DREAM: Distributed Reliable Event-based Application Management

A. Buchmann, C. Bornhövd[⋆], M. Cilia[⋆⋆], L. Fiege, F. Gärtner[⋆⋆⋆], C. Liebig[†], M. Meixner, and G. Mühl[‡]
buchmann@informatik.tu-darmstadt.de

Databases and Distributed Systems Group
Department of Computer Science
Darmstadt University of Technology, Germany

Summary. New applications and the convergence of technologies, ranging from sensor networks to ubiquitous computing and from autonomic systems to event-driven supply chain management, require new middleware platforms that support proactive event notification. We present a system overview and discuss the principles of Dream, a reactive middleware platform that integrates event detection and composition mechanisms in a highly distributed environment; fault-tolerant and scalable event notification that exploits a variety of filter placement strategies; content-based notification to formulate powerful filters and concept-based notification to extend content-based filtering to heterogeneous environments; middleware-mediated transactions that integrate notifications and transactions; and scopes, which are administration primitives for both deployment- and runtime configurability, as well as for the management of policies. We discuss four prototypes that were implemented as proof of concept systems and present lessons learned from them.

1 Introduction

We are experiencing a convergence of technologies that results in an explosion of information and requires new paradigms for data and information management and processing.

- The World Wide Web is a huge source of information that was conceived for interactive search by humans. To exploit the Web in a mode other than human browsing confronts us with the need for filtering and interpreting a large amount of heterogeneous and often short-lived data.

[⋆] IBM Almaden Research Center, USA
[⋆⋆] also Faculty of Sciences, UNICEN, Argentina
[⋆⋆⋆] EPFL, Switzerland
[†] SAP AG, Germany
[‡] TU Berlin, Germany

- The deployment of smart devices requires the continuous monitoring of events as well as the appropriate context information to interpret them properly.
- The miniaturization of sensors and their ubiquitous deployment will result in massive amounts of sensor signals that must be processed, often in real time.
- Huge distributed systems must be capable of detecting and correcting failures and return autonomously to stable operation.
- New business strategies, such as *event-driven supply chain management* and *zero-latency enterprise*, depend on the timely dissemination of information and business events.

Common to the above is that signals and data, which we abstract into the notion of *events* and *event notifications*, will flow to us. When dealing with streams of events our traditional, pull-based access mechanisms to stagnant data do no longer work. Compare traditional pull-based data processing, where queries are issued against a database, with drinking water with a straw from a glass. The same straw (and the pull mechanism) is useless when trying to drink from the garden hose.

To understand the fundamental difference between traditional applications and the scenarios described above we will analyze the information space and its interactions. There are two major dimensions that characterize an interaction pattern (see Table 1): who initiates an interaction and the knowledge that exists about the counterpart. Along the first dimension we distinguish the requestor of a service or consumer of a unit of information from the provider of a service or information. Along the second dimension we distinguish between having knowledge of the identity of one's counterpart and having no knowledge.

In the well-known request-reply interaction pattern, the interaction is initiated by the client or consumer of information and the request is directed at a specific server or information provider. If the identity of the service provider is unknown we have a case of anonymous request-reply. On the other hand, if the interaction is initiated by the producer of information, we have the cases depicted in the right column of the table. If the producer knows the identity of

		Initiator of Interaction	
		Consumer	Producer
Knowledge of	Yes	*Request/Reply*	*Messaging*
Counterpart	No	*Anonymous Request/Reply*	*Event-based*

Table 1. Taxonomy of information processing space

its counterpart, we have a classical messaging interaction. However, if the producer does not know the consumer a priori, we have the typical event-based interaction that depends on a mediator or broker to connect the interested parties.

While traditional applications in which the user is in control or where the responsiveness of the system is not critical are well served by user initiated request-reply interactions, automated processes in distributed environments must react to (possibly large quantities of) events and cannot rely on consumer initiated processing. In event-based systems the producers of the events can be unaware of the consumers, thereby allowing a loose coupling between producers and consumers. This facilitates the evolution of the system, since new applications that react to events can be added without affecting the deployed infrastructure. The inherent reconfigurability of event-based systems directly addresses volatile environments, which require agile applications and infrastructures, as was noted, for example, for enterprises that want to expedite their crucial business processes [1].

The fundamental difference in the interaction pattern found in emerging applications suggests, that slowing down the flow of events to use traditional tools, e.g. through the use of caches or databases, is only a patchwork solution. It is our contention that a reliable infrastructure for management of streaming information is needed and that the importance of this infrastructure will increase as we move to a world populated by huge amounts of interconnected devices, services, and applications with different capabilities that will react and automate processes on our behalf. Streaming events must be detected, interpreted, aggregated, filtered, analyzed, reacted to and eventually disposed of.

In this chapter we present the architecture and an overview of the components of Dream, a distributed reactive middleware layer that consists of a publish/subscribe event notification service and the reactive capabilities to consume and react to events in heterogeneous environments. Specific research results and the details of the individual components of Dream have been published elsewhere and the reader is referred to these publications throughout the text. Here we will concentrate on the motivation, the principles, the design decisions and their consequences, and we will illustrate the feasibility of our approach with four implemented prototypes.

We want to emphasize that different applications may have different notions of events and will stress different portions of the middleware. Therefore, we do not propose a one size fits all platform. However, the interaction principles we discuss here are common to many application domains. We present here the issues and one possible incarnation of a publish/subscribe reactive middleware.

2 Related Work

A reactive middleware platform draws on many existing technologies. Accordingly, related work is vast, and therefore is organized according to the technologies involved. We discuss in Section 2.1 data and event dissemination; Sections 2.2 and 2.3 deal with event and data aggregation and integration; Section 2.4 is devoted to the reactive functionality needed to respond to events; Section 2.5 discusses work related to fault tolerance and the integration of notifications and transactions; finally, Section 2.6 addresses software engineering and management issues.

2.1 Event/Data Dissemination

Early work on broadcast disks addressed the issues of push-based information dissemination in asymmetric communication environments [2]. Under this approach data is simply broadcast (pushed) to all consumers. Notification services (also called event notification services) are in charge of propagating data/events to interested consumers. For instance, in CORBA an event service [3] was introduced to provide a mechanism for asynchronous interaction between CORBA objects. Here, an event channel acts as a mediator between suppliers and consumers of events. To overcome deficiencies of this service specification, the notification service [4] was proposed as a major extension with support for quality of service specifications and basic event filtering.

The Java Message Service (JMS) [5] provides the Java technology platform with the ability to process asynchronous messages. JMS was originally developed to provide a common Java interface (API) to legacy Message Oriented Middleware (MOM) products like IBM Websphere MQ (formerly known as IBM MQ-Series) or TIB/Rendezvous. In this way, the JMS API provides portability of Java code allowing the underlying messaging service to be replaced without affecting existing code. JMS provides two models for messaging among clients: point-to-point (using a queue) and publish/subscribe (by means of topics). JMS has been part of Java Enterprise Edition (J2EE) since its origin but it was incorporated as an integral part of the Enterprise Java Beans (EJB) component model in the EJB 2.0 specification. It includes a new bean type, known as message-driven bean (MDB), which acts as a message consumer providing asynchrony to EJB-based applications.

In recent years, academia and industry have concentrated on publish/subscribe mechanisms because they offer loosely coupled exchange of asynchronous notifications, facilitating extensibility and flexibility. The channel model has evolved to a more flexible subscription mechanism, known as subject-based, where a subject is attached to each notification [6]. Subject-based addressing features a set of rules that define a uniform name space for messages and their destinations. This approach is inflexible if changes to the subject organization are required, implying fixes in all participating applications.

To improve expressiveness of the subscription model the content-based approach was proposed where predicates on the content of a notification can be used for subscriptions. This approach is more flexible but requires a more complex infrastructure [7]. Many projects in this category concentrate on scalability issues in wide-area networks and on efficient algorithms and techniques for matching and routing notifications to reduce network traffic [8–10]. Most of these approaches use simple Boolean expressions as subscription patterns and assume homogeneous name spaces.

