ACTrESS - Automatic Context Transformation in Event-Based Software Systems

Tobias Freudenreich, Stefan Appel, Sebastian Frischbier, Alejandro P. Buchmann
Databases and Distributed Systems, TU Darmstadt
lastname@dvs.tu-darmstadt.de

ABSTRACT

Event-based systems (EBS) enable companies to respond to changes in their environment in a timely manner. To interpret event notifications, knowledge about their context is essential. The matching mechanisms of publish/subscribe systems depend on a common interpretation of event notifications and subscriptions that may span organisational boundaries. To mediate between such semantic contexts, we developed ACTrESS, a distributed middleware addon for automatic context transformation in event-based software systems and message-oriented middleware (MOM) in general. Transformations are substitutable at runtime and transparent to the user. ACTrESS is built on top of a production strength open source MOM extending the Java Message Service API. In this paper we present the challenges arising from differing contexts in event-based systems. We introduce ACTrESS and evaluate our solution using workloads derived from findings from research projects dealing with real-world applications of EBS.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.12 [**Software Engineering**]: Interoperability—*data mapping, distributed objects*

General Terms

Management, Performance

Keywords

event-based systems, middleware, context mediation, event semantics, contextualization, transformation

1. INTRODUCTION

Event-based systems (EBS) enable enterprises to react to meaningful events in their environment in a timely manner. They decouple participants and allow for handling software systems dynamically joining and leaving business relationships over time [10, 23, 43]. Producers and consumers of events are usually independently developed software components. This is a key advantage of EBS, as components are loosely coupled. In fact, producers and consumers are decoupled in time, space and synchronization [15]. They are connected by a notification infrastructure, often called a message-oriented middleware (MOM) in which event notifications are passed from producers to consumers in the form of messages.

In today's economy, supply chains comprise of different companies around the globe. They require many different participants and their software systems to cooperate in dynamically changing relationships. In such a setting, a company has to continuously accumulate information from different participants to decide on stock-levels, lot-sizes and opportunities in sales. This includes capturing and processing information about meaningful events generated by production processes and flows of goods.

Today's supply chains lead to a heterogeneous software landscape in which components are developed independently. The problem of data and application integration has been researched intensely for business applications [26] and even commercial solutions like Informatica® exist. EBS are typically heterogeneous systems, in which clients of different vendors can be used for publishing and subscribing to events. However, EBS have unique properties that require a new approach to making applications understand each other's data: (1) communication partners are usually anonymous, thus a consumer cannot identify the producer of an event notification; (2) (newly developed) producers and consumers join and leave the EBS during runtime and on the fly; (3) communication happens quick and nearly in real-time, thus high latencies must be avoided; and (4) often, there is no governing instance, as communication happens between mutual partners. Thus, agreeing on a globally accepted data schema, structure and interpretation is not a feasible solution in real-world settings [25, 32, 3].

A component's context, which is a set of external parameters (e.g., programming language or unit system in country of use), defines how data is interpreted. Therefore, we advocate context-transformation in publish/subscribe MOM, to avoid redundancies and unburden resource-constrained producers and consumers (e.g., nodes in wireless sensor networks or mixed-mode systems). Our approach caters to the specifics of EBS. We assume that a single producer or consumer is a homogeneous component and thus allow each producer and consumer to specify a data interpretation context. Our contributions are:

- we identify challenges for context transformation mechanisms in general;
- we model data interpretation contexts as modular, reusable units tailored to the characteristics of EBS as outlined above;
- we suggest a distributed architecture that allows for modifications at runtime, supporting the dynamic nature of EBS;
- we provide a prototypic implementation of our design (ACTrESS), by modifying an existing JMS middleware, showing that the principles proposed can be integrated into industrial-strength middleware; and
- we evaluate our prototype, showing that our solution does not impose measurable overhead and is advantageous to approaches employing techniques like self-describing messages.

The remainder of this paper is structured as follows: Section 2 gives some background information on event-based systems and handling of contexts followed by a discussion of related work on semantics and contexts. In Section 3 we present the design of our approach to describe and implement distributed context transformation in detail. We discuss our implementation in Section 4. The evaluation of our approach is presented and discussed in Section 5. Section 6 concludes this paper with a summary of our findings and a brief outlook on future work.

2. BACKGROUND

In traditional, pull-based database systems, data is rather static and conforms to known schemata. In contrast, event-based systems have to deal with highly dynamic streams of information. Data is not known a priori and neither are communication partners. This creates new challenges when dealing with differing contexts. We developed ACTrESS to meet these challenges and enable automatic context transformation in notification-based systems.

2.1 Running example

We will use a running example for illustration purposes: today's supply chains are complex and involve many companies around the globe. The need for a continuous flow of information between the participants has increased significantly with the adoption of time-sensitive production strategies like *just-in-time* production. Manufacturers need to know the delivery status of components delivered by subcontractors (including geographical position and estimated time of delivery) to adapt as early as possible to any disturbances in stock supply. However, providers can use a multitude of data formats to provide positioning information. For example, latitude/longitude coordinates as found in GPS tracking systems are as valid as addresses. Furthermore, almost every country has its own mail address format.

A seamless information flow is also needed for dynamic price calculation. Logistics providers can make better offers if they can achieve higher capacity utilization. To calculate which container and vehicle to use, they need measurement and weight information from their customers. The customers in turn can use information about available space to get information about transportation capacity, allowing for dynamic planning. However, different software systems are likely to use different units. In world-wide settings, this can be caused by different unit systems (e.g., Metric vs. Imperial) and even within one unit system, there are different choices (e.g., giving weight in grams or kilograms).

These scenarios are not a mind construct but grounded on two ongoing research projects DynamoPLV[1] and Emergent[2], dealing with the seamless integration of production, logistics, traffic management and transportation. Software systems communicate with one another in an *n-to-m* fashion, often without directly knowing their communication partners.

2.2 Context

The word *context* has different meanings, depending on the discipline and application domain in which it is used. For example, in ubiquitous computing, context often refers to the user's situation or the state of his/her environment [1]. On the other hand, natural language processing researchers characterize context as the environment of a word in a sentence or text [27]. For the purpose of this paper, we see context as a set of external parameters that lead to a certain interpretation of data. Since a single producer or consumer can be assumed to be a homogeneous component, it makes sense to assign a set of transformation rules to each of them. This set of transformation functions makes up the external parameters. More specifically, a context (i.e., the set of external parameters) is a set of data types and an assignment of units (e.g., meters for distance) to each attribute of each data type. A context may also contain mapping instructions. Thus, we treat transformation contexts as first-class citizens in a message-oriented middleware. Our definition of transformation contexts enables easy structuring, reuse and extensibility.

We want to illustrate this with our running example: a logistics provider informs its customers about spare space available in a container, hoping to fill that spare space and offering a special price as incentive. Even if spare space events are just composed of three coordinates for the available space and an attribute for the available weight, individual software components may have different names for these attributes (e.g., x/y/z/w or height/width/depth/weight). Thus, each spare space event must be mapped to the customer's internal data type(s). Even if structure is the same, meaning might still vary. For example, the measurements of the available space are in centimeters for a Europe-based logistics provider. In an American company however, measurements might always be treated in inches and if data is not transformed, miscalculations will occur. Thus, the value itself needs to be converted, which is only possible if source unit and target unit are known.

