 2 and 3). The *CPL* enables a high level of flexibility, since it decouples the gateway from the *EC*: if these components are deployed locally to each other, the *CPL* will enact interaction 3 by means of a local method invocation; on the contrary, if the *EC* is remotely deployed, the interaction will follow the *WS-Notification* standard. The *EC*, itself, is in charge to handle both topic and content-based subscriptions, by employing a matching algorithm, whose details can be found in [21].

We adopted a two-layered subscription system, where gateways act as filters. When new external subscriptions arrive, the gateway layer stores them, checks if there are other subscriptions for the same topic (or kind of content), by querying the *SM*, and, if not, registers itself to the *EC* layer for the specified type of events. The subscription is finally stored in the *EC*.

When a service raises an event, this will be sent to the gateway by invoking its *notify* operation (4). When the event is received by the gateway, the latter is in charge of creating an identifier for the event, adding it to its description and forwarding the description to the *EC* (5, 6). The whole event is kept inside an *Event Store*, that is local to the gateway. There is no need to move the whole event inside the *EC*, since only its description is used to match existing subscriptions. Once the event description enters the *EC*, it will be inspected to verify if it matches any previous subscription. In case of matching, the *EC* notifies the gateway, by means of the *CPL* (7, 8). The gateway extracts the event identifier from the event description, queries the *Event Store* to retrieve the whole event and, finally, dispatches it to the actual subscribers, by means of the *SM* (9).

Despite the described architecture avoids attachments transfers from/to

[3]`http://www.oasis-open.org/committees/wsn`

the *EC*, attachments are moved from publishers to gateways, and then back to subscribers. To achieve a higher degree of efficiency, attachments could be transferred directly to subscribers. Moreover, such optimization should not impact the design of publishers' and subscribers' business logic. To address the problem we have designed and developed WS-Link.

It extends the DynO4WS framework, that provides Dynamic Object Offloading capabilities to web-services endpoints, by taking advantage of the Apache CXF [4] message interception API. DynO4WS (Dynamic Offloading for Web-Services) is a middleware aimed at decoupling the semantics of service invocations from the way data is moved between interacting service-endpoints. To this end, the framework allows for the customization of the transfer process of the attributes belonging to entities exchanged during service invocations. Such customization takes place by plugging a *LoadingStrategy* (see Figure 2). This abstract component decides which attributes, inside IN/OUT parameters of services invocations, should be serialized at once, and which ones should be made available for later access. This may be useful because such attributes are not likely to be used by the remote endpoint or because they exhibit a considerable size. The attributes that are not sent at once are made available from an *OffloadingRepository*. This component can be implemented either as an in-memory map, to achieve fast read/write accesses, or as a persistent store, in order to improve reliability.

Message persistence is extremely useful in the considered scenario and it is a feature provided by most message oriented middlewares (MOM). In the pub/sub communication style, publishers are not required to be active when a message is received and subscribers are not required to be active when a message is sent. This behaviour is ensured by persistent queues that typically are hosted by third party servers (message brokers). In our model, each publisher can manage its own message repository (that could also be shared among several publishers, if needed) that is used to decouple the publisher of a message X from all the subscribers for that message and to ensure message persistence. Therefore, the concept of overlay network built around a network of message brokers is still present in our model. To this end, it is worth to note that, with our approach, traditional message brokers (in practice, the EventCloud) are not removed: events are propagated in the usual way from the publishers to the subscribers through the network

[4]`http://cxf.apache.org`

of message brokers. Only attachments are stored in repositories local to the publishers waiting to be retrieved by the subscribers that are effectively interested in consuming also the attachments of an event. This way, our message broker overlay network is composed of local repositories and third party message brokers.

It is worth noticing that WS-Link can support any kind of persistence store, either relational or NoSQL. So far, we have implemented and evaluated both a JDBC based connector for relational databases (which as been tested with a MySQL database) and a MongoDB connector [2].

4. Annotation driven laziness with WS-Link

One key feature of WS-Link is its transparency: developers do not have to change any line of code of the endpoints business logic, thanks to a dynamic proxing system we have extensively described in [33]. When offloading takes place, the framework adds specific meta-data to the header of the outgoing SOAP message concerning which attributes have not been sent yet, and how they can be retrieved. When the framework instance on the remote endpoint receives the message, the *ProxyManager* generates a proxy according to such meta-data, thus hiding the loading strategy.

WS-Link extends our previous work by implementing a specific *Loading Strategy* that enables fine-grained configuration for those attributes that should not be transferred as in a common Web service interaction, by means of Java annotations [3].

Before we get inside the details about how we use Java annotations for the configuration of data transfers in WS-Link, it is important to clarify that annotations represent only a possible technique to specify non-functional requirements. In facts, Java annotations can be "a double-edged sword": despite they are more succint than XML and also avoid the problem of fragile linkages between the source code and the XML metadata, they are still part of the source code, with all the consequences this brings (e.g. application and framework are tightly coupled, the application must be rebuilt when the configuration changes, etc.) [23].

Our approach works even when we consider services as black boxes, since the configuration is applied only to exchanged entities (in our case the class representing events), not to services definitions or implementations. This does not mean the code implementing the service must be modified. In this specific scenario, in facts, the event service and the subscribers are services, while publishers are Web service clients. The event class on the publisher side is generated from the event service WSDL. The generated class is then annotated on the client-side, while the service implementation is not touched at all.

4.1. Java annotations: key concepts

Java annotations are modifiers aimed at associating information with the annotated program elements. An annotation is declared as a Java interface, but its name must begin with the '@' character. The information is stored in annotations in the form of key-value pairs or other annotations. They are always characterized by a *Target* and a *Retention*. The target indicates the program elements - type declarations, methods, fields, parameters - that can be annotated with the specified annotation (WS-Link uses annotations to annotate class fields). The retention indicates when an annotation should be processed and the Java language provides three types of policies:

- SOURCE: the annotation is discarded by the compiler. This kind of annotations are used to add meta-data which are useful for IDEs or for developers reviewing code (e.g. the @Override annotation indicates that a method declaration is intended to override a method defined in a supertype).

