Implementing a High Level Pub/Sub Layer for Enterprise Information Systems

Mario Antollini
Faculty of Sciences, UNICEN, Tandil, Argentina
manto@exa.unicen.edu.ar

Mariano Cilia*
Databases and Distributed Systems Group, Technische Universität Darmstadt, Darmstadt, Germany
cilia@dvs1.informatik.tu-darmstadt.de

Alejandro Buchmann
Databases and Distributed Systems Group, Technische Universität Darmstadt, Darmstadt, Germany
buchmann@dvs1.informatik.tu-darmstadt.de

Keywords: data heterogeneity; data integration; semantic metadata; data dissemination; publish/subscribe; notification services

Abstract: Enterprise application interactions based on events has been receiving increasing attention. It is based on the exchange of small pieces of data (called events) typically using the publish/subscribe interaction paradigm. Most pub/sub notification services assume a homogeneous namespace and do not support the interaction among heterogeneous event producers and consumers.
In this paper we briefly describe the concept-based approach as a high-level dissemination mechanism for distributed and heterogeneous event-based applications. We focus on the design and implementation issues of such a mechanism and show how it can be integrated on research prototypes or products and platforms.

1 Introduction

The global economy is causing drastic changes in the way business is conducted. On the organizational level, company mergers and acquisitions are leading to a consolidation of manufacturers and producers. At the same time goods are being offered across traditional borders regardless of economic, cultural or linguistic differences. To function properly, global electronic commerce requires the correct interpretation of exchanged data regardless of its origin and place of consumption.

Notice that the exchanged data encapsulate information about a given happening of interest, which can only be properly interpreted and used when sufficient context information is known. In traditional centralized systems, this context information is typically known by the users/developers and left implicit. When data and events are exchanged across component or institutional boundaries contextual information is usually lost. To process this data in a semantically meaningful way, explicit information about the semantics of events and data is required.

Event-based systems need an event dissemination mechanism to deliver relevant events to interested consumers. The publish/subscribe interaction paradigm has been gaining relevance for this purpose. It basically consists of a set of clients that asynchronously exchange events[2] decoupled by a notification service[3] (*NS* for short) that is interposed between them. Clients can be characterized as producers or consumers. Producers publish notifications, and consumers subscribe to notifications of interest by issuing subscriptions, which are essentially stateless message filters. After a client has issued a subscription, the notification service is responsible for delivering all future matching notifications that are published by any producer until the client cancels the respective subscription.

To the best of our knowledge (see Section 2 for details), almost all publish/subscribe mechanisms are restricted to expose the data structure of events to participants. This reflects a low level support for event consumers that based on this scarce information must express their interest without having neither a concrete definition of meaning nor explicit assumptions made by event/data producers. Without this kind of information event producers and consumers are expected to fully comply with implicit assumptions made by participating software components or

*Also Faculty of Sciences, UNICEN, Tandil, Argentina.

[2] In the context of this work the terms notification, message and event are used interchangeable to mean basically the same thing.

[3] Throughout this work the terms notification service and messaging service will be interchangeably used.

applications. Even in the cases of a very small set of applications within an enterprise this approach is questionable.

Since publish/subscribe mechanisms decouple producers and consumers, they should share a common understanding in order to express their mutual interests. In other words, events must be understandable beyond the closed confines of a single component or application. That includes applications that interact across traditional borders regardless of economic, cultural or linguistic differences (in its simplest form, i.e., system of units, currency or date/time format). Since the source of an event (in other words a publisher) cannot anticipate who is interested in a given event and when and where it must be delivered, a higher-level publish/subscribe infrastructure is needed. This basically adds to the current pub/sub mechanisms the following two requirements: a) the usage of a common vocabulary for defining interests, and b) the correct interpretation of information independently of its origin and place of consumption.

The *Concept-based addressing* was initially proposed as a part of a data dissemination mechanism in the context of Internet auctions (Bornhövd et al., 2000) and later in the context of reactive systems in distributed environments (Cilia et al., 2001). Based on this experience a higher level of abstraction to describe the interests of heterogeneous event producers and consumers was proposed. This is achieved by supporting from the ground up ontologies which provide the base for correct data and event interpretation.

Rather than requiring every producer or consumer to use the same homogeneous namespace (as is common in other publish/subscribe systems) we provide metadata and (extensible) conversion functions to map from one context to another. This last feature allows event consumers to simply specify the context to which events need to be converted before they are delivered for client processing.