2.3 Context Transformation Challenges

Hinze et al. analyzed application domains and identified a core set of features that are typical for event-based applications [23]. We analyzed those domains with regard to event producers and consumers residing in different contexts. We identified *value semantics*, *event representation*, as well as

[1]www.dynamo-plv.de
[2]www.software-cluster.com

extensibility and scalability as the three core challenges in addition to the *specifics of event-based systems*.

2.3.1 Specifics of event-based systems

In event-based systems consumers do not know the identity of producers and vice versa. Thus, consumers are unable to interpret data correctly unless event notifications always follow a globally agreed schema, or consumers get some information about the data. A third option is to deliver the data in a way that ensures the consumer receives it in the way expected.

Matching event notifications to subscriptions also requires context mediation, as subscriptions might be issued with different data interpretation in mind (e.g., boundaries to an attribute's value). This is imperative as EBS are designed for communications with rapidly changing partners.

2.3.2 Value Semantics

Values of events can have very different meanings. The value itself often does not yield enough information that allows correct interpretation in all contexts. Even within one system, a simple number is often ambiguous. Correct interpretation of the value, however, is crucial and automatic inference of the unit is impossible due to similar magnitude of some units, e.g. yards and meters. Currency complicates the problem with different units, as exchange rates are very dynamic. Thus, the conversion function must know whether to use the time of event production, the current time or some fixed point in time. There are numerous ways to express time: 13h, 13:00, 13:00:00, 1 pm, 1:00 pm are all valid notations for one hour after noon. In addition, the specified time might refer to global time, or local time. Local time can refer to the local time zone with or without the hour for daylight saving time included. Alternatively, a relative local time could indicate that the given time is the time that passed since another event occurred. Many notifications carry location information. This might be in the form of an absolute location (e.g. GPS coordinates) or in relation to another object. Furthermore, location can refer to logical or physical coordinates.

2.3.3 Event Representation

Event-based systems have an internal event representation, called *event model*: There are several ways to represent the structure of an event notification [31]. Notifications can be structurally typed or have no predefined structure. To allow flexibility even in typed systems, Oki et al. suggest self-describing notification objects [28]. They suggest to employ adapters at the client-side to map between different structures. However, when new producers join the system, several consumers will have to be modified to understand the new structural formats. Some EBS allow defining their event notifications in a hierarchical way, similar to object-oriented programming [16], while others have a flat, non-hierarchical representation [9]. Notifications can be represented in the Extensible Markup Language (XML), some binary format or any other, suitable format. Each notification has a set of attributes with corresponding values. However, the set of attributes can differ between two applications, even for the same application domain or two different versions of the same application.

Usually, a label identifies notification attributes. However, labels for a specific value can be ambiguous. For example, the identification number for an event is usually labeled 'id' but might as well carry the label 'identifier'.

2.3.4 Extensibility and Scalability

Extensibility has become a key requirement for nearly any software. A context transformation middleware must be extensible so that the clients can at least add new transformation information.

Message-oriented middleware often provides a mechanism to distribute the middleware across multiple servers. This allows scaling with an increasing amount of clients. Supporting context transformation in the middleware must account for that. Consequently, context handling should not hamper scalability.

2.4 Related Work

In our analysis of related work, we identified two categories of existing approaches: context mediation in publish/subscribe systems to improve matching algorithms, and integrating different data sources. The latter has been researched intensively in the data integration community, but we will give related work only on a few selected aspects, as our focus is on context mediation in publish/subscribe systems. Most related work exhibits one or more of the following shortcomings: (i) solutions operate on databases and use expensive computations, which are too complex for event-based systems (ii) approaches require an immutable definition of the common ground and (iii) they rely on a global definition of it.

Researchers identified the need to consider semantics when dealing with events [11, 34]. Cilia et al. suggest that the integration of new clients into an event-based system in presence of data heterogeneity requires mediators [11] and "explicit information about the semantics of events" [12]. They advocate enhancing the notification mechanism by allowing producers to pass semantic information and consumers to receive data in their semantic context. To exchange semantic information, they use a self-describing model described in [5]. While their ideas have influenced our work, our approach stores context information inside the broker network for performance reasons (see Section 5) while remaining modular and flexible. Furthermore, we show that our approach can be integrated into existing, standard software. Scherp et al. introduce an event model which supports different event interpretations. They approach semantics from another angle and argue that the same event or set of events can have different interpretations and causality, depending on the context of the observer. Their work is more conceptual and focused on developing a sound event model. Thus, they do not provide any mechanism how to transform between different interpretations.

Some related work addresses the field of matching metadata and subscriptions semantically. Ruotsalo and Eyvönen argue that different metadata schemas hinder interoperability and promote the idea to transform the individual schemas into a shared representation [32]. Their work focuses on the development of mapping rules. Skovronski and Chiu describe a framework for a publish/subscribe system, which uses semantic data to improve the expressiveness of the subscription formulation language [36]. The focus of their work lies on making subscriptions interoperable. The Semantic Toronto Publish/Subscribe System (S-ToPSS) [29] also aims at bringing semantics into publish/subscribe systems. S-ToPSS uses ontologies to match message data to subscriptions, making use of synonyms, relationship knowledge or direct mapping functions. The authors do not rely a on single, global ontology, as they believe that there will be multiple, domain-specific ontologies instead. Wang et al. build on this idea and introduce a publish/subscribe system where subscribers express their interest in events in the form of graph patterns [42]. They use the Resource Description Framework (RDF) to represent events, converting incoming events into RDF automatically. However, they deliver events to the subscribers always in RDF.

A lot of work has been done by Blair et al. in the context of the European research project CONNECT [3, 21, 2] and preceding work on reflective middleware [4] on the role of ontologies in establishing interoperability in heterogeneous and distributed software systems. Their goal is to provide automatically generated software connectors to mediate between software systems that are heterogeneous in regard to data (syntax and semantics) and application behavior. To that end, participating systems (*networked systems*) semantically enrich advertisements about the services they offer and requests for the services they want to consume. This information is provided by the use of different *Discovery Protocols*. *Discovery Enablers* collect these requirements and use machine learning algorithms encapsulated in *Learning Enablers* to find matchings between supply and demand of those heterogeneous services. Based on the output and the models stored in a *Model Repository*, *Synthesis Enablers* generate software connectors to mediate between requested and supplied services. Their work is complementary to ours for three reasons: (1) *scope*: the focus of CONNECT is to provide mediators at runtime so that systems with heterogeneous behavior and service descriptions can be hooked up; *ACTrESS* in turn is designed to work in message-based system federations that are already hooked up, focusing on heterogeneity of the data exchanged instead of heterogeneous application behavior; (2) *approach*: CONNECT makes heavy use of ontologies and machine learning to discover first and foremost the functionality of systems; *ACTrESS* in turn uses lightweight context descriptions which could be the result of such learning techniques; (3) *architecture*: CONNECT's *Discovery Enablers* have to rely on hard-wired plug-ins for each Discovery Protocol to interpret the data encapsulated in the advertisements and service-requests correctly; *ACTrESS* in turn aims exactly at transforming these different interpretations of advertisements and notifications.