- CLASS: the annotation is recorded in the class file by the compiler but need not be retained by the VM at run time. These annotations are usually employed for compile-time extensions like code obfuscators or bytecode engineering libraries.

- RUNTIME: the annotation is recorded in the class file by the compiler and retained by the VM at run time, so it may be read reflectively. These annotations are used to enable and drive the behaviour of frameworks that collect the attached information by means of the reflection API.

WS-Link uses this kind of retention policy to read attachments configuration and optimize data-transfers.

4.2. Java annotations in WS-Link

In WS-Link, data-transfers customization can take place by means of three key annotations. *@Strategy* allows the developer to specify a component that is in charge of deciding when the marked attribute should be transferred. Basically, an attribute can be transferred when the invocation takes place, or on-demand, when the remote endpoint actually needs its value. Anyway, there can be several reasons to choose between these options and some of them depending on the runtime value of the attribute. For instance, if the size of the attribute is negligible, it could be useless to delay its transfer. The framework lets the designer plug its own strategy and configure it at the attribute level, by means of key-value pairs that can be nested inside the *@Strategy* annotation (using the *@StrategyParam* syntax). Listing 1, for instance, shows a configuration instructing the middleware so that the attachments field is sent lazily only if its size is larger than 1MB. Every time an event is raised, the framework retrieves the value of the *@Strategy* annotation, along with the nested key-value pairs, and handles it by means of the *ConditionalLazyStrategy* class declared in the annotation.

Listing 1: Configuring WS-Link so that the attachments field is sent lazily, only if its size exceeds 1MB.

```
@Strategy(impl = ConditionalLazyStrategy.class,
    params = {
     @StrategyParam(
       name="minsize",
       value=(1024*1024)),
})
private byte[] attachments;
```

The strategy can dynamically decide whether the annotated attribute should be serialized, since at each invocation it receives a map containing the key-value pairs declared inside the *@Strategy* annotation and the event instance. More specifically, these parameters are used to invoke the *shouldOffload* method, defined in Listing 2, that each strategy must implement. The class implementing a strategy cannot be selected at runtime, but it could apply different criteria, based on runtime inputs.

@Consistency can be used to indicate whether it is necessary to keep a serialized copy of the attribute value, as it was at the moment of the invocation, in case its transfer is going to be delayed, according to the strategy.

Listing 2: The hook method each strategy must implement to get integrated into WS-Link.

```
public boolean shouldOffload(Object field,
    OffloadingContext context,
    Map<String, String> strategyParams);
```

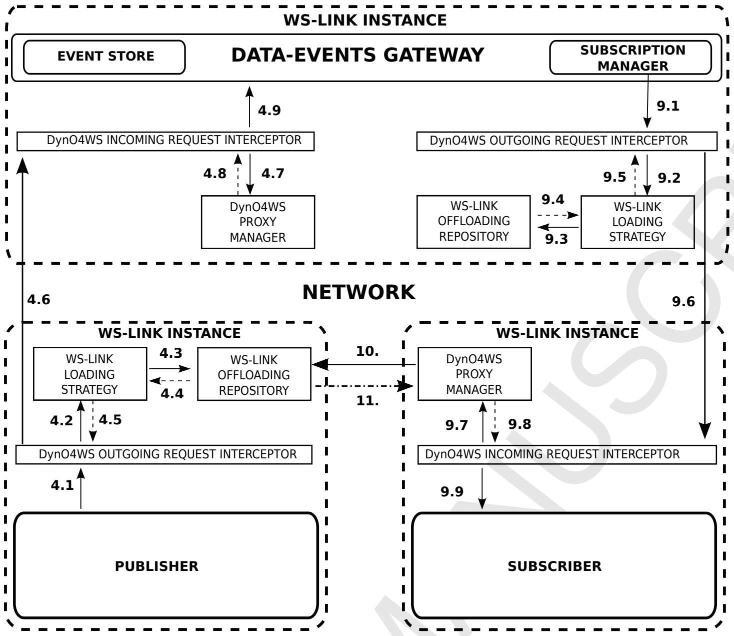


Figure 2: Detailed architectural view explaining how to enable transparent direct pub/sub data-transfers, by means of WS-Link.

The default value for this parameter is *"MAKE_COPY"*, since it guarantees that the receiving endpoint will obtain the same value as if no delayed transfers took place. If the attribute is not likely to change during the period between the service invocation and the attribute access, a *"KEEP_REFERENCE"* consistency value can be employed; in this case, a pointer to the attribute is kept until it is requested or the entity it belongs to gets garbage-collected on the receiver side. This last aspect can be handled by creating dynamic proxies for the exchanged entities and intercepting their garbage-collection. Finally, the *@Encoding* annotation allows for the customization of the serialization format (currently XML or JSON) as well as the possibility to enable message compression, as shown in Listing 3.

Listing 3: Configuration enabling message compression for the attachment transfer.

```
@Strategy(impl = PureLazyStrategy.class)
@Encoding(EncodingType.GZIP_XML)
private byte[] attachments;
```

4.3. WS-Link in action: configuration and runtime behaviour

It is worth to explain the role of the WS-Link framework inside the architecture depicted in Figure 1. For this purpose, Figure 2 gives more details about how we enable direct data-transfers. When the publisher raises an event, this is handled by the WS-Link framework, before it is sent to the gateway. In order to delay their transfer, we have configured the *attachments* attribute of our Event class, as shown in Listing 4. The WS-Link framework detects the annotated attributes [5] on the publisher-side and enacts the offloading process, according to the value of the declared annotations. Since the attachments should be transferred on-demand, WS-Link saves them in a local repository [6] and adds meta-data to the outgoing SOAP message, indicating how these attachments can be retrieved (interactions from 4.1 to 4.5).