More recently a new generation of publish/subscribe systems that are built on top of an overlay network has emerged. This is the case of Scribe [11] which is restricted to topic-based addressing and is implemented on top of Pastry. The mapping of topics onto multicast groups is done by simply hashing the topic name. Hermes [12] uses a similar approach, also based on Pastry. Additionally, the system tries to get around the limitations of topic based publish/subscribe by implementing a so-called "type and attribute based" publish/subscribe model. It extends the expressiveness of subscriptions and aims to allow multiple inheritance in event types. A content-based addressing on top of a dynamic peer-to-peer network was proposed in [13] where the efficient routing of notifications takes advantage of the topology graph underneath. This work combines the high expressiveness of content-based subscriptions and the scalability and fault tolerance of a peer-to-peer system.

2.2 Event Aggregation

Events and associated data can be aggregated according to aggregation operations. In the context of active databases (aDBMSs) event aggregation involves the occurrence of two or more events. Complex situations can be specified in order to aggregate information from "low-level" events into more complex ones. These are known as composite events and are usually expressed using an event algebra, such as those defined in HiPAC [14], Ode [15], SAMOS [16], or Snoop [17]. Such algebras require an order function between events to apply event operators (e.g. sequence), or to consume events. To determine which of these events should be consumed or selected, different consumption modes were defined [18]. Usually, events are point based and timestamped to provide a time-based order with the purpose of facilitating event selection. However, in open distributed environments global time is not applicable.

In [19], Lamport presented the happened before relation, which defines a partial ordering of events based on the causality principle. An event a happened before an event b (depicted $a \rightarrow b$) if a could have influenced b; a and b are said to be causally dependent. If neither $a \rightarrow b$ nor $b \rightarrow a$, the events are said to be concurrent and causally independent. A system of logical clocks is introduced which assigns a natural number to each event (logical timestamp). Logical clocks are consistent with causality [20]: if $a \rightarrow b$, then a's timestamp is smaller than b's timestamp - the contrary is not true. In [20] the concept of vector time is presented and it is shown that vector time characterizes causality: two events are ordered by vector time iff they are causally dependent. However, neither logical clocks nor vector clocks can deal with causal relations that are established through hidden channels and also can not represent timed real world events. Thus they are not appropriate for open systems.

An approximation for modeling the clock imprecision in distributed systems has been proposed. Assuming a sparse time base (where the points at which events can be generated are discretized and predetermined), Kopetz [21] proposed the *2g-precedence* model. This model establishes that if events are at least two time granules apart, the sequence of these events can be determined unequivocally. Here an upper bound to the precision is assumed and a virtual clock granularity g is defined. Since the granularity depends on the assumed precision, it is not a feasible approach for wide area networks and open distributed systems.

Schwiderski [22] adopted the 2g-precedence model to deal with distributed event ordering and composite event detection. She proposed a distributed event detector based on a global event tree and introduced 2g-precedence-based sequence and concurrency operators. However, event consumption is non-deterministic in the case of concurrent or unrelated events. Additionally, the violation of the granularity condition (2g) may lead to the detection of spurious events.

Dyreson [23] proposed the notion of valid time indeterminacy for temporal databases, to model the fact that it is not known exactly when an event occurred. An interval timestamp together with a probability distribution is suggested, to represent the time span during which the event is supposed to have occurred. As a consequence, querying the temporal database eventually results in multi-sets that represent alternative answers to the query.

In [24] an approach for timestamping events in large-scale, loosely coupled distributed systems is proposed. This uses accuracy intervals with reliable error bounds for timestamping events that reflect the inherent inaccuracy in time measurements.

Many projects on event composition in distributed environments such as [25–27] either do not consider the possibility of partial event ordering or are based on the 2g-precedence model. Therefore, they suffer from one or more of the following drawbacks [24]: they do not scale to open systems, they provide the possibility of spurious events, or they present ambiguous event consumption.

Systems that support composite events must also address the semantic issues associated with processing composite events. For example, how timestamps are generated and the way in which events are selected and consumed.

Several recent projects are dealing with queries on streaming data [28–31]. They are basically data flow systems where tuples flow through an acyclic directed graph of processing nodes that apply stream operators. These systems are mostly centralized and they monitor and aggregate data. The aggregated data serves as the triggering event.

2.3 Event Integration

As it has been clearly identified, additional semantic metadata for the exchange of data or messages among independent applications or services is needed, not only in the context of B2B frameworks like ebXML [32], BizTalk [33], or RosettaNet [34] but also by the W3C in efforts like Semantic Web [35], DAML+OIL [36], or OWL [37].

In the first case, XML [38] and XML Schema [39] are used to define common vocabularies to describe data and business processes. Other data models similar to XML include OEM [40] and the models described in [41, 42].

In the context of W3C's Semantic Web initiative RDF [43] and RDF Schema [44] are used to provide additional semantic metadata to better enable computer and users to exchange and integrate data. RDF provides an infrastructure that supports the representation and exchange of structured metadata to describe Web resources, like (parts of) Web pages, or other RDF metadata. RDF allows the description of properties of and interrelationships among those resources in terms of ⟨resource, attribute, value⟩ triples. The attributes used can be declared in RDF Schemas which, similar to XML Schemas, give information about their intended meaning, and specify restrictions on their values. RDF Schemas and XML Schemas can play a role similar to ontologies as a common semantic basis for data and metadata representation.

In our framework we use the MIX model [45, 46] for the representation of data (i.e., event content). Like XML/XML Schema or RDF/RDF Schema MIX provides a flexible representation model for data plus additional metadata based on a common domain-specific vocabulary. However, in addition to the functionality provided by the data models discussed above, MIX directly supports data integration by making the concept of semantic context (i.e., the explicit description of implicit assumptions about the meaning of the data) and conversion functions (which allow the automatic conversion of data/events from different sources to a common context) first class citizens of the model itself. MIX should not be seen as an alternative to the models being developed in the context of the W3C but as a complementing approach that provides features that hopefully will find their way into the other XML-based models and standards.

2.4 Reactive Functionality

Reactive mechanisms were introduced in the late '80s in the form of Event-Condition-Action rules (ECA-rules) in active databases [47]. The goal of active databases was to avoid unnecessary and resource intensive polling in monitoring applications where events are detected as changes to a database and the application reacts to the occurrence of these events.

Reactive functionality is used to support a wide spectrum of applications ranging from workflow management [26, 48], personalization [49, 50], alerters [51], business rule enforcement [52, 53] up to the internal support for relational and object-oriented databases [54] particularly with the purpose of supporting integrity of constraints, view maintenance, access control, etc. Most recently active functionality is been used in the context of XML repositories [55, 56].

Active database functionality developed for a particular DBMS became part of a large monolithic piece of software (the DBMS). Monolithic software is difficult to extend and adapt. Moreover, active functionality tightly coupled to a concrete database system hinders its adaptation to today's Internet applications, such as e-commerce, where heterogeneity and distribution play a significant role but are not directly supported by traditional (active) database systems [57].

Another weakness of tightly coupled aDBMSs is that active functionality cannot be used on its own without the full data management functionality. However, active functionality is also needed in applications that require no database functionality at all, or that require only simple persistence support. As a consequence, active functionality should be offered not only as part of the DBMS, but also as a separate service that can be combined with other services to support, among others, Internet-scale applications.

Unbundling active databases consists of separating the active part from active DBMSs and breaking it up into components providing services like event detection, rule definition, rule management, and execution of ECA rules on the one hand, and persistence, transaction management and query processing services on the other [57]. Afterwards, only necessary components can be rebundled in order to provide the required functionality. A separation of active and conventional database functionality would allow the use of active capabilities depending on given application needs without the overhead of components that are not needed. Various projects like C^{2}offein [58], FRAMBOISE [59], and NODS [60] have followed this approach.