In this paper we focus on the design and implementation issues of such an abstract layer and show how it can be integrated on research prototypes (such as Rebeca) and (commercial) products and platforms (like, JMS and J2EE). We do not aim to analyze its performance since its goal is to show the implementation of a novel approach that provides heterogeneous applications with a seamless way to interact and correctly interpret their exchanged data.

The rest of this paper is organized as follows. In section 2, background and related work is presented. Then, we describe our proposed approach in section 3. The main design decisions are described in section 4 while details about the implementation are presented in section 5. Finally, conclusions and future work are presented.

2 Background and Related Work

Since this work deals with various issues, like event dissemination, the publish/subscribe paradigm, distributed systems, this section serves as a review of these topics including related research.

2.1 Data Dissemination

Notification services are widely used to deliver events/data of interest to the corresponding consumers. A notification service is an application-independent infrastructure that supports the construction of collaborating systems (event-based systems), whereby producers of events publish event notifications to the infrastructure and consumers of events subscribe with the infrastructure to receive notifications of interest.

There are several research projects (e.g. SIENA (Carzaniga, 1998), REBECA (Mühl, 2001), CEA (Bacon et al., 1998), JEDI (Cugola et al., 1998), READY (Gruber et al., 1999), standard specifications (i.e., CORBA event service (Object Management Group, 1997), Java Message Service (Hapner et al., 1999)) and products (TIBCO Inc., 1996; IBM, ; Fiorano, ; Talarian, ; SonicSoftware, ; SpiritSoft,) that focus on different aspects of data dissemination.

The pub/sub model is a very general communication model to exchange data among participating applications commonly used in distributed environments. Its main features can be characterized by:

- *Natural Multicast Functionality*. Within the publish/subscribe model one application can send a message once and multiple participating applications receive it. This model contains basically two main players: data *producers* and *consumers*.
- *Event Filtering*. Some notification services also offer filtering facilities, which are based on clients' subscriptions (i.e., server-side filtering), avoiding irrelevant events to be forwarded to uninterested consumers.
- *Decoupling Producers and Consumers*. Producers and consumers of messages are anonymous to each other, so the number of publishers and subscribers may dynamically change. Individual publishers and subscribers may evolve without disrupting the existing system, facilitating extensibility and flexibility.
- *Mediator*. A NS responsible for finding and delivering matching notifications among publishers and subscribers.

A fundamental aspect of publish/subscribe systems is the expressiveness of the notification selection,

i.e, how consumers specify subscriptions. The publish/subscribe interaction offers four generic alternatives (or models) in order to address messages.

Channel-based addressing: The first generation of publish/subscribe systems (Harrison et al., 1997; Object Management Group, 2000; Sun Microsystems, Inc., 1998) used *channels*. Here, the producer publishes notifications into specific channels. Consumers subscribe to channels and get all notifications published on them.

Subject-based addressing: In *subject-based* pub/sub systems (Oki et al., 1993; TIBCO Inc., 1996) producers publish notifications attaching to them certain additional information, called subject, that usually synthesizes notifications' content. Subjects are arranged in a *subject tree* by using a dot notation, and clients can either subscribe to a single subject (e.g., `finance.quotes.NASDAQ.FooInc`) or use wildcards (e.g., `finance.quotes.NASDAQ.*`).

Topic-based addressing: This is similar to the idea of channels (here called *topics*) (Haase, 2002; Happner et al., 1999). Messages can carry additional properties that could represent the content of the message in question. Consumers express their interest by subscribing to a topic but they can additionally include a predicate (called *message selectors*) that refer to message properties.

Content-based addressing: It allows subscriptions to express interest based on the content of notifications, providing a more powerful and flexible notification selection (Aguilera et al., 1999; Mühl, 2001) than the others.

2.2 Dealing with Heterogeneity

The need for additional semantic metadata for the exchange of data or messages among independent entities or services has been clearly identified, not only in the context of B2B frameworks like ebXML (Eisenberg and Nickull, 2001), BizTalk (Microsoft Corp., 2000), or RosettaNet (RosettaNet, 2002) but also by the W3C in efforts like Semantic Web (Berners-Lee et al., 2001), or DAML+OIL (D. Conolly et al., 2001).