Wache and Stuckenschmidt describe a model for context transformation to achieve semantic interoperability between different information sources [38, 41]. Their approach uses description logic in first order logic to describe ontologies for the specific application domain, which they call the *shared vocabulary*. They establish the shared vocabulary for all information sources, before performing transformations. Furthermore, they suggest two different kinds of context transformation: rule-based functional transformation and classificationbased transformation and give a unifying model. Wache and Stuckenschmidt focus on presenting a formal model for context transformation, which they prove to be correct and complete for their example domain. They integrated their approach into a system, which operates on a database. Guo and Sun suggest a concept-centric approach to exchange product data between companies [22] with a focus on the domain of e-commerce data exchange. They suggest assigning a context to each company, and transform from one context to another when a company requests data. Their approach encompasses assigning concepts to product data, which is then transformed between contexts. Using contexts and concepts, they suggest creating XML product maps, which may then be queried by other companies. Similar to our approach, they advocate a common concept in relation to which other concepts are defined. However, they require all participants to agree on a global concept. Obtaining information from various heterogeneous databases is a similar problem. Researchers argue that a mediator should rewrite database queries and query results to abstract from different representations and units [7]. Gannon et al. extended this idea by developing a language that allows integrating new information sources by specifying transformation rules [20]. Both approaches cover only pull-based, static database access, with a known recipient of the requested data.

Our approach focuses on the dynamic and fast nature that is an inherent property of messaging systems. We built a system that is tailored to event-based systems, operating with low latencies and being modifiable at runtime, thus we do not rely on a fixed set of data or rules. New producers and consumers can easily be integrated, with little to no adaptation needed, as data transformation functionality is offloaded to the middleware. Furthermore, we do not rely on a globally accepted common ground. In fact, producers and consumers can connect to different message-oriented middleware systems (e.g., from different companies) at the same time, using a different context for each.

3. ACTRESS DESIGN

We suggest the following abstract architecture for distributed message brokers like Siena [9], Padres [18], Hermes [30], or peer-to-peer based approaches [39]: connections to producers and consumers are handled by a Connection Handler. Messages arriving at the broker must pass through the Context Handler, before being sent to the Message Handler. Likewise, outbound messages pass through the Context Handler, before being passed to the Connection Handler to be sent via the network (see Figure 1).

Upon receiving a message from a producer (P), the Context Handler uses its Transformation Engine to transform the message into the root context. The Transformation Engine looks up the respective client's context in the Client-Context-Mapping and queries the Context Repository for the context. After the transformation, the message is passed to the Message Handler and the middleware can then process the message further, e.g., calculating routing information or comparing it against existing subscriptions. Once the message reaches the fringe of the broker network and is about to be sent to the consumer (C), the Context Handler of the node at the fringe transforms the message from the root context into the consumer's context. Likewise, subscriptions from consumers are translated as well, so that each consumer may issue subscriptions in their own context (data flow of subscriptions is not shown in the figure).

Note that only the participants of the same broker network need to establish a root context, there is no need for a globally accepted agreement. Different root contexts can be used in different broker networks (in an extreme case the root context could even differ between individual brokers). Messages can even be passed between these broker networks by the same means as messages are exchanged between a broker network and its clients. Note that enforcing a globally accepted root context would reduce complexity even further, but does not reflect our typical event-based applications in which there is no central controlling unit.

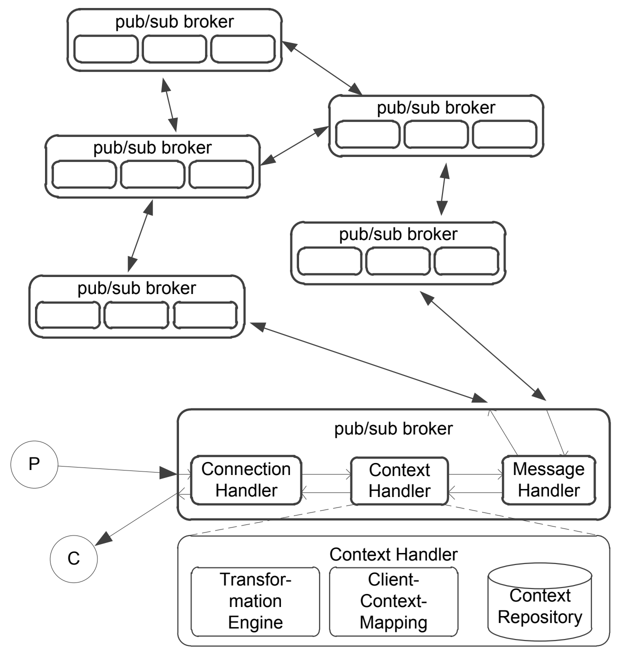


Figure 1: Abstract architecture of our solution. Each message and subscription passes through the Context Handler. The producer (P) and consumer (C) shown do not necessarily share the same context.

3.1 Context Repository

The Context Repository stores context definitions. Contexts are identified by a unique identifier. Since each node has its own Context Handler, context repositories may contain differing content. This is useful, as only the necessary contexts need to be stored. However, it might be desirable for consistency reasons or dynamic load balancing to have a shared repository across all nodes. Any suitable algorithm for data exchange in distributed storage can be used for that [35, 24]. As we have shown in previous work, we can integrate the necessary maintenance messages into the regular network traffic with little overhead [17].

If the middleware uses channel-based routing, transformations can be performed on any node along the routing path, allowing for highly flexible load balancing. However, nodes that want to transform the message must be able to obtain knowledge about the producer's or the consumer's context. If the Context Repository contains the required context (for example because the repository is replicated across all nodes), the node can simply query its own local copy. Alternatively, nodes could be allowed to query the broker network for specific contexts. Both approaches require a naming scheme, to identify contexts uniquely. We suggest a hierarchical naming scheme, similar to class naming schemes in modern object-oriented programming (OOP) languages. The identifier could consist of the node's unique identifier concatenated with the context's identifier within the node. With content-based routing, transformations must occur at the gateway nodes, because routing information is calculated based on message content and consequently, the message must be in the root context.

3.2 Context Specification

The middleware cannot know the context of a client without any prior knowledge. Thus, producers and consumers must specify which context they use when processing event notifications. To keep definitions simple and usable, we advocate specifying a context in relation to another context. Further, to greatly reduce $n \times m$ complexity, we propose to have a *root context*, which acts as a reference point for other context definitions, similar to [11]. Since we support a different root context on each broker node, complexity stays at $n \times m$ in theory. However, in practice the number of broker nodes is much less than the number of producers and consumers and not all broker nodes are interconnected. Our design does not require the root context to be defined before operation starts. In fact, message types can be added to the root context at runtime.