Listing 4: WS-Link based configuration of attachments in the Event class.

```
@Strategy(impl = PureLazyStrategy.class)
@Consistency(ConsistencyType.MAKE_COPY)
@Encoding(EncodingType.XML)
private byte[] attachments;
```

When the event reaches the gateway (4.6), another instance of WS-Link decodes the SOAP message and instantiates a dynamic proxy that hides the "laziness" of the attachments transfer (4.7 to 4.9). This proxy holds the remote references to attributes and is kept until the event gets delivered to all the subscribers. The gateway does not need to inspect the attachments, so it will never trigger an attachment transfer. Before the gateway sends the event to a subscriber, WS-Link copies the meta-data inside the proxy to the outgoing SOAP message (9.1 to 9.5).

Listing 5 details how remote references are encoded into SOAP messages. The SOAP header refers to the invocation of *notify* performed by the gateway. Line 2 shows both the id of the current call and the one related to the message sender. Lines from 3 to 10 show how the event attachments can

[5]In this phase we group the attachments together in a single attribute, but more attributes can be added, by extending the Event class, to create custom events where each attachment can be managed in a different way.

[6]The "local" term should not be intended in a strict way. By default, the repository is deployed locally, but it can be shared among several remote instances of WS-Link, in order to take advantage of network topology and to keep the attachments available, if the publisher goes offline.

be retrieved. Attribute *cid* at line 4 refers to the call that generated the *attachments* attribute (the invocation of *notify* performed by the publisher). Attribute *host* at line 5 indicates the endpoint of the repository hosting the attachments. Attribute *name* at line 6 indicates the name of the field that has been offloaded. Finally, the id at line 7 prescribes how to query such repository, to retrieve the desired element.

The meta-data is finally decoded by the WS-Link instance at the subscriber-side. This process, whose details can be found in [35], is called *"Attribute Loading Delegation"*; in this case, the gateway delegates the publisher's repository to make the attachments available for subscribers. Once the event is delivered to the subscriber (9.6), its WS-Link instance creates a new proxy, that is able to retrieve the attachments, directly from the publisher (9.7 to 9.9). A typical subscriber-side logic should inspect the description of the event, in order to understand why such event arrived (e.g. by checking the topic).

Listing 5: SOAP header containing metadata for proxy instantiation, produced when the Gateway invokes the notify operation exposed by the subscriber.

```
1  <soap:Header>
2   <wslink cid="9050078" nid="Gateway">
3    <param name="0">
4     <field cid="1660014"
5            host="http://10.0.0.9:9999/DynO4WS/repo?wsdl"
6            name="attachments">
7      <id>{nid:Pub,cid:1660014,param:0,name:attachments}</id>
8     </field>
9    </param>
10   ...
11  </wslink>
12 </soap:Header>
```

Subsequently, it should check the attachment descriptors, to decide whether it is able to handle the attachments, and possibly inspect the attachments themselves. Each of these attributes can be accessed by invoking the corresponding getter method of the event class. When a getter method is invoked on a proxy generated by the *Proxy Manager*, the latter checks if the value has been already transferred to the local endpoint; if not, it triggers an invocation to a remote repository, according to the meta-data attached to the incoming SOAP message. In our case, when the *getAttachments* method, inside the event class, is invoked, the *Proxy Manager* retrieves the attachments from the repository at the publisher-side (10, 11).

Thanks to the dynamic proxing system, both publisher's and subscriber's business logic can take advantage of delegation, without modifying any line of code. The WS-Link framework transparently avoids useless transfers, thus increasing the efficiency of our pub/sub system as detailed in the next two sections.

5. Pub/sub patterns analysis

To understand the benefits of WS-Link and the underlying pub/sub model in application contexts, we analyze and characterize three main pub/sub patterns in terms of both messages exchange and delivery times. These patterns can be considered as composable micro-architectures for pub/sub applications, since most of these are written as a combination of the analyzed patterns. In particular, by using the patterns we compare the conventional model, where the Event Service (ES) fully mediates pub/sub communication, with the WS-Link model, where the event attachments are transferred at the edge of the infrastructure without any mediation between publisher and subscriber.

The selected patterns are:

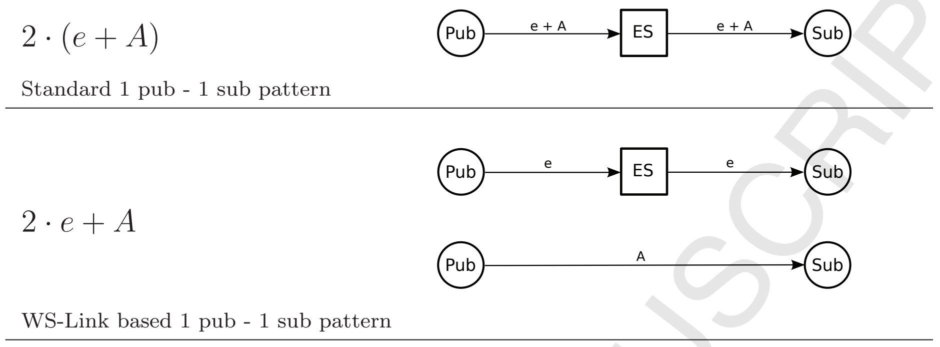
1. 1 publisher - 1 subscriber
2. 1 publisher - Many subscribers
3. Many publishers - 1 subscriber

It could be reasonably contended that selected patterns involving one-to-many and many-to-many events delivery may strongly benefit from IP Multicast. However, even though multicast reduces the number of messages in the network, it does not avoid message delivery to event subscribers that are not interested in the attachments. Besides, this scalable group disseminating primitive is not widely spread over the Internet [4], which is our target environment.

Pattern 1 takes place when two components are asynchronously coupled by means of one or more topics: only one subscriber is interested in receiving events from the considered publisher. In this scenario, by using the conventional model, the whole event $E = e + A$, where e is the event header and A is the set of attachments, is completely transferred from the publisher to the ES and, subsequently, from the ES to the subscriber.

Consequently, the total transferred data can be expressed as $2 \cdot (e + A)$. In a WS-Link based scenario, only the header e is transferred twice, while A

Figure 3: Comparison between standard and WS-Link based 1 pub - 1 sub pattern.



is moved directly from the publisher to the subscriber, only if the subscriber needs it. Consequently, the total transfer is expressed as $2 \cdot e + A$. The pattern is illustrated in Figure 3. The comparison clearly shows that the WS-Link based approach is effective when the size of A is large.