XML (Bray et al., 1998) and XML Schema (Fallside, 2001) are used to define common vocabularies to describe data and business processes. Other data models similar to XML include OEM (Papakonstantinou et al., 1995) and the models described in (Abiteboul et al., 1997; Deutsch et al., 1999).

For the representation of content of messages we use MIX (*Metadata based Integration model for data X-change*) (Bornhövd and Buchmann, 1999; Bornhövd and Buchmann, 2000; Bornhövd, 2000), a data model that combines the use of tags or attribute names from a common vocabulary with additional meta information that describes the corresponding data. MIX can be understood as a self-describing data model for data exchange since information about the structure and semantics of the data is given as part of the exchanged data itself.

The MIX model is based on the concept of a `SemanticObject`, together with its underlying `SemanticContext` which consists of a flexible set of meta-attributes that explicitly describe the assumptions about the meaning of the data item. Each semantic object has a concept label associated with it that specifies the relationship between the object and the real world aspects it describes. These labels have to be taken from a commonly known vocabulary, or ontology (Bornhövd, 1999).

In the MIX representation, simple attributes are represented as triplets of the form $<C; v; S>$, with C referring to a concept from the underlying ontology, v standing for the actual data value, and S representing the semantic context of v. The association of context information with a given data value serves as an explicit specification of the meaning of the data.

3 Proposed Approach

Originally introduced in (Bornhövd et al., 2000) for a specific application and with an initial implementation on top of a commercial pub/sub product (Cilia et al., 2001), the *concept-based approach* was proposed to tackle the problem of event-based interaction among heterogeneous cooperating applications. On one hand, it concentrates on resolving data interpretation problems. On the other hand, it provides a pub/sub abstraction that can run on top of (various) notification services independently of the addressing model used underneath.

The concept-based approach provides an abstraction where participating applications do not need to care about details of the underlying delivery mechanism. In this sense, applications involved in such interaction can replace this mechanism by relying on the concept-based layer causing minimal changes on collaborating applications if any.

3.1 High-Level Architecture

The concept-based approach is not intended to be a whole new notification service by itself, but a layer of software capable of running on top of different existing notification services, even supporting different addressing models. From an abstract point of view, it mediates in every subscription or publication that

clients issue and it delivers messages to the underlying notification service. As a matter of fact, clients do not directly communicate with the underlying notification service anymore, moreover they don't even know which or what kind of NS is running below the concept-based layer.

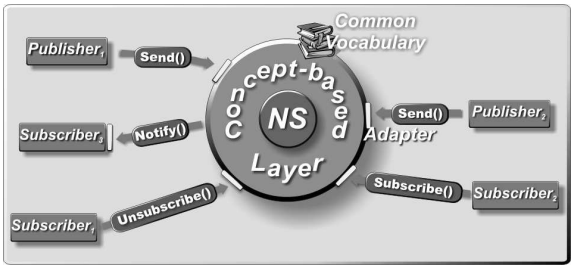

Figure 1: Concept-based high-level architecture

Figure 1 sketches this approach. The underlying NS is used as a data delivery mechanism where the concept-based layer is responsible for providing a higher-level interaction among heterogeneous applications (Cilia et al., 2004). A *vocabulary/ontology manager* is associated with this layer with the purpose of handling domain-specific vocabularies used by the participant applications. *Adapters* are the intermediaries between pub/sub clients and the concept-based layer. At the publisher side, they are responsible for resolving vocabulary issues and for enriching message content, i.e. with contextual information. At the subscriber, side they are in charge of transforming/converting message content according to the subscriber context. Additionally, this layer is responsible for mapping concept-based data into the data structures and addressing models of the delivery mechanism underneath.

3.2 Handling Heterogeneity of Exchanged Data

In order to propose a solution to the problem of data heterogeneity (or data integration) in the context of cooperating applications (particularly relying on pub/sub systems), several crucial issues are needed:

- the management of a shared vocabulary used by participating applications,
- a matching vocabulary which is a subset of the shared one used internally by the pub/sub system for matching purposes,
- a vocabulary agreement module that maps from the shared to the matching vocabulary,
- explicit definition of contextual information of all participants within the pub/sub system, and
- conversion functions to map from and to different contexts.

Making the vocabulary and the contextual assumptions explicit, events produced at source applications can be correctly interpreted and converted to the required target context by diverse consumers.

3.3 Context Transformation

Adapters are the contact of applications with the notification service. They are responsible for the mapping operations from local context to the matching one and vice versa.