From our point of view, unbundling active functionality from a concrete system and then rebundling the corresponding components in an open distributed environment is questionable. Unbundling in this context means to give up the "closed world" assumption that traditionally underlies a DBMS. Inherent characteristics of open distributed environments impose new requirements that were not considered in centralized environments, such as the lack of global time, independent failures of nodes or communication channels, message delays, etc. The consideration of these characteristics has an enormous impact on the event detector [24], which is the essential component of an aDBMS [61].

In [62] crosseffects and potential incompatibilities arising from the combination of selective features of active, real-time and distributed object systems are discussed.

2.5 Notifications and Transactions

Transactions are a well known concept to provide reliability of execution and have been well studied in research [63–65]. In database systems, data access operations are grouped into logical units of work and executed under transaction control. Database transactions guarantee atomicity, consistency, isolation and durability (ACID) for the execution of operations in a unit of work, despite concurrency and despite the presence of application errors and system failures. Thereby, the programmer is shielded from a variety of complex situations that otherwise arise in the presence of concurrency and the possible multitude of failure modes. Finally, this clear separation of responsibilities into application-specific logic and generic middleware services renders the concept of *transaction* so powerful [65].

In aDBMS, the concept of coupling modes has been introduced [14, 66] to determine the transaction context for the triggered action with respect to the triggering user-submitted transaction. Basically, the triggered action could be executed *immediately* on behalf of the triggering transaction, *deferred* as a pre-prepare phase of the commit processing, or *detached* in a new and separate transaction. In addition, more advanced couplings are suggested which encompass a dependency between the triggering transaction and a separately spawned transaction for the rule. In particular, the execution of the triggered action may causally depend on the commit or failure of the triggering transaction. As described above, the aDBMSs have been designed on top of a central and monolithic database server. In particular, the event triggering and rule dispatching is integral part of the aDBMS and not separated into a generic middleware layer.

In distributed settings, the application process typically spans multiple transactional information systems. Grouping the information access into a single distributed transaction requires resources to be locked for the duration of the transaction and termination must be coordinated by a 2-phase-commit protocol. While this approach is realized in standardized and commonly applied middleware services [67, 68], the applicability thereof is restricted to tightly coupled systems and thus is not suitable for the integration of autonomous components.

Various extensions to the traditional transaction models have been proposed [69] which relax atomicity and/or isolation, in favor of increased concurrency, and in order to preserve the transaction and execution autonomy of involved components. In order to do so, application semantics must be taken into account. A clear separation of concerns between application logic and generic extended transaction services remains an open issue.

Queued transactions [70] support reliable, asynchronous messaging based interactions. Queued transactions are based on the concept of message recovery [65,71] and provide for the exactly once execution of a request issued from a client application to a transactional information server. Enqueuing/dequeuing of persistent messages is enclosed in a unit of work and dependent on the overall transaction outcome. Only if the unit of work is committed, the messages will be sent out and consumed. Manipulation of client application state is executed in the same transaction as enqueuing the request and later dequeuing the corresponding reply. The receiver application groups dequeuing of request message, service execution and enqueuing of reply into a single unit of work. Both, client and server are loosely coupled in terms of time-dependency and transaction autonomy, as they do not share a synchronous communication state nor a single transaction context.

In transactional publish/subscribe [72] push-style event notifications and database transactions are integrated, separating this active functionality in a middleware layer. In [73] the authors identify the restrictions of queued transactions in common message oriented middleware and suggest message delivery and message processing transactions. Middleware mediated transactions [74] (MMT) suggest a synthesis of queued transactions, transactional publish/subscribe and distributed object transactions, including database transactions. MMT are the basis for integrating notifications and transactions, as we will discuss in Section 3.5.

2.6 Software Engineering/Management

In publish/subscribe systems the control flow is not explicitly coded. This results in the desirable loose coupling and scalability but makes system management more difficult. Adding or deleting producers of notifications, for example, will affect the system's overall functionality. Therefore, notification services can be differentiated by their ability to structure sets of producers and consumers. Operational controls and management tasks can then be bound to these structures.

The channels of the CORBA notification service [4] can be interconnected and managed in event management domains [75]. The domains provide a uniform management interface, but do not offer any filtering of notifications between coupled channels. However, producers must still explicitly publish into a specific channel, moving information about application structure into the components and limiting system evolution; a problem which is only recently addressed by reflective middleware [76].

As it was described in Section 2.1, subject-based addressing schemes use a predefined tree of subjects to classify, partition and select notifications [6]. The subject tree can only be defined according to a specific viewpoint, be it notification content, network or application layout, thereby making the integration of systems difficult [77]. Many commercial systems exist (e.g. from TIBCO, IBM, etc.) that extend the basic features with bridges connecting multiple busses, transactional processing, and security. However, the management of notification services is separated from the application functionality, which is affected only implicitly, obscuring the interdependencies and complicating management of the overall system.

The Java Message Service [5] does not offer a management interface beyond selecting the persistence and time-to-live of notifications.

The SIENA event notification service is a popular example of a distributed service utilizing content-based filtering [78]. As for all other content-based filtering approaches, the filters may be used to realize visibility constraints, but these issues are not explicitly addressed. The idea of defining event zones is present in several prototypes [79,80] to limit the distribution of notifications. None of the approaches focus on visibility control as the central mechanism to coordinate applications and to localize management tasks which is a known concept in other areas and employed in Dream [81].

Peer-to-peer systems have developed interesting strategies in environments without central control. These strategies allow them to be self-organized, maintain a well-defined network topology despite of frequent node failures, offer bounded delivery depth and load sharing. As it was mentioned before, several recent efforts [13,82,83] try to combine publish/subscribe and peer-to-peer to complement the communication efficiency of the first with the manageability of the latter.

In classical software engineering, event-based interaction is also known. The observer pattern [84] directly follows this approach, and events are used in graphical user interfaces and in software integration and composition. Sullivan and Notkin introduce mediators [85] as a design approach to explicitly instantiate and express integration relationships. An implicit invocation abstraction is used to bundle components and mediators, and, with their own interfaces, to compose new components. The Field environment [86] is an early work on tool integration that relies on a centralized server to distribute events. The original approach realizes content-based filtering in a flat space of notifications, which was later extended with the Field Policy Tool by introducing a (manual) mapping of any sent message to a set of message-receiver pairs.

3 Architecture

The Dream middleware platform consists of two main portions: the event notification subsystem and the reactive functionality service. These two subsystems are complemented by the mechanisms for transaction support and scopes, an orthogonal management support. Figure 1 describes schematically the architecture.

Events are produced by sensors or applications. Event producers may be either actively pushing events or be wrapped by event detectors which pull sensor data or query the application for relevant state changes. Events can be consumed either by a reactive application, i.e. an application that integrates the reactive functionality, or by the reactive functionality service that handles events on behalf of passive applications and invokes them as needed. In addition, the reactive functionality service can invoke, via plug-ins, external systems or services. Details are given in Section 3.4.

Event producers publish event notifications. An event is the observation of a happening of interest. A notification is the reification of an event and includes the event's identity, a timestamp, and time-to-live for an event. An event notification also includes the parameters of the event, i.e. data, and in

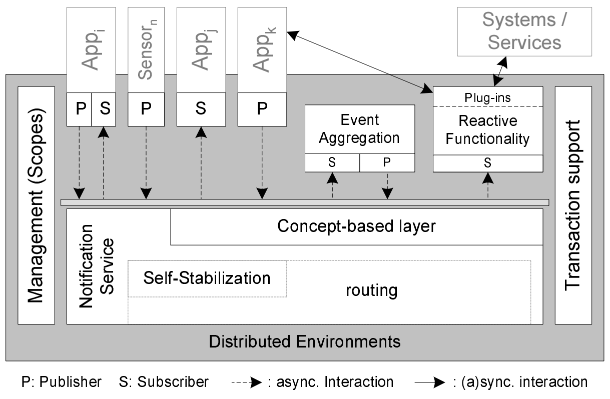

Fig. 1. DREAM Architecture

the case of heterogeneous environments, the event's interpretation context. Additionally, events carry a transaction context, if the notification service is enhanced by transaction support. Details of the event representation are given in Section 3.2.