Producers and consumers either reuse an existing context, referencing it by its identifier, or provide their own specification. They can specify their context in relation to another context. Note that this does not necessarily have to be the root context. Thus, our design allows for context hierarchies (see Figure 2), similar to type hierarchies in OOP languages, keeping specification and maintenance efforts at a minimum. By design, producers and consumers do not need any knowledge about other participants' contexts. They do not even need to know which other clients are connected.

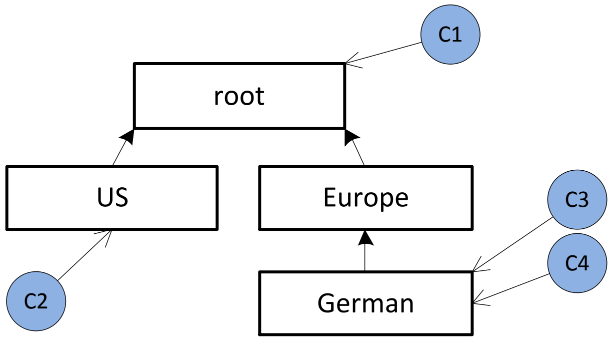
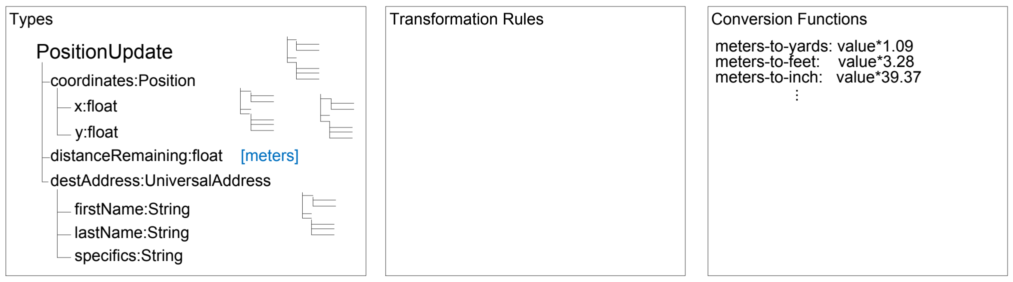
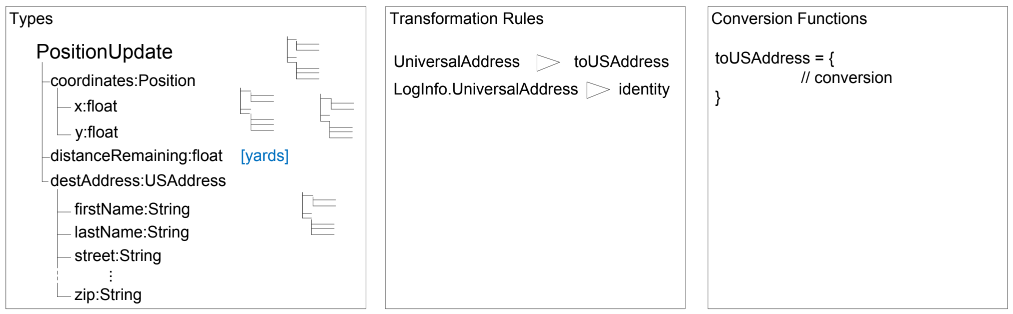


Figure 2: Context example. Client C1 has been defaulted to the root context. C2 has the US context, while C3 and C4 both have the German context, which inherits all definitions of the Europe context.

The root context consists of a set of event notification types and common conversion functions (e.g. unit conversions). Event notification types are a set of attributes, each of which has its own data type, as it is common in most programming languages. It is also important that the root context defines the unit in which it expects values. This can be achieved by additional information carried by the data type (e.g. meters for the attribute "distance") or implicitly by the data type (e.g. `UniversalAddress` vs. `USAddress`). Figure 3(a) shows an example for a root context. The `PositionUpdate` notification type has a set of attributes defining a position update of a delivery vehicle. Since the data



(a) Example Root Context



(b) US Context

Figure 3: Example context definition

type `float` does not implicitly define its unit, the unit for the remaining distance is explicitly stated. Among the `PositionUpdate` notification type are other types (indicated by the small, nested structures). The set of transformation rules is empty and the conversion functions are a set of various unit conversions.

A context defines its own event notification types and additionally, a set of rules how to transform from/to the root context's event notification types (note that a single context can only define transformations either to or from the root context). Since often only a few attributes have to be converted, we allow for a very fine grained rule definition, details can be found in [14] including conflict resolution mechanisms. Often, this is enough to transform event notifications from or to the root context. However, some transformations might require custom conversion functions. Thus, a context may also provide conversion functions as part of its definition. These functions may overload existing functions from higher up the hierarchy. New contexts only need to specify the differences to their parent context. This allows clients to reuse existing context definitions easily.

As an example, refer to Figure 3(b). It illustrates how a US context can be defined. Note that `PositionUpdate` now contains a `USAddress` attribute, rather than a `UniversalAddress` attribute. This of course means that – at least from a programming language perspective – the two types are not compatible. However, since we aim at tailoring the notifications to the client's needs, this is not a problem. As in the root context, the unit for the remaining distance is explicitly stated. In addition, the transformation rules define which conversion function to use for transforming `UniversalAddress` attributes. The first rule states that all attributes of type `UniversalAddress` should be converted to US addresses with the provided function. However, logging functionality should not be affected and thus, the identity function should be applied to `UniversalAddress` attributes appearing in `LogInfo` types (representing an item to be logged).

Figure 4 shows an example event notification transformation when converting a `PositionUpdate` from the root context to the `USContext`. For easier readability, we chose XML as the data representation, but any other representation is possible as well. The remaining distance is converted with the standard unit conversion functions, while the destination address is converted using the `toUSAddress` function (as stated in the rules).

3.3 Extending the Root Context

In Section 2 we argue that the root context must be extensible. Our design allows for easy and seamless extension of the root context. New event notification types can be added to the root context at runtime. Existing contexts will not be affected, since their transformation rules do not refer to the new types. New conversion functions can be added as well. Even if the new function is named exactly like a function of an extending context, this will not change how that extending context works, as the function is simply overloaded.

The need to extend the root contexts usually arises when new producers or consumers join the broker network and require new notification types. They have to agree on the new part of the root context, which is then simply added. Since transformations happen in the middleware, this extension happens transparently to all other producers and consumers.