Pattern 2 takes place when many components subscribe for a single topic and all the events characterized by such topic are raised from the same component. This pattern is very frequent, for example, in e-Commerce enterprise systems, where the event concerning the creation of an order must be propagated to several subsystems handling aspects like billing, user profiling or shipping of goods. In this case, for a standard scenario, the event E is moved once from the publisher to the ES and one more time from the ES to each subscriber. The total transfer can be expressed as $(N + 1) \cdot (e + A)$, where N is the number of subscribers. With WS-Link, if all the subscribers need A, the total transfer can be expressed as $e + N \cdot (e + A)$ (see Figure 4). The comparison shows that, regardless of the size exhibited by A, when the value of N grows, the two approaches behave almost in the same way.

Pattern 3 consists of many publishers raising events that are captured by a single subscriber. This pattern is very common, for example, in Big-Data stream processing, where several streams must be composed and filtered by a single logical component. In a standard scenario, considering a single event raised by each publisher, we have N complete events moved from the N publishers to the ES and then to the subscriber. In this case, if we assume all attachments exhibit size A, the total data transfer can be expressed as $2 \cdot N \cdot (e+A)$. If a WS-Link scenario is considered, the total transfer is expressed as $N \cdot (2 \cdot e + A)$. This pattern is illustrated in Figure 5. The comparison shows

Figure 4: Comparison between standard and WS-Link based 1 pub - N sub pattern.

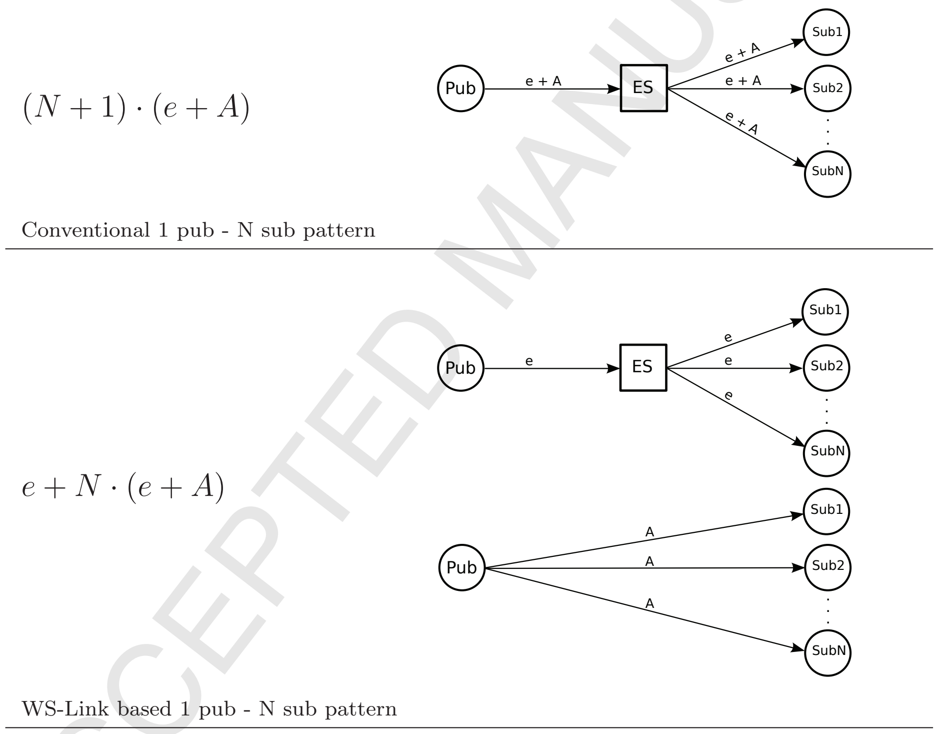
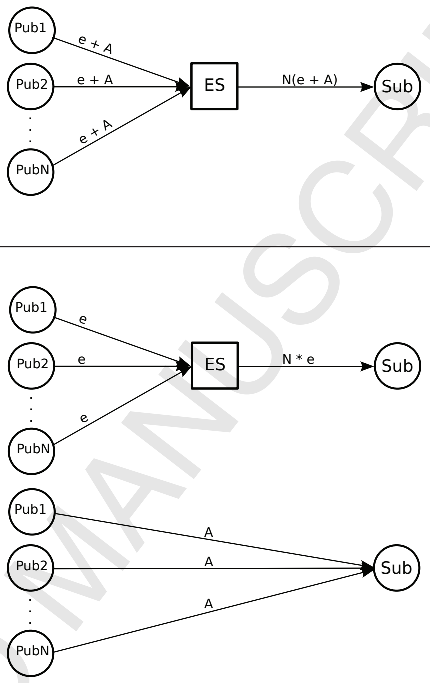

$(N+1)\cdot(e+A)$
Pub
e + A
ES
Sub1
Sub2
SubN
Conventional 1 pub - N sub pattern
$e+N\cdot(e+A)$
e
A
WS-Link based 1 pub - N sub pattern

Figure 5: Comparison between standard and WS-Link based N pub - 1 sub pattern.

$2 \cdot N \cdot (e + A)$



Conventional N pub - 1 sub pattern

$N \cdot (2 \cdot e + A)$

WS-Link based N pub - 1 sub pattern

that, even by considering a single event raised by each publisher, WS-Link is effective, especially when the size of A becomes considerable.

Once it is clear which data get transferred in each pattern, it is worth to characterize the conventional and WS-Link based models in terms of time $T_{delivery}$ needed to deliver a whole event $(e + A)$. To this end, we can rewrite the expressions from Pattern 1 by taking into account the SOAP level network latency L and throughput Th. For the conventional approach, the time to deliver an event E can be expressed as follows:

$$T_{delivery} = 2 \cdot (L + \frac{S_e + S_A}{Th}) \tag{1}$$

where S_e is the size of e and S_A the one characterizing A. Network latency L is assumed to be the same on each hop, thus it is counted twice, since the scenario involves two messages.