The notification service is the delivery mechanism responsible for matching published event notifications with subscriptions. It routes events from their sources to their destinations. The routing mechanism may be realized at the physical level (IP addresses), at the level of subject hierarchies, on the basis of message content if all parties belong to one homogeneous context, or on the basis of concepts, i.e. the terms of an ontology, if publishers and subscribers belong to heterogeneous contexts. The DREAM architecture allows you to bypass a layer if it is not needed.

Publish/subscribe systems work on the basis of filters. Filters reduce the number of notifications by forwarding only those for which subscriptions exist. As such, filters typically do not combine events. In many applications, however, added value can be derived from aggregating events and reacting to a combination of primitive events. The aggregated event should then be forwarded to the subscriber. Event aggregation is discussed in Section 3.3. Although aggregation could be viewed as being part of the notification mechanism, in DREAM we have realized event aggregation outside the notification mechanism. Event aggregators are treated like any other event consuming application that can subscribe to events, aggregate them and publish a new, aggregated event. In this way we can accommodate different kinds of event aggregation, e.g. based on event graphs or on streaming queries. Details of the notification service, as well as the optimizations for filter placement and how to achieve fault-tolerance via self-stabilization are given in Section 3.1.

In publish/subscribe systems the quality of service of the delivery mechanism is not compatible with traditional notions of transactions. Transactional behavior is usually limited to guaranteed delivery of the notification from the publisher to the broker but does not include the delivery to the subscriber. In

Dream we developed a transaction mechanism, that allows the client, i.e. the subscriber, to determine the transactional quality of an interaction. Details of how notifications and transactions interact are given in Section 3.5.

Finally, the Dream architecture includes a component that supports modeling and management of event-based systems. The notion of scopes is used to structure the applications and to manage policies bound to this structure. Policies may determine the range of visibility or relevance of events, adapt the notification service functionality, and may be used for the enforcement of security policies. Scopes are described in Section 3.6.

3.1 Notification Service

The foundation of the Dream architecture is a delivery mechanism that spans the underlying distributed system. It consists of a distributed event notification service to which applications and other system services are connected as clients. These clients act as producers and consumers of notifications. The notification service itself is an overlay network in the underlying system, consisting of a subset of nodes connected in a network of event brokers. The brokers receive notifications, filter and forward them in order to deliver published notifications to all attached consumers having a matching subscription. This well-known architecture is used in a number of existing systems. However, we studied the influence of the routing strategy on the scalability and performance of the notification mechanism, specified its characteristics formally to analyze the routing and extended its dependability with self-stabilization.

Routing

We assume content-based routing to develop the routing strategies. Flooding is the simplest approach to implement routing: brokers simply forward notifications to all neighboring brokers and only those brokers to which clients are connected test on matching subscriptions. Flooding guarantees that notifications will reach their destination, but many unnecessary messages (i.e. notifications that do not have consumers) are exchanged among brokers. The main advantage of flooding is its simplicity and that subscriptions become effective instantly since every notification is processed by every broker anyway.

An alternative to flooding is filter-based routing. It depends on routing tables that are maintained in the brokers and contain link-subscription pairs specifying in which direction matching subscriptions have to be forwarded. The table entries are updated by sending new and cancelled subscriptions through the broker network. Different flavors of filter-based routing exist. Simple routing assumes that each broker has global knowledge about all active subscriptions. It minimizes the amount of notification traffic, but the routing tables may grow excessively. Moreover, every (un)subscription has to be processed by every broker resulting in a high filter forwarding overhead if subscriptions change frequently.

Our experiences have shown that in large-scale systems, more advanced content-based routing algorithms must be applied [87]. Those algorithms exploit commonalities among subscriptions in order to reduce routing table sizes message overhead. We have investigated three of them, identity-based routing, covering-based routing [78], and merging-based routing [9]. Identity-based routing avoids forwarding of subscriptions that match identical sets of notifications. Covering-based routing avoids forwarding of those subscriptions that only accept a subset of notifications matched by a formerly forwarded subscription. Note that this implies that it might be necessary to forward some of the covered subscriptions along with the unsubscription if a subscription is cancelled. Merging-based routing goes even further. In this case, each broker can merge existing routing entries to a broader subscription, i.e., the broker creates new covers. We have implemented a merging-based algorithm on top of covering-based routing [88]. In this algorithm, each broker can replace routing entries that refer to the same destination by a single one whose filter covers all filters of the merged routing entries. The merged entries are then removed from the routing table and the generated merger is added instead and forwarded like a normal subscription. Similar, the merger is removed if there is an unsubscription for a constituting part of the merged filter or if a subscription arrives that covers a part or the whole merger.

Advertisements can be used as an additional mechanism to further optimize content-based routing. They are filters that are issued (and cancelled) by producers to indicate (and revoke) their intention to publish certain kinds of notifications. If advertisements are used, it is sufficient to forward subscriptions only into those subnets of the broker network in which a producer has issued an overlapping advertisement, i.e., where matching notifications can be produced. If a new advertisement is issued, overlapping subscriptions are forwarded appropriately. Similarly, if an advertisement is revoked, it is forwarded and remote subscriptions that can no longer be serviced are dropped. Advertisements can be combined with all routing algorithms discussed above.

Self-Stabilization of Broker Networks

Being able to formally reason about the correctness of a system is an often neglected prerequisite for avoiding bugs and misconceptions. For Dream a formalization of the desired system semantics is used to evaluate the routing strategies and their implementation [88, 89]. Based on these formal semantics it was also easier to incorporate and validate a very strong fault-tolerance property into the notification service: self-stabilization.

Self-stabilization, as introduced by Dijkstra [90], states that a system being in an arbitrary state is guaranteed to eventually arrive at a legitimate state, i.e., a state starting from which it offers its service correctly. The arbitrary initial state models the effect of an arbitrary transient fault. Therefore, self-stabilization is generally regarded as a very strong fault-tolerance property. A self-stabilizing publish/subscribe system is guaranteed to re-satisfy its specification in a bounded number of steps as long as the broker topology remains connected and the programs of the surviving brokers are not corrupted.

Self-stabilization in the broker network is realized by discarding broken and outdated information about neighbors. This is accomplished by the use of leases. A node must renew its subscription to maintain it in the network. Since the process of lease renewal is idempotent, multiple renewals don't affect the functioning of the system and it is enough to renew the lease any time before it expires. A lease renewal just updates the timestamp for the lease expiration time that is stored by the brokers with each entry. If filter merging is performed, the resulting routing entries must be renewed, too. To clean up the effects of internal transient faults, brokers validate their routing tables periodically to remove entries with expired leases. Transient failures of network links are not masked by this approach and lost notifications are not automatically retransmitted. However, the approach using leases guarantees that transient faults do not upset system operation longer than necessary. In this sense, the system infrastructure can be regarded as *self-healing*.

The timing conditions for lease renewal depend on the link delay and the network diameter, i.e., the time needed to process and forward a subscription message and the number of links that must be traversed, respectively. The leasing period is the time for which a lease is granted and determines the stabilization time needed to recover from an error. There is an obvious tradeoff between these values in that more accurate behavior demands more control messages and increases the overhead. The system has been designed to adapt leasing periods to changing network characteristics. For this, the self-stabilization methodology can be applied again.

3.2 Concept-based Layer

To express subscriptions, consumers need to know about the content of the events/messages that are being exchanged. That means, that consumers must know details about the representation and assumed semantics of message content. Notification services at best specify message content by means of Interface Definition Language (IDLs) or no explicit specification is made at all. Thus, in the best case, only the data structure of the notification content is specified while leaving required information about data semantics implicit. This reflects a low level of support for event consumers that based on this scarce information must express their interest (subscriptions) without having a concrete definition of the meaning of messages assumed by their producers. Without this kind of information event producers and consumers are expected to fully comply with implicit assumptions made by participants.

The approach taken here tries to solve this problem by providing a concept-based layer on top of the delivery mechanism. This layer provides a higher level of abstraction in order to express subscription patterns and to publish events with the necessary information to support their correct interpretation outside the producers' boundaries.