3.4 Updating Existing Contexts

Clients can update contexts stored in the broker's context repository by sending the new definition of a context to the broker. However, a context can only be updated if it is not used by another producer or consumer, because they depend on the current definition. A producer uses a context c, if it

```xml
<PositionUpdate>
  <coordinates>
    <x>30.48303</x>
    <y>20.30840</y>
  </coordinates>
  <distanceRemaining>3082</distanceRemaining>
  <destAddress>
    <firstName>David</firstName>
    <lastName>Miller</lastName>
    <specifics>street=Main;number=3791;
                   ...;zip=30834</specifics>
  </destAddress>
</PositionUpdate>
```

```xml
<PositionUpdate>
  <coordinates>
    <x>30.48303</x>
    <y>20.30840</y>
  </coordinates>
  <distanceRemaining>3359.38</distanceRemaining>
  <destAddress>
    <firstName>David</firstName>
    <lastName>Miller</lastName>
    <street>Main</street>
    <number>3791</number>
     ...
    <zip>30834</zip>
  </destAddress>
</PositionUpdate>
```

Figure 4: An exemplary event notification transformation according to contexts defined in Figure 3. The event notification is transformed from the root context into the US context of a consumer.

advertises events in the context c. Likewise, a consumer uses a context c, if it defined its subscription in the context c and expects events in this context. Since we have a hierarchical context model, a context c is also in use, if a context that directly or indirectly extends c is in use. More formally, if $uses(c)$ denotes the set of producers and consumers that use context c and $extends(c)$ denotes the set of contexts that directly extend c, we define the transitive closure $extends^*(c)$ as

$$\begin{aligned} c' \in extends^*(c) \Leftrightarrow & (c' \in extends(c)) \vee \\ & (\exists c_1, \ldots, c_n : c_1 \in extends(c) \wedge \\ & c_2 \in extends(c_1) \wedge \ldots \wedge \\ & c' \in extends(c_n)) \end{aligned}$$

Then

$$depend(c) = uses(c) \cup \{uses(c') : c' \in extends^*(c)\}$$

A context c can be updated if and only if

$$depend(c) = \emptyset$$

Please note that a context might also be in use, even if the corresponding client is not connected to the broker network.

This poses the question of what should happen if $depend(c)$ is not empty. Obviously, c cannot be changed as that would break operation of other participants, but a new context c' can be created, extending c and overriding the elements that were intended to be updated. The client who intended to do the update needs to be informed about this step so it does not use the wrong name in future.

The new context definition has to be synchronized across all nodes. Furthermore, the broker network has to be queried to determine if there are other clients using the context. This causes a short synchronization overhead. However, the frequency of context updates is several orders of magnitude lower than compared to the frequency of normal event notifications.

3.5 Transformation in the Middleware

We suggest doing the transformation in the middleware rather than in the clients for several reasons:

- **Manageability** of context changes and supporting new producers or consumers.
- **Reusability** of existing context definitions.
- Support for **Resource-constrained clients** that have limited processing capabilities.
- **Easy integration** of our approach into existing software infrastructures.

It is easier to update contexts if they are stored in the middleware, rather than each client storing its own transformation instructions. The context repository allows reusing contexts and easily defining new contexts, helping both reusability and easy integration. Reusing a context in our case only requires referencing it by an identifier. Defining a new context is easy if it can be based on already existing contexts. Easy integration is especially important when software from different vendors is used. EBS are typically heterogeneous systems, in which clients of different vendors can be used for publishing and subscribing to events, e.g. by using the advanced message queuing protocol (AMQP) [40]. Thus, context transformation in clients requires adapting various code bases rather than adapting a central code base of the middleware. Furthermore, resource-constrained clients like wireless sensor nodes do not have enough processing capabilities to perform a transformation.

3.6 API Extension

To supply the server with the necessary information, we suggest extending the API of the messaging system. Typically, message brokers provide interfaces for producers and consumers (e.g., JMS provides the interfaces `MessageConsumer` and `MessageProducer`). We suggest adding two methods to the interfaces for producers and consumers:

`setContext(String identifier)` sets the context identified by `identifier` for the producer or consumer. Clients may call this method at any time, allowing them to switch contexts, should they feel the need to do so.

`defineContext(String identifier, String definition)` supplies a context definition to the middleware, identified by `identifier` and defined by `definition`. If the context repository is shared, calling this method might trigger a synchronization of the repository. We expect that new definitions are issued rarely, so this small overhead is acceptable.

4. ACTRESS IMPLEMENTATION

We implemented our approach for the Java programming language on top of ActiveMQ [37]. ActiveMQ is an open-source, fast and reliable JMS [13] broker developed by the

Apache Software Foundation. In ActiveMQ, event notifications are modeled as messages. The following sections describe our implementation and illustrate it with our running example. This approach covers all the challenges we outlined in Section 2.3.

ActiveMQ supports plugins which allow intercepting messages before and after processing. This enables us to read and modify the payload of messages before they are processed any further and before they are dispatched to their respective consumers. We implemented such a plugin, representing the Context Handler. Developing a plugin also shows that our approach is easily adaptable to other systems and programming languages, as we do not require in-depth modifications to ActiveMQ.

4.1 Context Transformation

We assume that event notifications contain hierarchically structured data types. Flat data types can easily be emulated as they are simply hierarchical data types with only one nesting level. We feel that it is important to support hierarchical structures as XML is a popular choice to structure event data [31].

As described in Section 3 extending another context only requires specifying the differences to the parent context. The question now is how to efficiently transform event notifications from and to the root context. To avoid repeated searching for definitions, we generate and compile a separate class for every defined context that has no dependencies on other contexts, by analyzing the defined data types and transformation rules. The generated classes contain the necessary information to access the required attributes and possibly needed conversion instructions. We also ensure that the context for a message can be found in constant time.

We employ a greedy transformation approach: messages are transformed by the first broker they reach. The message is then routed through the broker network and only transformed to the consumer's context by the last broker in the chain.

The transformation result is cached to avoid double work in case of multiple consumers with the same context. Since messages are usually rather small in size and the number of contexts is relatively small, the cache can be kept purely in memory and is large enough to store all transformation results for each message currently processed by the middleware. This avoids accidental latency increases due to disk accesses. Since each broker knows the number of connected consumers, it knows when a cached result is no longer needed and can free resources, avoiding memory contention.

In the case of JMS durable messages – messages that will be delivered to consumers, once they reconnect – this might lead to clearing the cache too early. However, as the next section will show, transformation impact is minimal. Since durable messages have to be loaded from a persistent storage, transformation overhead is governed by disk access.

Since the transformation happens in the middleware, our approach is transparent for any producer or consumer. However, the bootstrapping (i.e. informing the middleware about the context) is done at some point. We assume that this will be done by the event-based administrator so that application developers can write producers and consumers without worrying about context.

4.2 Administrative messages

Since we implemented our approach on a distributed broker network, we require some mechanism to pass administrative messages between the individual nodes (for example to synchronize the context repository or to provide producers and consumers with a facility to tell the middleware about their context). For our prototype, we chose to use designated JMS topics through which producers and consumers can inform the middleware about their contexts by sending appropriate messages. This approach also prevents the need for heavy modifications to ActiveMQ and the JaveEE libraries.

Similarly, brokers need to exchange some administrative messages. Again, we use JMS messages on a designated topic to provide this facility. In ActiveMQ, neighboring brokers in the broker network can be treated like consumers. These special messages are intercepted by our plugin and destroyed after processing, avoiding unnecessary processing by the rest of ActiveMQ's message handling stack.