With WS-Link $T_{delivery}$ can be expressed as follows:

$$T_{delivery} = 4 \cdot L + \frac{2 \cdot S_e}{Th} + \frac{S_A}{Th} \tag{2}$$

In this case, latency L impacts four times on $T_{delivery}$ because the model adopted by WS-Link enforces the exchange of four small messages: two for delivering the header e and two more for asking the delivery of A. Since the size S_A is usually much larger than S_e and given that the WS-Link scenario requires one attachment transfer instead of two, the value of expression 2 should be smaller than the one from expression 1. Anyway, expression 2 does not take under consideration the overhead $T_{middleware}$ added by WS-Link (mainly due to the message exchange model adopted), which has been extensively analysed in [35] and is mainly related to the process that is performed to write the set of attachments inside the Offloading Repository and to eventually retrieve it.

By taking this aspect under consideration and combining equations 1 and 2, we obtain that the performance improvement brought by the WS-Link based approach is preserved if $T_{delivery(wslink)} + T_{middleware} < T_{delivery(standard)}$, that is:

$$T_{middleware} < \frac{S_A}{Th} - 2 \cdot L \tag{3}$$

In the next section we empirically evaluate whether the improvements achieved through the proposed approach can be preserved or the $T_{middleware}$ overhead is too high for the considered patterns. Moreover, it is worth to note that in Pattern 1, by using the conventional pub/sub model, the event throughput could benefit from the parallel communication that characterizes the two logical links from the publisher to the ES and from this one to the subscriber. The ES behaves as a store and forward message router with the ability of overlapping the communication of E_i from the ES to the subscriber with the communication of the next event E_{i+1} from the publisher to the ES. However, even in this extreme case our approach theoretically outperforms the conventional one since the delivery time of the last event, with this model, is not overlapped.

6. Performance Evaluation

In this section, we evaluate the proposed approach by comparing the performance obtained by a WS-Link based ES and the ones characterizing an ES based on the Apache Camel integration framework [12], which is an open source solution widely adopted in the industry.

Camel allows for the specification of message routing rules by means of domain-specific languages like its Java-based Fluent API, Spring XML, Scala, and it can work with any kind of transport such as HTTP and ActiveMQ [29], by means of URIs. Camel can be easily employed to implement the publish-subscribe interaction style. It also provides components to automatically expose an endpoint as a Web service. Combining these two features, Camel can be used to create an event-service where subscribers, analogously to the WS-Notification standard, are implemented as Web services.

The evaluation has been organized in three phases. First, we compare the 90th percentile of the delivery time for Pattern 1, in order to verify whether the overhead introduced by the WS-Link framework dominates the improvements introduced by the proposed approach, as we have outlined in Section 5.

In the second phase, we compared the measurements concerning the application level throughput, in terms of events per second, for Patterns 1, 2 and 3.

Thirdly, we evaluated the overhead related to the attachments offloading process, when a persistent store is used. This is important since, as we mentioned in Section 3, persistence is a desirable feature provided by MOMs that ensures pub/sub semantics.

All the tests have been performed on the Grid5000 infrastructure, by deploying each publisher and each subscriber on a different node. To ensure a fair workload distribution, subscribers register themselves to be notified about every event injected in the system. The goal is to measure the benefit of our solution when the pub/sub layer is working at its maximum capacity. For patterns $1-N$ and $N-1$ we have chosen to stress the system with the maximum number of nodes available. Each machine is equipped with two Intel Xeon E5520 processors @ 2.27 GHz and 32 GB of memory. The bandwidth on each link has been limited to 100Mb/s, in order to emulate a realistic Internet scenario. Delivery time meaurements have been computed from a set of 100 runs, for each attachment size. Throughput measurements have been obtained by using a synthetic workload characterized by a parametric delay between events. The measurements have been repeated for different attachment sizes (1, 5, 10, 25 and 50 MBs). For each size, a number of events has been emitted until the throughput has become stable, whereas the inter-events delay has been changed in order to maximize the

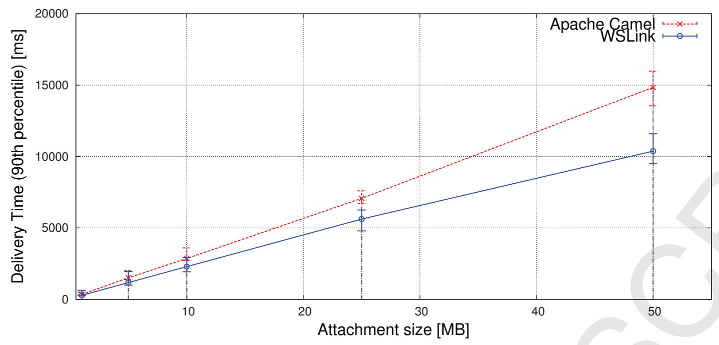


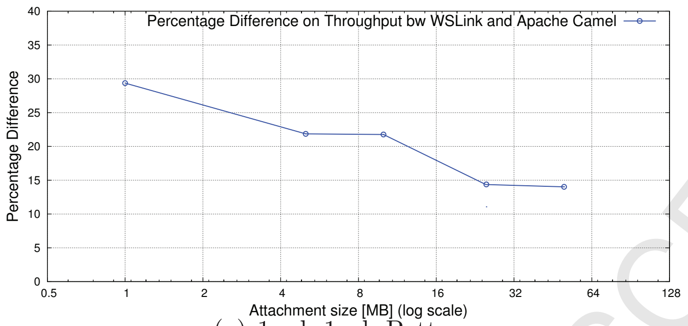
Figure 6: Comparison between WS-Link and Camel event delivery time (90th percentile) for the 1 pub - 1 sub pattern.

throughput.