Representing Events

Events are represented using the MIX model (**M***etadata based* **I***ntegration model for data* **X***-change*) [45, 46]. MIX is a self-describing data model since information about the structure and semantics of the data is given as part of the data itself. It refers to concepts from a domain-specific ontology to enable the semantically correct interpretation of event content, and supports an explicit description of the underlying interpretation context in the form of additional metadata. In other words, the underlying ontology defines the set of concepts that are available to describe data and metadata from a given domain.

In our infrastructure ontologies are organized in three categories [91]:

- The *Basic Representation Ontology* contains the basic numeric and character data types, URL, currency, date, time and physical dimensions. The representation ontology contains the necessary definitions for (un)marshalling.
- The *Infrastructure-specific Ontology* contains events (primitive, temporal, composite, etc.), notifications, and policies, such as consumption modes or lifetimes of events and notifications. These are the basic concepts of an event-based distributed infrastructure.
- The *Domain-specific Ontologies* contain the concepts for the various application domains, for example, auctions, car-specific services, or air traffic control. Some detail on these application domains is given in Section 4.

Events from heterogeneous sources can be integrated by converting them to the target context required by the respective consumer. This can be done by applying conversion functions that may include the calculation of a simple arithmetic function, a database access, or the invocation of an external service. Conversion functions can be specified in the underlying ontology if they are domain-specific and application-independent. Application-specific or service-specific conversion functions may be defined and stored in an application-specific conversion library and will overwrite those given in the ontology.

Concept-based Addressing

Since events are represented with concepts of the ontology, we can provide a high-level subscription specification. Consumers define their subscription patterns by also referring to the underlying ontology (what we call *concept-based addressing*). Subscriptions include the possibility of expressing consumers' interests using local conventions for currency, date time format, system of units, etc. In this way, consumers do not need to take care of proprietary representations and all participants use a common set of concepts to express their interests. Moreover, event consumers are allowed to express their interests without previously knowing the assumptions made by event producers. This means that one producer can send events that contain prices expressed in USD while other producers can express prices in EUR. Consumers of these events simply need to specify at subscription time the context (in this case the currency) in which they want to get the content of events. This is done by automatically applying the respective conversion functions to events before they are passed to consumers. Additionally, event consumers can benefit from the ontology and their relationships, e.g. generalization/specialization. They can use abstract concepts for specifying their subscriptions and receive instances of specializations of the abstract concept.

3.3 Event Aggregation

Events can be either primitive or composite. In most practical situations primitive events, e.g. events detected by basic sensors or produced by applications, must be combined. Usually, this composition or aggregation relies on an event algebra that may include operators for sequence, disjunction, conjunction, etc. Existing event algebras were developed for centralized systems and depend on the ability to determine the order of occurrence of events [92].

However, these event algebras and consumption policies depend on a total order of events and are based on point-based timestamps of a single central clock. These assumptions are invalidated by the inherent characteristics of distributed and heterogeneous environments. Exact knowledge of event occurrence as a point on the timeline ignores the effects of granularity and indeterminacy of time instants. As a matter of fact, granularity and indeterminacy of event occurrence time are two sides of the same coin. The timestamp granularity might reflect the inaccuracy of clocks and time measurements [93], as well as the inaccuracy of event observation - think of a polling event detector running once an hour. A coarse granular event occurrence time might be chosen on purpose, because in the universe of discourse a finer granularity is not appropriate. For example, the event could be associated with an activity that itself has a duration but this duration is not of interest. In that case, the atomicity of an event is the result of an abstraction. As another example, consider planning and scheduling of activities, which is best modelled at an appropriate time granularity related to the heartbeat of the process. In typical Supply Chain Management (SCM) scenarios for example, events are recorded at the granularity of calendar day, and only in some cases down to hours and minutes.

In all cases, the event occurrence time must be considered to be indeterminate to some extent. As a consequence, time indeterminacy must be reflected in the time model and explicitly recognized and reported when composing events in distributed and heterogenous systems.

Three factors are crucial in a generic event composition service: i) the proper interpretation of time, ii) the adoption of partial order of events, and

iii) the consideration of transmission delays between producers and consumers of events.

The first point is basically related to timestamping events. Here, timestamps are represented with accuracy intervals with reliable error bounds that reflect the indeterminacy of occurrence time, imposed by timestamp granularity and inaccuracy of time measurements [24]. In our infrastructure an abstract timestamp concept is defined and particular timestamp representations can be specialized for different scenarios and environments according to the adopted time model.

In regard to the second point, we adopted a partial order of events. Consequently, correlation methods include the possibility of throwing an exception (e.g. CannotDecide) in order to announce such an uncertainty when comparing events. In this way, the underlying infrastructure is responsible for announcing an ambiguous situation to a higher level of decision allowing the use of application semantics for the resolution [91]. In this approach, user- or pre-defined policies can be configured in order to handle these situations.

Third and final, we handle transmission delays and network failures by using a combination of a window scheme with a heartbeat protocol. The window mechanism is used to separate the history of events into *stable past* and the *unstable past and present* that are still subject to change. For composition purposes only events in the stable past are considered. With all this, it can be guaranteed in all cases that: a) situations of uncertain timestamp order are detected and the action taken is exposed and well defined, and b) events are not erroneously ordered.

The infrastructure of the event aggregation service is based on the principles of components and containers. Containers control the event aggregation process while components define the event operators logic. As mentioned before, the aggregation service is treated like any other event consumer that can subscribe to events, it aggregates them and finally publishes the aggregated event. The handling of time indeterminacy and network delays are encapsulated in such a container.

3.4 Reactive Functionality

An event-based system depends on the appropriate reaction to events. Because we are targeting open environments the reactive components must be able to interpret events originating from heterogeneous sources. Therefore, the context information presented in Section 3.2 is not only needed for event composition and notification, but also for interpretation of events and their parameters by the components reacting to them.

In Dream the traditional processing of Event-Condition-Action rules (ECA-rules) is decomposed into its elementary and autonomous parts [94]. These parts are responsible for event aggregation, condition evaluation and action execution. The processing of rules is then realized as a composition of these elementary services on a per rule basis. This composition forms a chain of services that are in charge of processing the rule in question. These elementary services interact among them based on the notification service. As mentioned before, the reactive service is treated like any other event consumer that can subscribe to events. When events of interest (i.e. those that trigger rules) are notified, the corresponding rule processing chain is automatically activated. Elementary services (i.e. action execution) that interact with external systems or services use *plug-ins* for this purpose. Besides that, plug-ins are responsible for maintaining the semantic target context of the system they interact with making possible the meaningful exchange of data.

On the foundations of an ontology-based infrastructure, a reactive functionality service was developed providing the following benefits: services interact using an appropriate vocabulary at a semantic level, events from different sources are signaled using common terms and additional contextual information, and rule definition languages can be tailored for different domains using a conceptual representation, providing end-users the most appropriate way to define rules. This conceptual representation enables the use of a “generic” reactive functionality service for different domains, making the underlying service independent from the rule specification. For instance, this service is used in the context of online meta-auctions as well as the Internet-enabled car as described in Section 4. Details related to the reactive functionality service can be found in [91].

3.5 Notifications and Transactions

In the event-based architectural style the event producer is decoupled from the event consumer through the mediator. Therefore, any transaction concept in an event-based system must include the mediator. On the other hand, applications will be implemented in some (object-oriented) programming language. Object transactions, as provided for example by CORBA in the OTS, are based on a synchronous, $1:1$ request/reply interaction model that introduces a tight coupling among components. The challenge is therefore, to combine notifications with conventional transactional object requests [95] into middleware mediated transactions (MMT) [74]. MMTs extend the atomicity sphere of transactional object requests to include mediators and/or final recipients of notifications.

To properly understand MMTs and their benefits, we briefly describe below the basic issues of component connection, interaction, and reliability.

The **topology** of interacting components may be *fixed* or *variable* and could be $1:1$, $1:n$, or $n:m$. Dependent on the knowledge of the counterpart, the **binding** of producer to consumer may be *reference-based* (at least one party has an exact reference to the other) or *mediator-based* (parties have no direct reference to each other but interact through the mediator based on subjects, topics, content or concepts).