5. EVALUATION

To show the capabilities of our implementation, we compare it against a baseline of not doing any transformations and an approach which uses Java reflection to identify the attributes of incoming messages and transform them accordingly. This is how a library implementation (or a manual implementation without any additional contextual knowledge) would work, as it has to process messages of unknown types. Especially the extreme case of self-describing messages will need to employ this or a similar mechanism of inspection. On the upside, this allows new types to be straightforwardly added at runtime. However, our experience shows that new or changed message types are rare compared to the number of messages sent.

We used our findings from our research projects (see Section 2) to design a realistic workload in terms of event size and composition as well as producer-to-consumer ratio. Furthermore, we used our experience in benchmark design [33] to ensure following sound benchmarking techniques.

5.1 Evaluation Setup

We ran our experiments in a distributed environment, using servers with Intel Xeon Quad-Core processors with 2.33GHz and 16GB RAM. We distributed producers and consumers across servers with Xeon Dual-Core processors with 2.4GHz and 16GB RAM, allowing multiple clients on the same machine. We ensured that driver machines (the server running the producers and consumers) did not become the limiting factor in any way. Since we are not interested in the performance of the broker network itself, but rather in the added overhead we did measurements just on one broker. This is still feasible, as message traffic can be assumed to be uniform across all brokers.

In addition, we also ran a few experiments in a local setting: producers, consumers and a broker instance ran on the same machine. We use this setting to exclude network contention and network latency and to gain a better understanding about limiting factors and influential sizes.

5.2 Scenarios

We compared four scenarios: *base*, *none*, *actress* and *reflect*. Scenarios *base* and *none* form the baseline and brokers do not perform any transformation work. The difference between *base* and *none* is that in *none* we simulate content-based publish/subscribe by accessing message content (requiring unmarshalling of the message). Scenario *actress* uses our approach to transform incoming event notifications to the root context and further to the respective consumer's context. Producers and consumers define their contexts as explained in Section 3. Scenario *reflect* uses –as outlined above– reflection for transformation identification and applications. To obtain fair results, we applied the same optimizations (e.g., caching) to the reflection-based approach as we used in our approach. Naturally, the transformations themselves were the same.

The transformations comprised typical transformation operations like replacing a type and converting between units (see Section 3). Transformation design too was guided by our experience from the research projects.

We did not evaluate scenarios in which one context extends another context, because extension happens on a purely conceptual level and as explained, results in a compiled class. Thus, it does not matter during runtime how deep in an inheritance-chain a context was defined.

We evaluated each scenario with a producer-to-consumer ratio of up to 1:10. We found that this is a realistic ratio in large scale enterprise applications. However, we also evaluated smaller ratios, as we were also interested in the performance development from a 1:1 to a 1:10 ratio.

5.3 Results

We did not observe any measurable difference for maximum throughput between scenarios *none* and *actress* (*base* achieved about 20% more throughput, which is not surprising given that it uses only channel-based routing). Any measured differences were within measurement precision. However, the maximum throughput for the *reflect* scenario was about half as much across all configurations and test runs. Due to the almost constant ratios of throughput performance, we omit detailed figures for brevity and focus on latency results.

Figure 5(a) shows the latencies between the different scenarios with varying numbers of consumers. As the figure illustrates, we did not observe any difference between *none* and *actress*. Compared to *base* the latency was 20% (one consumer) up to 60% (ten consumers) higher. The reflection-based approach however does not only add to latency, but latency increases much more with a growing number of consumers. Thus, our approach performs and scales better than a reflection-based approach.

To better dissect the results, we ran the same experiment in a local setting (see Figure 5(b)). Maximum throughput is lower than in a distributed environment, because one machine has to do the whole work now. However, the relative throughput performance between the scenarios is the same as above. It becomes even more evident that the reflection-based approach adds severely to latency (it is $\approx$10 times slower), while Actress does not add any measurable overhead. Differences to the *base* scenario were about the same but are less visible due to the scale of the y-axis.

Besides illustrating that our approach is faster than a reflection-based library implementation, this analysis also shows that our approach is as effective as a tedious manual coding of transformations, since we do not add any measurable overhead to the baseline performance with no transformations.

Since event notifications are usually of different complexity, we evaluated the influence of the number of attributes in an event notification. We added ten attributes to the comparably small number of five attributes which notifications already contained. This results in a total of 15 attributes, modeling very large event notifications. We added these attributes both in a flat manner and by increasing nesting levels. Figure 5(c) and 5(d) show the impact on latency under the respective modifications (throughput is only marginally affected again). Due to increased (de-)serialization effort, more attributes generally result in higher latency. Generally, we observed that flat event notifications have less negative impact on performance than deep structures (please note the different scale on the y-axis). Interestingly, scenarios *none* and *actress* perform better with flat structures compared to deep ones, while *reflect* performs better with a deep nesting level compared to flat structures (the steep increase in the graph does not continue for higher consumer counts). However, the difference for the first two scenarios is very small, while the reflection-based approach is heavily affected. In our experience, event notifications yield mostly flat structures and thus the advantage of our approach is even more apparent.

Since our approach allows for pushing transformations into the middleware and thus closer to the producers, we can avoid redundant transformations. To evaluate the benefit of this, we evaluated the effect of transforming messages close to the producer and the same transformation happening at each consumer. As the results show, being able to push the transformation close to the producer leads to huge performance gains, both in terms of maximum throughput (Figure 5(e)) and latency (Figure 5(f)).

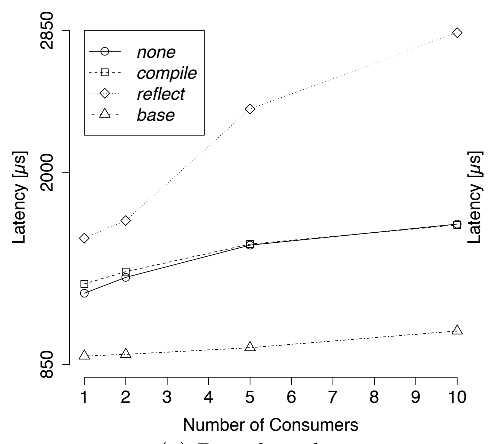
We conclude, that our approach is superior to reflection-based approaches as used by self-describing models. It negatively impacts pure channel-based performance, but does not add any measurable overhead in comparison to content-based mechanisms, as simulated by the *none* scenario. Thus, we can relieve clients from doing transformation manually, which is both tedious and can lead to negative impacts on performance.

Because the added overhead is so little, we omit comparisons with other techniques like message filters using XSLT. The clear advantage of our approach is that transformation instructions are very light-weight and easily definable, while the model underlying our approach is provably type safe.