Adapters are responsible for converting messages' context to the context used inside NSs boundaries (known as default context). This is done based on the explicit specification of clients about their contextual information. As soon as a message arrives to the concept-based layer, it is automatically converted to the matching context and then delivered to the underlying NS. In this way, NSs' internal matching operations are always carried out over expressions and messages expressed in the same context.

Before events arrive at the subscriber side, adapters automatically convert message's context to the one specified by subscribers at subscription time. In this way, consumers receive data converted according to their contextual preferences and no further conversion is needed at the client application.

4 Design of the Concept-based Pub/Sub Layer

This section explains how the concept-based layer was designed, exposing in detail the way subscriptions and publications are represented through the concept-based layer.

4.1 Integrating MIX with Participating Applications

The MIX model plays the role of a common vocabulary into the concept-based architecture, as the basis for the correct interpretation of events coming from different sources. This important component is the shared mean between publishers and subscribers in our concept-based implementation. Relying on it, heterogeneous applications can interact seamlessly.

Figure 2: The role of MIX among concept-based clients

As can be seen in Figure 2, MIX is basically composed of four sub-components (i.e., *conversion and matching operations*, *events*, *expressions* and *semantic context*). Participating applications take advantage of the functionality provided by MIX using the appropriate sub-components. Publishers create events and enhance them with the available semantic information. On the other hand, subscribers use MIX to create expressions indicating the kind of events they are interested in and add contextual information to them as well. In this way, participating applications can intercommunicate with each other by sharing a common component (i.e., MIX) which provides all the functionality required to handle data heterogeneity.

In the context of concept-based, the conversion and matching operations are used in order to achieve interoperation between the contextually different applications. This sub-component is only used by the adapters and no participating application is supposed to deal with it.

4.1.1 Event and Subscription Representation

Events, or to be more precise event content, are represented using the MIX model. When a producer application publishes an event/notification through the concept-based layer, it sends a `SemanticObject` representing the desired event. This so-called *semantic event* is created using the MIX data model, which defines the set of concepts that are available to describe data and metadata from a given domain. This event is composed by a tree-shape object structure.

Regarding the representation of subscriptions, subscribers express their interests (or subscription patterns) using boolean expressions that include predicates on the context of the events in question (Cilia et al., 2005). These expressions include contextual information for the correct interpretation and matching of incoming events.

4.2 Notification Service Independence

An important issue of this work was to find a way to achieve loosely coupling interaction between the concept-based layer and different underlying notification services.

The idea was to encapsulate common functionality (needed by every NS) to be used by the core of the concept-based layer, but letting specific functionality to be implemented by NS-dependent components. Thus, the main concept-based layer functionality (i.e., its core) obtains enough independence to allow easy integrability with existing NSs.

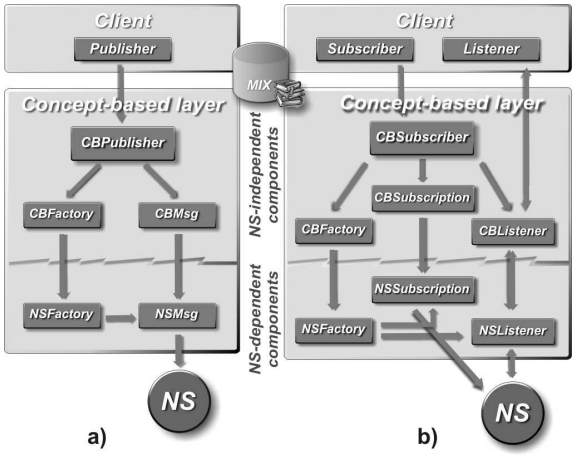

Figure 3: Publisher and subscriber components

Figure 3 (a) illustrates the collaboration among components needed to achieve the publication of messages. As can be seen, the concept-based layer is (logically) split into two "parts": the NS-independent functionality remains unaware of specific notification service details, while the NS-dependent part explicitly interacts with the notification service in question. As a consequence cooperating applications (or client code) is freed of specific notification service details. It only needs to communicate with the NS-independent part. This allows the migration from one NS to another without modifying clients' code.

In the same way, Figure 3 (b) shows the components involved with consumers. It can be noticed here again, that the concept-based layer is also split into two parts: NS-independent components (upper layer) and NS-dependent components (lower layer). As a consequence clients only interact with the upper layer.