We may consider the **life-cycle** dependencies between interacting components. The interaction model may not require the components to be available at the same time. This means, that components can execute as *time-independent*. If components must be available at the same time in order to interact, then they are *time-dependent*. Time-independence is a major aspect of loose-coupling in addition to mediator-based interactions.

The **synchronicity** of interactions describes the synchronization between components and whether they block while waiting for completion of the interaction. We distinguish *synchronous*, *asynchronous*, or *deferred synchronous* behavior.

The **delivery** guarantees for notifications may be *best-effort*, *at-most-once*, *at-least-once* or *exactly-once*. The first two are unreliable while the last two are reliable. In addition to delivery we must consider the coupling to the execution of reactions. The **processing** of the subscriber may be coupled *best effort* or *atomic transactionally coupled*. In case there is a coupling between execution of producer, subscriber and mediator, the **recovery** from situations of failures could be *forward* (outgoing) or *backward* (incoming) from the point of view of the publishing transaction.

To illustrate the differences among well known mechanisms we consider object-oriented communication and transactions and traditional publish/subscribe. For common remote method invocation styles, such as in CORBA and J2EE, the topology is fixed to 1:1 connections and bindings are reference-based. Also, components are time-dependent on each other and invocations are typically synchronous. As there is no mediator, transactions either group the initiator (client) and the responder (server) of interactions into a single atomicity sphere, or the execution of initiator and responder is carried out in independent transaction contexts. There is no middleware support for message based recovery at the client. As a consequence, forward recovery after client failure must be dealt with at the application level and guaranteeing exactly-once execution - from the client's point of view - is a tedious task.

Traditional publish/subscribe systems provide a more flexible and loose coupling, as they support variable n:m topologies with time-independent interactions via a mediator. However, transactional delivery is restricted to the mediator and subscribers are restricted to transactional consumption of the notifications. There is no means for transactionally coupling with respect to the contexts of publisher and subscriber.

In order to integrate producers, mediators and subscribers, a more flexible transactional framework is needed. This framework must provide the means to couple the visibility of event notifications to the boundaries of transaction spheres and the success or failure of (parts of) a transaction. It must also describe the transactional context in which the consumer should execute its actions. It must specify the dependencies between the triggering and the triggered transactions, dynamically spanning a tree of interdependent transactional activities.

Table 2 illustrates the possible options for coupling event producers, mediators and recipients of notifications. With *immediate visibility*, events are visible to consumers as soon as they arrive at the consumer's site and independent of the outcome of the triggering transaction. *On commit (on abort) visibility* specifies that a consumer may only be notified of the event if the transaction has committed (aborted). *Deferred visibility* requires that the consumer be notified as soon as the producer starts commit processing.

	modes
Visibility	immediate, on commit, on abort, deferred
Transaction Context	none, shared, separate
Forward Dependency	none, commit, abort
Backward Dependency	none, vital, mark-rollback
Production	transactional, independent
Consumption	on delivery, on return, atomic, explicit

Table 2. Transaction framework provided couplings

A commit (abort) *forward dependency* specifies that the triggered reaction only commits if the triggering transaction commits (aborts). *Abort forward dependencies* are a powerful concept to realize compensations and to enable recovery of long running processes.

A *backward dependency* constrains the commit of the triggering transaction. If the reaction is *vitally* coupled, the triggering transaction may only commit if the triggered transaction has been executed successfully. If the consumer is coupled in *mark-rollback* mode, the triggering transaction is independent of the triggered transaction's commit/abort but the consumer may explicitly mark the producer's transaction as rollback-only. Both backward dependencies imply that a failure to deliver the event will cause the triggering transaction to abort.

If the reaction is coupled in *shared mode*, it will execute on behalf of the triggering transaction. Of course this implies a forward and a backward dependency, which is just the semantics of spheres of atomicity. Otherwise, the reaction is executed in its own atomicity sphere, i.e. *separate top* and the commit/abort dependencies to the triggering transaction can be established as described above.

Once an event has been consumed, the notification is considered as delivered and will not be replayed in case the consumer crashes and subsequently restarts. The consumption modes allow a loose or tight coupling of event consumption to the consumer's execution context and its transaction boundary.

We have implemented the above framework in the X^2TS prototype [95]. It is discussed in Section 4 with the other prototypes.

3.6 Scopes

Despite the numerous advantages offered by the loose coupling of event-based interaction, a number of drawbacks arise from the new degrees of freedom. Event systems are characterized by a flat design space in which subscriptions are matched against all published notifications without discriminating producers. This makes event systems difficult to manage. A generic mechanism is needed to control the visibility of events, e.g. for security reasons, and for structuring system components, extending visibility beyond the transactional aspects presented in Section 3.5.

Scopes [96] allow system engineers to exert explicit control on the event-based interaction; it is a functionality orthogonal to the transaction mechanism that stretches across the different layers of the Dream architecture (see Figure 1). The following examples illustrate the kinds of explicit control offered. In order to overcome the flat design space of event systems, scopes offer interfaces that limit the distribution of published notifications. A scope bundles several producers and consumers, it limits the visibility of events in the sense that notifications are delivered to consumers within the same scope but are a priori invisible elsewhere. According to an assigned interface a scope may forward internal notifications to the outside, and vice versa. It encapsulates the composed producers and consumers and may itself be a member of other scopes, thus creating new clients of the notification service as long as the resulting graph is acyclic [89]. The achieved information hiding may be used to abstract from details of application structure.

Ontologies are used to map notifications between producers and consumers of different application contexts. However, context is typically not identified with arbitrary single producers or consumers, but instead characterizes different (parts of) applications that are to interact at runtime. Context descriptions can be bound to scopes, which bundle coherent groups of cooperating clients and share a common application context. For example, a scope could restrict visibility to trade-event notifications originating from NASDAQ. The scope could have an associated context "US-exchange" that defines the currency, the valid fractions in a quote, the date and time formats, etc.

Scopes are also used to exert a finer control of who is going to receive a given notification. A typical example for such a delivery policy is a 1-of-n delivery to only one out of a set of possible consumers. Note that, from the producer's and consumer's point of view, event-based semantics are maintained, only the interaction within a specific scope is to be refined. Delivery policies bound to a scope specify the refined semantics of delivery in this scope.

Different aspects of controlling visibility of events are addressed in a number of existing products and research contributions, ranging from administrative domains [97] to full database access control [98]. However, prior to proposing scopes we were missing an approach that is orthogonal to the other aspects of Dream. We see scopes as the means by which system administrators and application developers can configure an event-based system. Scopes offer an abstraction to identify structure and to bind organization and control of routing algorithms, heterogeneity support, and transactional behavior to the application structure. They delimit application functionality and contexts, controlling side effects and associating ontologies at well-defined points in the system. This is of particular importance as platforms of the future must be configurable not only at deployment time but also once an application is in operation.

4 Proof of Concept Systems

In this Section we present prototypical implementations that illustrate the concepts discussed above.

4.1 Rebeca

Rebeca (Rebeca Event-Based Electronic Commerce Architecture) [89,99] is a content-based notification service that implements the event notification and routing described above and does not rely on only a single routing algorithm, but implements all the algorithms presented in Section 3.1.

Scopes are implemented on top of Rebeca. Several options exist which differ in the degree of distribution, flexibility to evolve, and the necessary management overhead. A completely distributed solution divides the existing routing tables into scope-specific tables and uses the plain routing mechanisms for intra scope delivery. Cross scope forwarding and the application of interfaces and context mappings are handled when notifications are transfered between routing tables. A number of alternatives are available here: scope crossing may be centralized to enforce some security policy, to audit traffic, or to bridge specialized implementations of notification delivery. On the other hand, single scopes may be realized by a centralized notification broker, offering a finer control of notification forwarding and delivery policies.