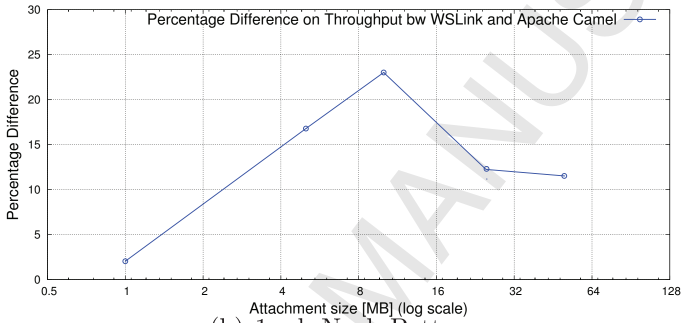
The results from the first phase are summarised in Figure 6. When the attachments size is negligible (i.e. 1MB) the WS-Link approach behaves similarly to the Camel based one. However, when the attachment size grows, WS-Link outperforms Camel by exhibiting an improvement of 30%. This is consistent with the expression 3 in Section 5, where the main term consists of the attachment-size/throughput ratio.

The second phase has been organized into three sub-phases, in order to analyze all the patterns introduced in Section 5. The experimental results are depicted in Figure 7. More precisely, Figure 7(a) illustrates the results concerning Pattern 1. Here it can be clearly observed how the attachment size impacts the throughput: when small attachments are considered, WS-Link outperforms Camel, because there are several 1MB sized messages that get transferred once (from the publisher to the subscriber) instead of twice. When the attachment size grows, the throughput difference becomes less significant, because the link between the publisher and the subscriber gets overloaded. With the Camel approach, the ES behaves as store-and-forward router, thus the transfer, from the ES to the subscriber, of event E_i can be overlapped with the transfer, from the publisher to the ES, of event E_{i+1}. Nevertheless, in the 50MB sized attachment test, WS-Link still outperforms Camel by 14%.

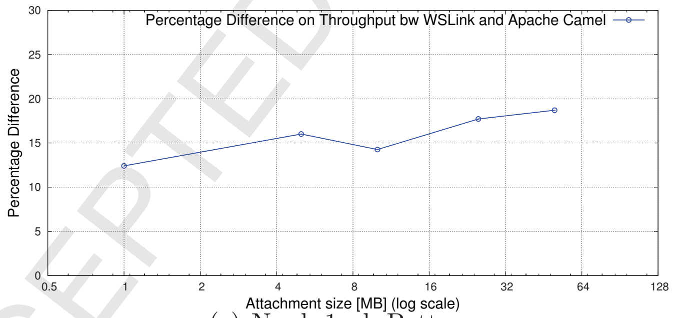
Figure 7(b) describes the results concerning Pattern 2. In this test, we have deployed 10 subscribers requesting attachments from the same publisher. In Section 5, we have defined this as the worst case for the WS-Link approach; this is confirmed, especially when small attachments are trans-



(a) 1pub-1sub Pattern



(b) 1pub-Nsub Pattern



(c) Npub-1sub Pattern

Figure 7: Comparison of events throughput for the considered patterns.

ferred. More surprisingly, performances increase with medium sized attachments (i.e. 10MB). This behaviour is due to the overload of the network interface of the ES. In fact, in order to preserve event ordering, the ES must first deliver event E_i to every subscriber and then it can proceed with event E_{i+1}.

With the WS-Link approach, events get delivered quickly and the attachments are recovered asynchronously. This way, the overall system benefits of a higher level of concurrency and the throughput becomes higher. Finally, when bigger attachments are considered for the WS-Link approach, the network interface at the publisher side becomes overloaded, but, since there is a higher level of concurrency, the overall performances are still better than the ones obtained with the conventional approach.

Figure 7(c) illustrates the results for Pattern 3. In this case, we have deployed 10 publishers and 1 subscriber catching all the events that are sent. In Section 5, we described this as the best case for the WS-Link approach. The results confirm this analysis, especially when the size of attachments grows. More specifically, when small attachments are transferred, the performances are not much better than the ones obtained with Camel, since the latter can take advantage of concurrency, by overlapping the delivery of messages as for Pattern 1. Moreover, the WS-Link approach requires 4 network messages, instead of 2, in order to deliver a complete event. When bigger attachments are considered, the network delay becomes less significant. Finally, neither the ES nor the publishers get overloaded, since the load is shared among them.

Finally, in a last experiment we evaluated the influence of using a persistent repository with the WS-Link framework. The reasons of storing events in a sustainable manner are multiple. The main reason is about the moment at which a subscriber wants to retrieve the attachments for the event it has received. It may have passed several time units before a subscriber decides to fetch the offloaded attributes. Thus, according to the nature of the events that are published, and the garbage collection policy applied, the attachments may not fit in the main memory of the WS-Link instance at the publisher side. Table 1 summarizes the comparison we have made between in-memory and widely used storage engines. Each value contained by a cell represents respectively the overhead in time (in ms) for storing or retrieving attachments in the considered persistent engine with respect to the in-memory solution. It shows how much additional time is required to deliver a complete event.

For this experiment, the MongoDB engine has been configured to work with its GridFS extension, in order to support attachments bigger than 16MB. The results show that MongoDB achieves different write performances according to the configuration of write concerns (WC). The *acknowledged* one ensures a client receives an ack for each submitted write operation. However, no guarantee is given regarding the robustness of the write operation. In the

	MongoDB (Acknowledged WC)		**MongoDB (Journaled WC)**		**MySQL**	
	Write	*Read*	*Write*	*Read*	*Write*	*Read*
1MB	138	122	346	122	260	126
5MB	900	612	1500	611	1302	623
10MB	1920	1200	2795	1202	2537	1215
25MB	4871	2988	6785	2993	6345	3087
50MB	10424	6009	16201	5996	12756	6173

Table 1: Overhead in time (in ms) of persistent compared to in-memory datastores for storing and retrieving attachments.

contrary, the *journaled* WC guarantees the write operation can survive to a failure. This difference of semantic explains the higher difference for writing attachments when using the *journaled* WC. On the other hand, MySQL is a better approach to store attachments with the same guarantees as MongoDB, by using a *journaled* WC, but at the price of a slightly slower retrieval time. Consequently, the choice really depends on the scenario and the considered context.

Finally, the percentage difference for writing attachments between persistent engines and in-memory solution remains almost stable (∼174% for MongoDB journaled WC, and ∼172% for MySQL). In other words, the time required to store (write) binary payloads with persistent engines follows the same pattern as the one for in-memory when the size of attachments increases. Moreover, even though the percentage difference for retrieving (read) attachments rises with the size of the attachments, the effect could be balanced by using an in-memory cache mechanism. Thanks to such cache, we could avoid the rising overhead of the read operation.