Both figures include the MIX repository in order to allow publishers and subscribers to interact. The shared vocabulary is used by all participating clients. Using MIX, possible differences among participants at the vocabulary level and at the information context level are solved by a vocabulary agreement module.

Achieving NS independence concentrated on identifying those components that were commonly needed in every NS. We arrived to the following three entities: a) *Message*, b) *Listener*, and c) *Subscription*.

In addition to these, a factory component was needed. It is in charge of creating these entities. That's the reason why a component that plays the role of a factory (named `CBFactory`) appears in both figures. Let's briefly explain the entities involved.

4.2.1 Message (`CBMsg`)

This component is a message representative within concept-based boundaries. It is automatically created by the concept-based layer to store and manipulate the original event sent by the publisher application. It provides a common interface to access most needed message properties, hiding the real NS message to the concept-based layer. It presents an abstract interface, helping the core functionality to ignore the concrete message representation it is currently being used.

4.2.2 Listener (`CBListener`)

This component is used by consumer applications to be notified when an event of interest arrives. It acts as an intermediary between the underlying NS-listener and client's listener. Right before a notification is delivered to the subscriber it is first automatically converted to the context specified at subscription time. This conversion function is needed to deliver ready-to-process data to consumers so that no further data conversions are needed.

4.2.3 Subscription (`CBSubscription`)

When a subscriber issues a new subscription, it receives a *subscription ID* as a response. This subscription ID will be later needed to issue the corresponding unsubscription. This component hides the way an unsubscription must be notified to the underlying NS mechanism. Subscriber applications just need to invoke its `unsubscribe` method.

4.2.4 Factory (`CBFactory`)

This component is in charge of creating previously mentioned entities. It is aware of the underlying delivery mechanism (i.e., NS). For each underlying NS there is a corresponding factory. It presents an abstract interface to the concept-based layer, which remains unaware of the concrete factory it is interacting with. It provides an interface to:

- *create* a `CBMsg` wrapping a NS-dependent message,
- *publish* a `CBMsg` into the NS this factory is able to communicate with, and
- *subscribe* to events/notifications of interest, returning a `CBSubscription` to consumer applications.

5 Implementation

This section describes implementation issues, detailing the integration with different implementations of the JMS specification, as well as the J2EE platform and Rebeca.

5.1 Notification Service Independence

As previously stated, we proposed NS-independence, which led us to use the *Abstract Factory* design pattern (Gamma et al., 1995). This pattern allows to create families of related or dependent objects without specifying their concrete classes.

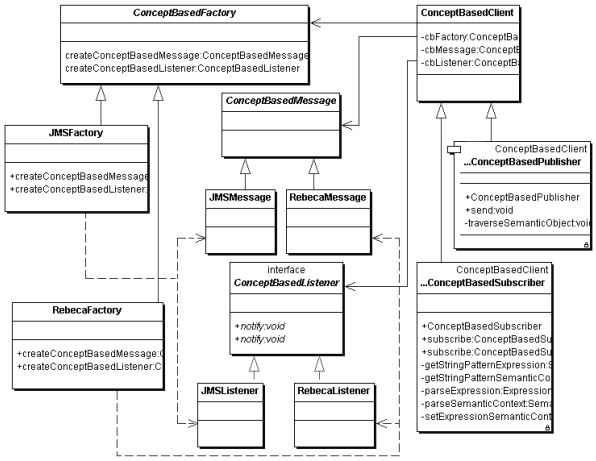

Figure 4: Concept-based's abstract factory

As can be seen in Figure 4, the `ConceptBasedFactory` provides an interface to create concept-based entities (e.g., `ConceptBasedMessage` or `ConceptBasedListener`). This factory is created at startup and is NS-dependent, so it "knows" what kind (subclasses) of entities will be needed to interact with the underlying notification service. It must be noticed that factory creation is made at system startup and cannot be changed at runtime. Thus, once the concept-based layer is up and running it remains fully attached to the underlying NS until it is restarted.

Participant applications interact with either the `ConceptBasedSubscriber` or `ConceptBasedPublisher` (consumer applications communicate with the former, while publisher applications interact with the latter). These two components (together with the abstract entities) contain the core functionality of the concept-based layer and remain unaware of the NS-dependant entities required to achieve interaction with the underlying NS.