Measurements of the characteristics of the routing algorithms were conducted with the help of a stock trading application that processed real data feeds taken from the Frankfurt stock exchange [87]. The results clearly show the beneficial effects of using advanced routing algorithms in the Rebeca notification service.

4.2 X^2TS

X^2TS integrates distributed object transactions with publish/subscribe systems [95] and is a prototype for the MMT concept in Dream. X^2TS supports the full range of object transaction features, such as indirect context management, implicit context propagation, interposition for and integration with X/Open XA resources such as RDBMS and MOM resource managers. The prototype supports a push-based interface with one-at-a-time notifications. It assumes that events are instances of types and that subscriptions may refer to specific types and to patterns of events. Subscription to patterns of events and coupling modes are imposed by the consumer by configuring the service proxy.

The X^{2}TS prototype implementation currently supports the following middleware mediated transaction (MMT) features:

- Deferred, on abort and on commit visibility.
- Checked transactions to support forward couplings and vital backward dependencies.
- Backward processing dependencies with the possibility to define groups of subscribers to be vital.
- Production policy is non-transactional but can be explicitly programmed as transactional.
- Consumption policies are atomic, on delivery, on return, explicit.
- X^{2}TS managed transactions at the consumer side. The consumer may provide its own transaction context, select a shared context or let X^{2}TS create a new transaction for each notification. Grouping of notifications in one transaction is possible through the definition of a composite event.

Reliable, exatly-once delivery and recoverable processing of events is realized in a straight forward combination of *producer at-least-once* and *consumer-at-most-once* strategy. The basic idea is as follows. The consumer site uses a persistent and transactional log, which keeps track of the identifiers of processed notifications. Arriving notifications are only delivered, if the identifier is not found in the log. Logging the identifier is atomically executed as part of the consumer's unit of work, and on successful completion thereof an acknowledgment is returned to the publisher. The publisher keeps on resending notifications until the transmission is acknowledged by each subscriber. X^{2}TS allows subscribers to the same event to specify different couplings. While one subscriber may react on commit of the triggering transaction, the other may need to react as soon as possible.

The realization of visibilities and dependencies is basically achieved by sending events about transaction status changes, and the propagation of the publisher's transaction context within the notification message. Thereby, events need only to be published once. At the consumer site, a built-in event composition enforces the visibilities and dependencies. Detection of event patterns is realized through pluggable event compositors. The application of these concepts to multi-level transaction services, which encompass activities at the process level, is considered to be straight forward.

4.3 Meta-Auctions

Auctions are a popular trading mechanism. The advent of auction sites on the Internet, such as eBay or Yahoo has popularized the auction paradigm and has made it accessible to a broad public that can trade practically anything in a consumer to consumer interaction. However, the proliferation of sites makes life more difficult for the buyer. Consider the case of a collector. With the current auction sites, she has to manually search for the item of interest, possibly visiting more than one auction site. If successful, she might end up being engaged in different auctions at multiple auction sites. There are two obvious shortcomings to this approach: first, the user must poll for new information and might miss the window of opportunity, and second, the user must handle different auction sites with different category setups and different handlings. This motivates the need for the meta-auction broker [100], which provides a unified view of different auction sites and services for category browsing, item search, auction participation and tracking (see Figure 2).

Notifications about events, such as the placement of a highest bid, and their timely delivery represent valuable information. Propagation of events leads to a useful and efficient non-polling realization of an auction tracking service. Therefore, publish/subscribe as an additional interaction paradigm is needed for disseminating process-related information efficiently. Events related to the auction process are disseminated using the concept-based notification service presented in this paper. This way, publishers and subscribers use a common semantic level of subscription.

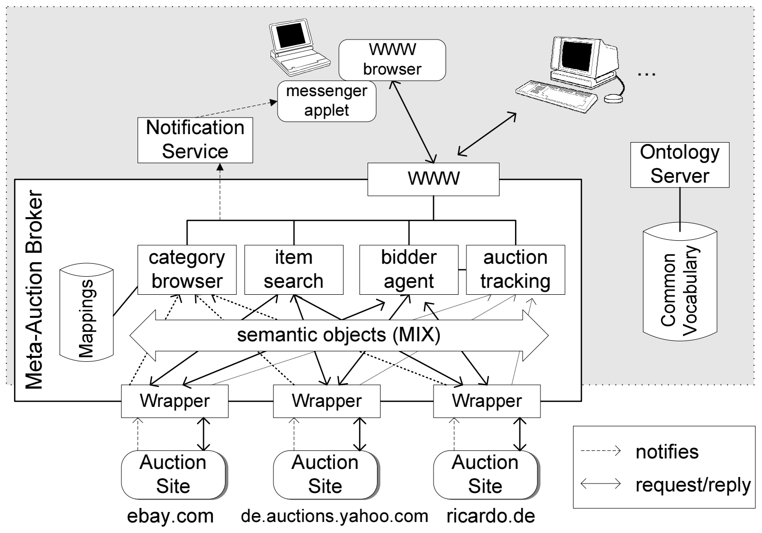

Fig. 2. Meta-Auction Architecture

Each site participating in the meta-auction system provides information about items and the auction process but does not share a global data schema nor may we assume a global schema for notifications. Today, the exact meaning of terms, entities and notifications used by different auction sites is still left implicit. To enable the brokering between different participating auction sites, the precise understanding of the terms used by each site is needed and should be made explicit through a domain-specific ontology. We introduced an ontology-based infrastructure for explicit metadata-management on top of which the meta-auction service can be realized and the mappings through which the domain-specific ontlogies can be combined.

The auction process itself can be defined using state charts. Since they are event-driven, they can be easily implemented with ECA-rules. Different sets of rules can describe different types of auction processes (ascending, reverse, dutch, etc.) [101].

To track an item of interest during an auction process, e.g. to detect that another bidder has reached a highest bid, or that the deadline of an auction is approaching, an agent can be used. Bidders can benefit from the reactive service to program their own agents. In contrast to current agent bidders that are owned, controlled and implemented by the auction house, these agents can react to happenings of the auction process according to the bidders' strategy.

4.4 Internet-enabled Car

Similar to other pervasive computing environments, cars will see a convergence of Internet, multimedia, wireless connectivity, consumer devices, and automotive electronics. Wireless links to the outside world open up a wide range of telematics applications. Automotive systems are no longer limited to information located on-board, but can benefit from a remote network and service infrastructure.

Consider the scenario, where vehicles, persons and devices have a web presence (or portal). Within this scenario new possibilities emerge, e.g. the adjustment of instruments according to personal preferences, favorite news channels, sports, music or access to one's e-mail and calendar through the portals. Through the portals this can be made independent of a particular car and could be applied to any rental car. But not only instruments can be adjusted, services can be personalized too. Services such as, "find and set the route to the next gas station", or "book an appointment to change oil" can take into account company's, and/or driver's preferences.

The portals are enhanced with the reactive functionality service in order to react to events of interest according to user preferences. Preferences are stored and managed by the portal manager.

Vehicles are equipped with a GPS receiver and a box. This box plays the role of a mediator between the vehicle itself and the external world. It can access a vehicle's electronic and diagnostic interfaces (like interfaces J1850, ODB-II) and it is responsible for announcing status changes to its portal. The portal manager can react to them by using the reactive functionality service (see Figure 3).

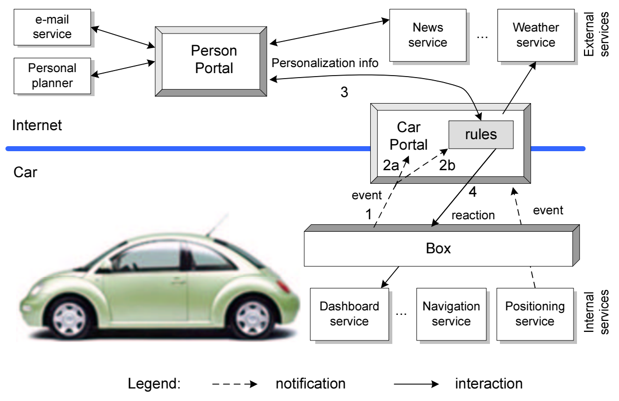

Fig. 3. An Internet-enable Car

A prototype [50] was developed in conjunction with industry to show the reaction of a personalized car to different situations based on a set of user-defined rules. Implemented services include:

Adjustment of instruments: When the driver gets into the car all her preferences (preferred language, units used for temperature, distance and velocity, format of date/time, radio stations, music preferences, etc.) are automatically loaded. The driver's identity is detected (e.g. smartkey) when getting into the car and the car's box communicates this event to its portal. As a reaction, a rule is fired which reads the driver's preferences and contacts the box to set/load them into the car's instruments.