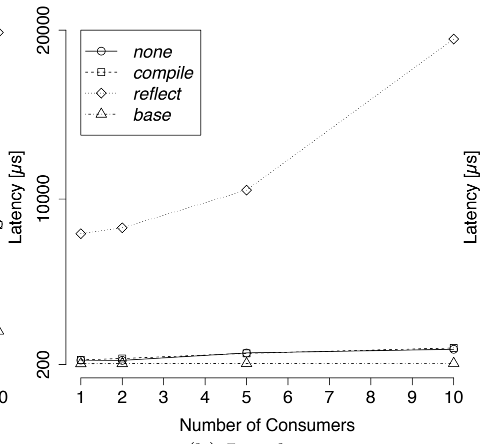
5.4 Case Study: Implementation Effort

When dealing with different data schemas or different interpretations, mediation has to happen at some point. We show that our approach reduces the implementation effort of the mediation part. We compare the necessary implementation effort and metadata necessary to describe the transformation we used in the performance analysis. We analyze our implementation and a reflection-based implementation, which does the mediation in the consumer by analyzing incoming event notifications. We did not count so called boilerplate code (e.g., class headers, import statements, etc.) which are usually automatically generated by an IDE.

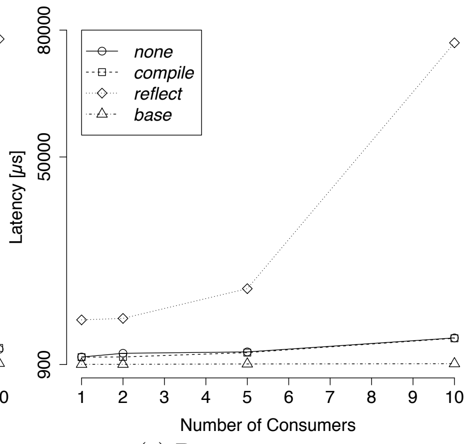
Table 1 shows the results of our analysis. Using ACTrESS



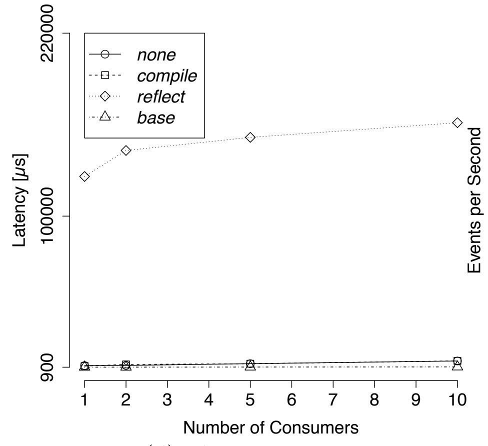
(a) Distributed



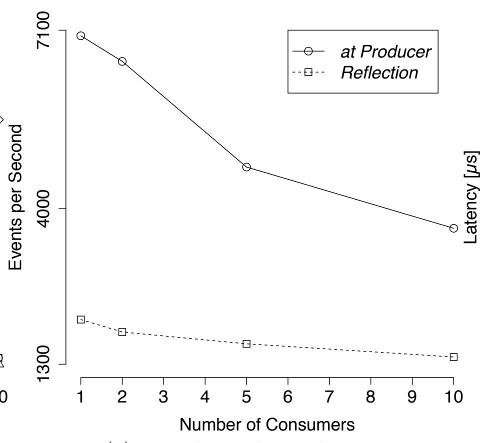
(b) Local



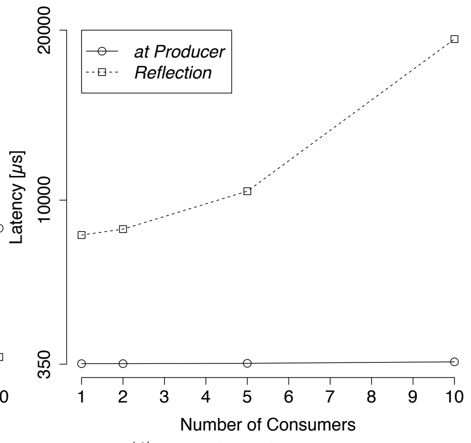
(c) Deep structure



(d) Flat structure



(e) Locality: throughput



(f) Locality: latency

Figure 5: Performance results

to offload transformations to the middleware saves about 50% of necessary code compared to a manual approach. Please note that the number of saved lines are not for the whole system, but per component that participates in the event-based system.

The advantage of our approach becomes even more apparent if we consider what happens when a new producer joins the system. If we use a manual approach in which each consumer is responsible for the transformation, we have to modify (and recompile) each consumer. This involves searching for the right point to insert code, inserting the new code and ensuring it is working correctly. One might be able determine that only a certain subset will receive the new producer's publications and thus reduce the number of components that have to be modified. However, this does not follow the spirit of event-based systems as every producer might start producing new data and the consumers who were not modified before are now unable to understand that producer's event notifications. In addition, a manual approach violates the independence of producers and consumers by forcing a new producer to notify consumers to change. With ACTrESS, the producer can seamlessly blend into the new system without consumers ever noticing the new participant (except for the published data), favoring the wanted anonymity between components.

ACTrESS

Component	*Lines of Code*
Context Rules	5 + 2 †
Conversion Functions	7
API Call	1
Total	**15**

† We have two contexts, one extending the other

Reflection

Component	*Lines of Code*
Conversion Functions	7
Reflection Analysis	14
Bootstrapping	8
Total	**29**

Table 1: Implementation effort of our case study

6. CONCLUSION

Implementing large-scale event-based systems which connect heterogeneous software components raises the challenge of allowing components to understand each other's data. A prominent domain for large-scale event-based systems is production and logistics [8]. We investigate this area in detail in current research projects (see Section 2).

In the resulting heterogeneous software systems, many event producers and consumers reside in different semantic contexts. This makes a mutual understanding of each other's data semantics indispensable for seamless interactions. Amongst others, event-driven enterprise software systems face these problems since today's IT infrastructures tend to incorporate multiple traditional systems and span company borders [19].

To enable easy event-based integration and development of systems with different semantics, we presented ACTrESS, an automatic context transformation architecture for event-based software systems, which can be added to any messaging middleware. ACTrESS maintains a runtime modifiable context repository and supports transparent context transformations for event producers and consumers. Our implementation is JMS-based with JMS being the de facto industry standard for messaging and widely used in enterprise software systems.

Compared to other approaches, our design and implementation do not rely on a globally accepted base schema and are extensible at runtime. We realized our implementation on top of an existing publish/subscribe middleware, showing that our design can be integrated into production-strength systems. Our implementation allows for easy integration of new event producers and consumers with their respective contexts.

We evaluated our system using workload based on the findings from our research projects and expertise in benchmark design. We showed that our approach does not add any measurable overhead (neither to throughput nor latency) to a content-based publish/subscribe system. Furthermore, our approach performs better than more dynamic approaches like self-describing models.

In furture work, we plan several extensions to ACTrESS. The greedy transformation approach works best in most cases, but we identified topologies, in which this might lead to unnecessary work (e.g., a producer publishing an event that two consumers will receive, one of which shares the same context as the producer). We want to develop a strategy of where to transform. Furthermore, we want to investigate if it is possible to generate context definitions automatically from ontologies (e.g., described in OWL [6]). This would help developers already familiar with OWL and allow for easy integration into other frameworks. Finally, we aim at making the management of the context repository more intuitive, e.g. by providing a graphical user interface to specify context transformation definitions.