5.2 Integrating JMS

Before a semantic event can be delivered to a JMS notification service, first it must be mapped to a JMS message as defined in the JMS specification.

When a semantic event arrives to the concept-based layer, it is converted to the default semantic context and then traversed[4] to gather every relevant information. This acquired data is inserted one by one into a `CBMsg` which in turn is mapped to an `ObjectMessage`[5]. The properties extracted from the traversed message need to be mapped to the NS-dependent message format. This data is inserted as properties of the JMS-dependent message. This mapping operations are NS-dependent and they greatly vary from the kind of JMS message type.

Now, this `ObjectMessage` is ready to be delivered to the underlying NS since it is an accurate copy (in another representation/format) of the semantic event in question. This JMS message not only contains the data/properties required by the underlying NS to perform matching and delivery operations, but also the (serialized) semantic event it stands for.

Later on, when the JMS message (i.e., `ObjectMessage`) is delivered to the concept-based layer at the subscriber-side (as a result of a matching subscription), the originally dispatched semantic event can be extracted from the JMS message, converted to the subscriber's context and finally delivered.

5.3 Integrating J2EE

This integration was accomplished using message-driven beans (MDBs) that according to the EJB specification (Sun Microsystems, 2001) are responsible for dealing with asynchronous communications. An schematic view of our approach is sketched in Fig. 5.

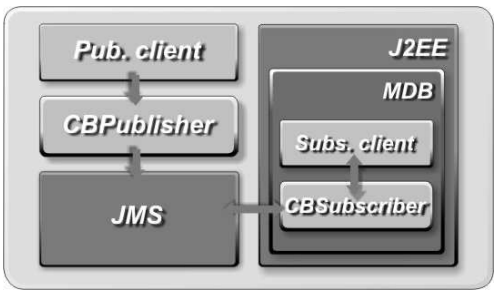

Figure 5: High-level architecture of concept-based on J2EE

[4]Remember that a semantic event is a tree-shape object structure.

[5]An `ObjectMessage` is one of the five kinds of messages supported by JMS. This particular message provides the capability to store and transport serialized objects.

J2EE subscribers are notified about events of interest through a `CBSubscriber` that acts as a MDB within the J2EE platform, whereas publishers issue events through a `CBPublisher` as usual. Interaction among participating applications is achieved by relying on a JMS delivery mechanism [6].

MDB's native functionality comprises the subscription to JMS messages and their later announcement. In this way, when concept-based publishers send notifications to a JMS NS, every MDB with a matching subscription will be notified.

Achieving this interaction between different participating applications is not a complex task. Every subscriber client at the J2EE-side has to extend the `ConceptBasedMDB` class. As usual in MDB, this class provides a hook that has to be implemented since it is invoked when corresponding notifications arrive.

It is worth mentioning that during deploy-time the MDB gets data regarding contextual information utilized by the consumer application. This information is available in its XML deployment descriptor file and every time a new message arrives to the MDB, it is used to convert the message to the desired context. Subsequently, this event can be effectively given to its consumer application.

5.4 Integrating Rebeca

Rebeca[7] is an open source event notification service framework. Its data model is an essential part of it and it is composed by two closely interrelated components: *Notifications* and *Subscriptions*. The former are messages that reify and describe `Event` occurrences emitted by publishers, while the latter are message `Filters` that express subscribers interests. The `Event` class is the base class of all notifications and encapsulates message data. The `Filter` class is the base class of all filters. Its most important functionality is to determine if a given event matches a filter.

Originally, Rebeca only supported content-based addressing model but, it was extended to support the concept-based addressing model (Antollini et al., 2004). That's why, before coding Rebeca's dependent components into the concept-based layer we firstly enhanced its own data model to support the mentioned addressing model seamlessly. We only coded two new classes:

[6]MDBs were restricted to JMS until the latest EJB specification

[7]Rebeca is a pub/sub research framework that relies on its own content-based addressing model. Nevertheless, it also provides a JMS-compliant interface that allows it to provide services resembling a JMS NS. In this section we do not take into account its JMS functionality; we just consider its pure and basic operational mode.

- `SemanticEvent`. This `Event` subclass represents a concept-based message into the Rebeca NS. It not only contains the `SemanticObject` it represents, but also the `ConceptBasedEvent` that holds additional information required by the concept-based layer.
- `SemanticFilter`. It represents an expression which is used to check if a subscription matches a given event. It is a `Filter` specialization.