7. Related Work

Merging EDA, and SOA has been and still gaining increased attention from researchers and industry [19, 9, 8, 1, 25, 10]. The combination between EDA and SOA can be benefical in order to provide complex distributed applications (e.g., finance applications scenario, business process managment, RFID, manufactory, military application scenarios) with a complete dependable and flexible middleware system [1].

Eugester *et al.* [7] give a comprehensive survey about the pub/sub communication paradigm. A lot of works focus on the adoption of the pub/sub paradigm itself for web services [7, 27, 25]. Nevertheless, most of the efforts focus more on the definition of protocols rather than on improving the efficiency of data exchanges.

In [27], the authors propose to combine pub/sub and Web-Services to overcome the request/response model. To carry out a data-transfer, the publisher notifies the subscriber about the availability of data to be retrieved through a further and explicit request/response interaction. In our work, we also rely upon the request-response mechanism, but we make it completely transparent, by means of dynamic proxies.

In [28] the authors provide a system based on their *DSProxy* as a cross-platform solution. It provides store-and-forward capability to SOAP messages, by applying compression of SOAP and XML and facilitating the traversal of multiple heterogeneous networks. This system employs an overlay network of DSProxys and polling capabilities to store and forward messages. The authors focus more on providing a solution that complies with the web service standards rather than explicitly addressing huge volumes of SOAP messages or attachments, as we do.

The authors in [13] offer an experimental pub/sub infrastructure to be used with Web-Services enabled applications. They demonstrate that the performance gap between traditional event-based technologies and the Web-Services based approach is not necessarily significant. They emphasize the decoupling characteristics of the Web-Services and event-based design, and they propose a pub/sub infrastructure compliant with the WS-Notification standard. In any case, this solution does not take into account the nature of the data exchanged between services by means of the pub/sub mechanism. This aspect is extensively addressed in our work.

Nevertheless, how to deliver huge data among Web-Services by means of pub/sub, remains an unanswered question in the scientific community. Data-centric pub/sub has been addressed by the *Data Distribution Service (DDS)* OMG standard [20] and by some related research papers [24, 15]. The purpose of the specification is to provide a common application level interface that defines the data-distribution service. The DDS architecture relies on the management of a global data space and consists of two layers: (1) the data-centric pub/sub (DCPS) layer that provides APIs to exchange data based on specified QoS policies and (2) the data local reconstruction layer (DLRL) that makes data appear local. The specification focuses on the quality of services for data delivery in distributed environments. Many works implement the DDS specification for real time control systems [24, 18, 17].

In [15], the authors propose a data propagation model based on the DDS pub/sub specification to provide data persistence in distributed environments. The framework guarantees the persistence by transferring data samples through relay nodes to the receiving nodes that have not participated in the data distribution network at the data sample distribution time. However, since the DDS approach requires the data to be transferred to a global data space, managing global data might introduce some performance degradation due to distribution unawareness. This problem should be addressed by optimizing the distribution of the global space according to data consumption, similarly to the approach in [32] to guide the deployment in a distributed environment. The authors in [32] propose clear semantics to program object-based distributed computing by introducing a light container, called micro-object. They handle the mutable part of micro-objects through tokens (references) that can be distributed over the network. In that sense, the proposed technique is similar to ours, since the object payload is transferred only when needed. However, the contexts of the two works are different. We do not introduce a new programming model, but we focus on the application of a similar technique (lazy transfer) to efficiently distribute messages in existing infrastructures for Web services based asynchronous communication, such as ESBs. Therefore, since our messages are only data to transfer, we do not observe impacts over common operations that are present in object-based programming (delete, update, etc.).

Data-intensive applications has been a subject of research to integrate event driven interactions. In [16], the authors present an event driven architecture that integrates the content-based event notification system with workflow management. They address data-driven applications which dynamically discover, ingest data from, and interact with other applications. The main contribution of this work is to be able to reconfigure the workflow on the fly on the reception of important events from both external systems and from computations. The work focuses more on the integration of event-driven technology on such workflows and on the behavioral aspects of the workflow management rather than on the data transfer efficiency between workflow activities.

8. Conclusion

We have presented an architectural model, a middleware and some annotation based mechanisms integrated in the WS-Link framework to allow for efficient, data-intensive, event-driven communication.

The experimental evaluation showed that we can reduce the average event delivery time, compared to a fully mediated solution, but we are planning to improve our results, by optimizing the serialization process for large attachments. In fact, since the proposed model prescribes attachments to be stored and then sent to the subscriber on-demand, a specific delay is added to the critical path of the whole event delivery.

To overcome this drawback, we are implementing a caching mechanism enabling fast access to the attachments that are stored in the repository. By concurrently storing attachments into the in-memory cache and into the persistent repository, the framework will be able to fastly handle attachment accesses by using the cache (which is loaded every time the framework restarts after a crash).

Finally, we want to add autonomicity features to our middleware, so that repositories can be shared, replicated or migrated, based on run-time workload, in order to improve scalability, availability and fault-tolerance.

Despite this work does not focus on scalability aspects, it is worth noticing that the proposed architecture can scale horizontally by replicating gateways. Consider a scenario including X external subscribers for topic T; in this scenario, if an event related to T occurs, it is dispatched to all the subscribers in $O(X)$ messages, if only one gateway has been deployed and a round-robin approach is assumed. Anyway, if Y gateways are deployed and the subscribers are uniformly assigned to all of them, the event is dispatched in $O(X/Y)$ messages, since different gateways can serve different sets of subscribers, in parallel.

Starting from the promising results achieved for the event delivery efficiency, in the future, a more in depth analysis of scalability of WS-Link and in particular of the underlying model will be performed.

Acknowledgment

This work is supported by Province of Benevento, EU FP7 STREP project PLAY and French ANR project SocEDA.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see `https://www.grid5000.fr`).

References

[1] R. Baldoni, R. Beraldi, G. Lodi, L. Querzoni, Combining Service-Oriented and Event-Driven Architectures for Designing Dependable Systems, Foundations of Computing and Decision Sciences 35 (2010) 77–90. URL: `http://www.dis.uniroma1.it/~midlab`.