7. ACKNOWLEDGEMENTS

This work was performed within the LOEWE Priority Program Dynamo PLV (http://www.dynamo-plv.de) supported by the LOEWE research initiative of the state of Hesse/Germany. Funding by German Federal Ministry of Education and Research (BMBF) under research grants ADiWa (01IA08006) and Software-Cluster project EMERGENT (01IC10S01). The authors assume responsibility for the content.

We also thank Patrick Eugster for his valuable feedback.

8. REFERENCES

[1] G. D. Abowd, A. K. Dey, R. Orr, and J. Brotherton. Context-awareness in wearable and ubiquitous computing. *Virtual Reality*, 3:200–211, 1998.

[2] A. Bennaceur, G. Blair, F. Chauvel, H. Gang, N. Georgantas, P. Grace, F. Howar, P. Inverardi, V. Issarny, M. Paolucci, et al. Towards an architecture for runtime interoperability. *ISoLA'10*, 2010.

[3] G. Blair, A. Bennaceur, N. Georgantas, P. Grace, V. Issarny, V. Nundloll, M. Paolucci, et al. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. *Middleware'11*, 2011.

[4] G. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the Lancaster experience. In *RM'04*, 2004.

[5] C. Bornhövd. Semantic metadata for the integration of web-based data for electronic commerce. In *WECWIS'99*, 1999.

[6] K. Breitman, A. Filho, E. Haeusler, and A. von Staa. Using ontologies to formalize services specifications in multi-agent systems. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 92–110. Springer Berlin / Heidelberg, 2005.

[7] S. Bressan, C. H. Goh, K. Fynn, M. Jakobisiak, K. Hussein, H. Kon, T. Lee, S. Madnick, T. Pena, J. Qu, A. Shum, and M. Siegel. The context interchange mediator prototype. In *SIGMOD'97*, 1997.

[8] A. Buchmann, H.-C. Pfohl, S. Appel, T. Freudenreich, S. Frischbier, I. Petrov, and C. Zuber. Event-Driven services: Integrating production, logistics and transportation. In *SOC-LOG'10*, 2010.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Journal Transactions on Computer Systems (TOCS)*, 19:332–383, 2001.

[10] K. Chandy. Event-driven applications: Costs, benefits and design approaches. In *Gartner Application Integration and Web Services Summit*, San Diego, USA, 2006.

[11] M. Cilia, M. Antollini, C. Bornhövd, and A. Buchmann. Dealing with heterogeneous data in pub/sub systems: The concept-based approach. In *DEBS'04*, 2004.

[12] M. Cilia, C. Bornhövd, and A. Buchmann. Cream: An infrastructure for distributed, heterogeneous event-based applications. In R. Meersman, Z. Tari, and D. Schmidt, editors, *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 482–502. Springer Berlin / Heidelberg, 2003.

[13] N. Deakin. Java Message Service (JMS) API. `http://www.jcp.org/en/jsr/detail?id=914`, 2002.

[14] P. Eugster, T. Freudenreich, S. Frischbier, S. Appel, and A. Buchmann. Sound transformations for federated objects. Technical report, TU Darmstadt, 2012.

[15] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[16] P. T. Eugster and R. Guerraoui. Content-based publish/subscribe with structural reflection. In *COOTS'01*, 2001.

[17] D. Eyers, T. Freudenreich, A. Margara, S. Frischbier, P. Pietzuch, and P. Eugster. Living in the present: on-the-fly information processing in scalable web architectures. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, page 6. ACM, 2012.

[18] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. PADRES distributed publish/subscribe system. In *ICFI'05*, 2005.

[19] S. Frischbier, M. Gesmann, D. Mayer, A. Roth, and C. Webel. Emergence as competitive advantage - engineering tomorrow's enterprise software systems. *ICEIS 2012*, 2012.

[20] T. Gannon, S. Madnick, A. Moulton, M. Siegel, M. Sabbouh, and H. Zhu. Framework for the analysis of the adaptability, extensibility, and scalability of semantic information integration and the context mediation approach. In *HICSS'09*, 2009.

[21] P. Grace, N. Georgantas, A. Bennaceur, G. Blair, F. Chauvel, V. Issarny, M. Paolucci, R. Saadi, B. Souville, and D. Sykes. The CONNECT architecture. *SFM 2011*, 2011.

[22] J. Guo and C. Sun. Context representation, transformation and comparison for ad hoc product data exchange. In *DocEng'03*, 2003.

[23] A. Hinze, K. Sachs, and A. Buchmann. Event-based applications and enabling technologies. In *DEBS'09*, 2009.

[24] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Oper. Syst. Rev.*, 44:35–40, April 2010.

[25] T. Landers and R. L. Rosenberg. Distributed systems, vol. ii: distributed data base systems. chapter An overview of MULTIBASE, pages 391–421. Artech House, Inc., Norwood, MA, USA, 1986.

[26] D. S. Linthicum. *Enterprise application integration.* Addison-Wesley Longman Ltd., Essex, UK, UK, 2000.

[27] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing.* MIT Press, Cambridge, MA, USA, 1999.

[28] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *SOSP'93*, 1993.

[29] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS: semantic toronto publish/subscribe system. In *VLDB'03*, 2003.

[30] P. R. Pietzuch and J. M. Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW'02*, 2002.

[31] S. Rozsnyai, J. Schiefer, and A. Schatten. Concepts and models for typing events for event-based systems. In *DEBS'07*, 2007.

[32] T. Ruotsalo and E. Hyvönen. An event-based approach for semantic metadata interoperability. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 409–422. Springer Berlin / Heidelberg, 2007.

[33] K. Sachs, S. Appel, S. Kounev, and A. Buchmann. Benchmarking publish/subscribe-based messaging systems. In *Database Systems for Advanced Applications: International Workshops: BenchmarX '10*, LNCS. Springer-Verlag, 2010.

[34] A. Scherp, T. Franz, C. Saathoff, and S. Staab. F–a model of events based on the foundational ontology dolce+dns ultralight. In *K-CAP'09*, 2009.

[35] F. B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4:125–148, April 1982.

[36] J. Skovronski and K. Chiu. Ontology based publish subscribe framework. In *iiWAS'06*, 2006.

[37] B. Snyder, D. Bosanac, and R. Davies. *ActiveMQ in Action.* Manning Publications Co., 2011.

[38] H. Stuckenschmidt and H. Wache. Context modeling and transformation for semantic interoperability. In *KRDB'00*, 2000.

[39] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS'03*, 2003.

[40] S. Vinoski. Advanced message queuing protocol. *IEEE Journal Internet Computing*, 10(6):87–89, 2006.

[41] H. Wache and H. Stuckenschmidt. Practical context transformation for information system interoperability. In *CONTEXT'01*, 2001.

[42] J. Wang, B. Jin, and J. Li. An ontology-based publish/subscribe system. In *Middleware'04*. 2004.

[43] R. Welke, R. Hirschheim, and A. Schwarz. Service oriented architecture maturity. *IEEE Computer Journal*, 44:61–67, 2011.