With such additions the Rebeca NS is ready to support the concept-based addressing model directly. Thus, concept-based layer's modifications to support Rebeca were quite straightforward (i.e., just coding required Rebeca-dependent classes).

6 Conclusions and Future Work

This work was motivated by cooperating information-driven applications, particularly on open distributed heterogeneous environments. These applications require the dissemination of vast amounts of data, and the reaction to notifications concerning events of interest. Usually, these applications rely on pub/sub mechanisms. Interaction among such applications faces up the problem that these notifications must be understandable beyond the closed confines of a single application/module/component or enterprise. Within the context of pub/sub the source of a notification cannot anticipate who is interested in a given notification.

We proposed the extension of notification services by adding a higher-level layer that provides an abstraction to data producers and consumers that support the interaction among heterogeneous participants. This layer is responsible for mapping subscriptions and publications to the underlying notification delivery mechanism. These enhancements were designed to be incorporated on top of existing notification services. Basically, we abstract participating clients about different messaging service interfaces while providing notification service independence. We showed how to free participating applications from any data conversion responsibility and provided such a functionality as a de facto context, allowing heterogeneous applications to seamlessly interact among each other.

A variety of tests were successfully performed over different NS implementations. In the case of JMS, we discovered that the JMS specification lacks some important issues to enable full compatibility among distinct JMS vendors. On the other hand, no major difficulties arose while performing these practical evaluations on the open-source notification service called Rebeca.

In this work we do concentrate on the problem of interpretation of exchanged data among autonomous, distributed and heterogeneous information systems. We proposed a layer that resolves the problem of data conversion with the aim of reducing/avoiding misinterpretations when exchanging data. Performance measurements are needed to see the overhead induced by this layer on the different setups. Preliminary performance experiments show a "penalty" ranging from 20-40%. It must be bore in mind that bare comparisons here are not fair since producing and consuming applications do need to compute (ad-hoc) data conversions scattered among many applications. Therefore, the impact of development and maintenance needs to be taken into consideration.

As a final remark, we can add that the integral use of ontologies to support the correct interpretation of data coming from heterogeneous sources was of particular interest. The data representation of the MIX implementation used in this prototype is based on serializable java objects. An OWL-based implementation was built (Kabus, 2003) and is being integrated in the next generation of our prototype.

Additionally, a centralized administration user interface for the concept-based layer is under development. Through this interface a common vocabulary can be defined, the ontology can be edited, new conversion functions can be described, the matching context can be specified, and the mapping strategies into the underlying addressing model can be configured. Our current efforts also include different performance experiments.

REFERENCES

Abiteboul, S., Cluet, S., and Milo, T. (1997). Correspondence and Translation for Heterogeneous Data. In *Intl. Conf. on Database Theory*. Delphi, Greece.

Aguilera, M., Strom, R., Struman, D., Astley, M., and Chandra, T. (1999). Matching Events in a Content-based Subscription System. In *Procs of ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 53–61.

Antollini, J., Antollini, M., Guerrero, P., and Cilia, M. (2004). Extending Rebeca to Support Concept-based Addressing. In *Proceedings of the Argentinean Symposium on Information Systems (ASIS'04)*, Cordoba, Argentina.

Bacon, J., Moody, K., and Bates, J. (1998). Opera: Active systems. Technical Report GR/K77068, University of Cambridge - Computer Laboratory.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5):35–43.

Bornhövd, C. (1999). Semantic Metadata for the Integration of Web-based Data for Electronic Commerce. In *Procs of WECWIS'99*, pages 137–145. IEEE Press.

Bornhövd, C. (2000). *Semantic Metadata for the Integration of Heterogeneous Internet Data*. PhD thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany.

Bornhövd, C. and Buchmann, A. (1999). A Prototype for Metadata-based Integration of Internet Sources. In *Procs of CAiSE*, volume 1626 of *LNCS*, pages 439–445.

Bornhövd, C., Cilia, M., Liebig, C., and Buchmann, A. (2000). An infrastructure for meta-auctions. In *Proc. of (WECWIS'00)*.

Bornhövd, C. and Buchmann, A. P. (2000). Semantically meaningful data exchange in loosely coupled environments. In *Procs of Intl Conf on Information Systems Analysis and Synthesis (ISAS'00)*, Orlando, FL, USA.