[2] K. Banker, A Document Database for the Modern Web, Manning, 2011.

[3] J. Bloch, JSR 175: A metadata facility for the Java programming language, `http://jcp.org/en/jsr/detail?id=175`, 2004.

[4] C. Diot, B.N. Levine, B. Lyles, H. Kassem, D. Balensiefen, Deployment issues for the ip multicast service and architecture, Network, IEEE 14 (2000) 78–88.

[5] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[6] T. Erl, SOA Design Patterns, 1st ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[7] P.T. Eugster, P.A. Felber, R. Guerraoui, A.M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2003) 114–131.

[8] F. Facca, S. Komazec, M. Zaremba, Towards a semantic enabled middleware for publish/subscribe applications, in: Semantic Computing, 2008 IEEE International Conference on, pp. 498 –503.

[9] I. Filali, F. Bongiovanni, F. Huet, F. Baude, A survey of structured p2p systems for rdf data storage and retrieval, Transactions on Large-Scale Data-and Knowledge-Centered Systems III (2011) 20–55.

[10] S.Y. Ghalsasi, Critical success factors for event driven service oriented architecture, in: Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human, ICIS '09, ACM, New York, NY, USA, 2009, pp. 1441–1446.

[11] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[12] C. Ibsen, J. Anstey, Camel in action, Manning Publications Co., 2010.

[13] B. Kowalewski, M. Bubak, B. Baliś, An event-based approach to reducing coupling in large-scale applications, in: Proceedings of the 8th international conference on Computational Science, Part III, ICCS '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 358–367.

[14] O. Lassila, R. Swick, et al., Resource description framework (rdf) model and syntax specification, 1998.

[15] C. Lee, J. Hwang, J. Lee, C. Ahn, B. Suh, D.H. Shin, Y. Nah, D.H. Kim, Self-describing and data propagation model for data distribution service, in: Software Technologies for Embedded and Ubiquitous Systems, volume 5287 of *LNCS*, Springer Berlin / Heidelberg, 2008, pp. 102–113.

[16] C. Lee, B. Michel, E. Deelman, J. Blythe, From event-driven workflows towards a posteriori computing, in: V. Getov, D. Laforenza, A. Reinefeld (Eds.), Future Generation Grids, Springer US, 2006, pp. 3–28.

[17] M.A. Mastouri, S. Hasnaoui, Performance of a publish/subscribe middleware for the real- time distributed control systems, IJCSNS International Journal of Computer Science and Network Security 7 (2007).

[18] J.D. Mitchell, M.L. Siegel, M.C.N. Schiefelbein, A.P. Babikyan, Applying publish-subscribe to communications-on-the-move node control, MIT Lincoln Laboratory Journal 16 (2008) 413–430.

[19] S. Overbeek, M. Janssen, P. Bommel, Designing, formalizing, and evaluating a flexible architecture for integrated service delivery: combining event-driven and service-oriented architectures, Service Oriented Computing and Applications 6 (2012) 167–188.

[20] G. Pardo-Castellote, Omg data-distribution service: Architectural overview, in: ICDCSW, p. 200.

[21] L. Pellegrino, F. Huet, F. Baude, A. Alshabani, A distributed publish/-subscribe system for rdf data, in: Proceedings of 6th International Conference on Data Management in Cloud, Grid and P2P Systems (Globe 2013).

[22] E. Prud'Hommeaux, A. Seaborne, Sparql query language for rdf, W3C working draft 4 (2008).

[23] C. Richardson, Untangling enterprise java, Queue 4 (2006) 36–44.

[24] M. Ryll, S. Ratchev, Towards a publish / subscribe control architecture for precision assembly with the data distribution service, in: Micro-Assembly Technologies and Applications, volume 260 of *IFIP International Federation for Information Processing*, Springer Boston, 2008, pp. 359–369.

[25] A. Sengupta, V. Nandey, S. Sengupta, Etdsoa: a model for event and time driven service oriented architecture, SIGSOFT Softw. Eng. Notes 35 (2010) 1–9.

[26] A. Shalloway, J. Trott, Design patterns explained : a new perspective on object-oriented design, Addison-Wesley, Boston, Mass., 2004.

[27] I. Silva-Lepe, M. Ward, F. Curbera, Integrating web services and messaging, in: Web Services, 2006. ICWS '06. International Conference on, pp. 111 –118.

[28] E. Skjervold, T. Hafsø ande, F. Johnsen, K. Lund, Enabling publish/-subscribe with cots web services across heterogeneous networks, in: Web Services (ICWS), 2010 IEEE International Conference on, pp. 660 –668.

[29] B. Snyder, D. Bosnanac, R. Davies, ActiveMQ in action, Manning, 2011.

[30] A.S. Tanenbaum, M.v. Steen, Distributed Systems: Principles and Paradigms (2Nd Edition), Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[31] H. Taylor, A. Yochem, L. Phillips, F. Martinez, Event-Driven Architecture: How SOA Enables the Real-Time Enterprise, 1 ed., Addison-Wesley Professional, 2009.

[32] J.M.S. Wams, M. van Steen, Simplified distributed programming with micro objects, in: FOCLASA, pp. 1–15.

[33] Q. Zagarese, G. Canfora, E. Zimeo, Dynamic object offloading in web services, in: K.J. Lin, C. Huemer, M.B. Blake, B. Benatallah (Eds.), SOCA, IEEE, 2011, pp. 1–8.

[34] Q. Zagarese, G. Canfora, E. Zimeo, I. Alshabani, L. Pellegrino, F. Baude, Efficient data-intensive event-driven interaction in soa, in: S.Y. Shin, J.C. Maldonado (Eds.), SAC, ACM, 2013, pp. 1907–1912.

[35] Q. Zagarese, G. Canfora, E. Zimeo, F. Baude, Enabling advanced loading strategies for data intensive web services, in: C.A. Goble, P.P. Chen, J. Zhang (Eds.), ICWS, IEEE, 2012, pp. 480–487.