Bray, T., Paoli, J., and Sperberg-McQueen, C. (1998). Extensible Markup Language (XML) 1.0. W3C Recommendation, W3C, `http://www.w3.org/TR/REC-xml`.

Carzaniga, A. (1998). *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy.

Cilia, M., Antollini, M., Bornhövd, C., and Buchmann, A. (2004). Dealing with heterogeneous data in pub/sub systems: The Concept-Based approach. In *Procs of DEBS'04*, Edinburgh, Scotland.

Cilia, M., Bornhövd, C., and Buchmann, A. (2001). Moving Active Functionality from Centralized to Open Distributed Heterogeneous Environments. In *Proceedings of CoopIS*, pages 195–210, Trento, Italy.

Cilia, M., Bornhövd, C., and Buchmann, A. (2005). Event handling for the universal enterprise. *Information Technology and Management – Special Issue on Universal Enterprise Integration*, 5(1):123,148.

Cugola, G., Nitto, E. D., and Fuggetta, A. (1998). Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems. In *Proc. of ICSE'98*.

D. Conolly et al. (2001). DAML+OIL (March 2001) Reference Description. W3C Note, W3C, `http://www.w3.org/TR/daml+oil-reference`.

Deutsch, A., Fernandez, M., and Suciu, D. (1999). Storing semistructured data in relations. In *Workshop on Query Processing for Semistructured Data and Non-standard Data Formats*. Jerusalem, Israel.

Eisenberg, B. and Nickull, D. (2001). ebXML Technical Architecture Specification v1.04. Technical report, `http://www.ebxml.org`.

Fallside, D. (2001). XML Schema Part 0: Primer. W3C Recommendation, W3C, `http://www.w3.org/TR/xmlschema-0/`.

Fiorano. FioranoMQ. `www.fiorano.com`.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gruber, R., Krishnamurthy, B., and Panagos, E. (1999). The Architecture of the READY Event Notification Service. In *Proceedings of th 19th IEEE Intl. Conf. on Distributed Computing Systems Middleware Workshop*.

Haase, K. (2002). *Java Message Service API Tutorial*. Sun Microsystems.

Hapner, M., Burridge, R., and Sharma, R. (1999). Java Message Service. Specification Version 1.0.2, Sun Microsystems, Inc., Java Software.

Harrison, T. H., Levine, D. L., and Schmidt, D. C. (1997). The Design and Performance of Real-time CORBA Event Service. In *Proc. of OOPSLA'97*, pages 184–200.

IBM. MQ-Series. `www-4.ibm.com/sotware/ts/mqseries`.

Kabus, P. (2003). Implementing Semantic Data Integration for the Internet (in german). Master's thesis, Department of Computer Science, Darmstadt University of Technology, Germany.

Microsoft Corp. (2000). Biztalk Framework 2.0: Document and Message Specification. Microsoft Technical Specification.

Mühl, G. (2001). Generic Constraints for Content-based Publish/Subscribe Systems. In *Proc. of CoopIS'01*, volume 2172 of *LNCS*, pages 211–225. Springer.

Object Management Group (1997). Event Service Specification. Technical Report formal/97-12-11, Object Management Group (OMG), Famingham, MA.

Object Management Group (2000). CORBA Event Service Specification, version 1.0. OMG Document formal/2000-06-15.

Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. (1993). The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th Symposium on Operating Systems Principles*.

Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. (1995). Object Exchange Across Heterogeneous Information Sources. In *Prof. of ICDE '95*.

RosettaNet (2002). RosettaNet Implementation Framework: Core specification v2.00.01. RosettaNet Technical Specification, `http://www.rosettanet.org/rnif`.

SonicSoftware. SonicMQ. `www.sonicsoftware.com/products`.

SpiritSoft. Spirit Lite. `www.spirit-soft.com/products/lite/overview.html`.

Sun Microsystems, I. (2001). Java 2 Enterprise Edition Platform Specification. Technical Report Version 1.3, Sun Microsystems, Inc.

Sun Microsystems, Inc. (1998). *Distributed Event Specification*. Mountain View, CA, USA. `http://www.javasoft.com/products/javaspaces/specs`.

Talarian. SmartSockets for JMS. `www.talarian.com/products/jms/index.shtml`.

TIBCO Inc. (1996). TIB/Rendezvous. White Paper, `http://www.rv.tibco.com/rvwhitepaper.html`.