Low fuel: A sensor signals this event and a location service is invoked to find the next gas station considering current geographical position, destination and the preferred vendor.

Driving to work: A commuter gets into the car on a workday and the current time is between 8:00am–9:00am. As a reaction the best route to work avoiding traffic jams is offered and passed to the navigation service; today's scheduled meetings are reviewed; company news and other personalized news are obtained; and e-mails can be read. Because drivers should concentrate on driving, all this information is read out by using a text-to-speech service.

5 Conclusions and Future Work

We have shown the main properties that we expect a reactive event-based middleware platform to exhibit:

- event detection and composition/aggregation mechanisms in a highly distributed environment;
- robust and scalable event notification that exploits a variety of filter placement and self-stabilization strategies;
- content-based notification to formulate powerful filters;

- concept-based notification to extend content-based filtering to heterogeneous environments;
- middleware-mediated transactions that integrate notifications and transactions with well-defined semantics for visibility, life-cycle dependencies, synchronicity, delivery and recovery dependencies;
- configuration and administration primitives, such as scopes, for both deployment- and runtime configurability, as well as attachment and administration of policies.

The complete set of requirements for the wide spectrum of potential applications cannot be anticipated. In addition, experience has shown that generic platforms can only cover a certain percentage of the business semantics. Therefore, we believe that two conditions must be clearly fulfilled by the Dream platform: 1) no false claims should be made, i.e. if the semantics cannot be clearly defined, the middleware should not give a false sense of security to the user and rather let the application developer compensate based on application semantics, and 2) configurability of event-based platforms must be guaranteed.

We have developed both the necessary concepts and prototype solutions for individual parts of such a system. The results have been published individually. In this paper we presented a system overview that describes how the parts interact. We are in the process of reengineering the platform to eliminate the gaps and inconsistencies that are typical when integrating several theses.

References

1. Roy Schulte. A Real-Time Enterprise is Event-Driven. Research Note - T-18-2037, Gartner Group, 2002.
2. S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast Disks: Data Management for Asymetric Communications Environments. In *Proceedings of the ACM Intl Conference on Management of Data (SIGMOD)*, 1995.
3. Object Management Group. Event Service Specification. Technical Report formal/97-12-11, Object Management Group (OMG), Famingham, MA, May 1997.
4. Object Management Group. CORBA Notification Service Specification. Technical Report telecom/98-06-15, Object Management Group (OMG), Famingham, MA, May 1998.
5. M. Hapner, R. Burridge, and R. Sharma. Java Message Service. Specification Version 1.0.2, Sun Microsystems, JavaSoftware, November 1999.
6. B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus – An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th Symposium on Operating Systems Principles (SIGOPS)*, pages 58–68, USA, December 1993.
7. A. Carzaniga, D. R. Rosenblum, and A. L. Wolf. Challenges for Distributed Event Services: Scalability vs. Expressiveness. In *Engineering Distributed Objects (EDO'99)*, Los Angeles, CA, May 1999.

8. Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. Exploiting IP Multicast in Content-based Publish-Subscribe Systems. In J. Sventek and G. Coulson, editors, *Proceedings of the IFIP/ACM Intl Conference on Distributed Systems Platforms and Open Distribued Processing(Middleware)*, volume 1795 of *LNCS*, pages 185–207. Springer, 2000.
9. G. Mühl, L. Fiege, and A.P. Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *Proc. of ARCS*, volume 2299 of *LNCS*, 2002.
10. F. Fabret, F. Llirbat, J. Pereira, A. Jacobsen, K. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *Proceedings of the ACM Intl Conference on Management of Data (SIGMOD)*, pages 115–126, 2001.
11. Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In J. Crowcroft and M. Hofmann, editors, *Third International on Networked Group Communication (NGC 2001)*, volume 2233 of *LNCS*, pages 30–43, London, UK, 2001. Springer-Verlag.
12. Peter Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In J. Bacon, L. Fiege, R. Guerraoui, A. Jacobsen, and G. Mühl, editors, *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, July 2002. IEEE Press. Published as part of the ICDCS '02 Workshop Proceedings.
13. Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In H.-Arno Jacobsen, editor, *In Online Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003.
14. U. Dayal, B. Blaustein, A.P. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D.R. McCarthy, A Rosenthal, S.K. Sarin, M.J. Carey, M. Livny, and R. Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. *ACM SIGMOD Record*, 17(1), March 1988.
15. Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model & implementation. In Li-Yan Yuan, editor, *Proceedings of the Intl Conference on Very Lage Data Bases (VLDB)*, pages 327–338, Vancouver, Canada, August 1992. Morgan Kaufmann.
16. S. Gatziu and K. R. Dittrich. Events in an Active Object-Oriented Database System. In *Proc. of RIDS'93*, 1993.
17. S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language for Active Databases. *Data and Knowledge Engineering*, 14(1):1–26, November 1994.
18. S. Charkravarthy, V. Krishnaprasad, E. Anwar, and S. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the Intl Conference on Very Lage Data Bases (VLDB)*, pages 606–617, September 1994.
19. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.
20. R. Schwarz and F. Mattern. Detecting Casual Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3), 1994.
21. H. Kopetz. Sparse Time versus Dense Time in Distributed Real-Time Systems. In *Proc. of ICDCS*, 1992.

22. S. Schwiderski. *Monitoring the Behaviour of Distributed Systems*. PhD thesis, Selwyn College, Computer Lab, University of Cambridge, 1996.
23. Curtis E. Dyreson and Richard T. Snodgrass. Valid-time Indeterminacy. In *Proceedings of the IEEE Intl Conference on Data Engineering (ICDE)*, 1993.
24. C. Liebig, M. Cilia, and A.P. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings of the Intl Conference on Co-operative Information Systems (CoopIS)*, 1999.
25. C. Ma and J. Bacon. COBEA: A CORBA-based Event Architecture. In *Proc. of COOTS'98*, 1998.
26. A. Geppert and D. Tombros. Event-based Distributed Workflow Execution with EVE. In *Proceedings of the IFIP/ACM Intl Conference on Distributed Systems Platforms and Open Distribued Processing(Middleware)*, The Lake District, September 1998.
27. S. Yang and S. Chakravarthy. Formal Semantics of Composite Events for Distributed Environments. In *Proceedings of the IEEE Intl Conference on Data Engineering (ICDE)*, 1999.
28. Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of the Intl Conference on Very Lage Data Bases (VLDB)*, 2002.
29. Hari Balakrishnan, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Michael Stonebraker, Ying Xing, and Stanley Zdonik. Aurora: A Distributed Stream Processing System. In *Proceedings of the Intl Conference on Innovative Data Systems Research (CIDR)*, 2003.
30. Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proceedings of the Intl Conference on Innovative Data Systems Research (CIDR)*, 2003.
31. Sirish Chandrasekaran, Amol Deshpande, Mike Franklin, Joseph Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the Intl Conference on Innovative Data Systems Research (CIDR)*, 2003.
32. B. Eisenberg and D. Nickull. ebXML Technical Architecture Specification v1.04. Technical report, February 2001. `http://www.ebxml.org`.
33. Microsoft Corp. BizTalk Framework 2.0: Document and Message Specification. Microsoft Technical Specification, December 2000.
34. RosettaNet. RosettaNet Implementation Framework: Core Specification v2.00.01. RosettaNet Technical Specification, March 2002.
35. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. In *Scientific American*, May 2001.
36. D. Conolly, F. van Harmelen, I. Horrocks, and at al. Daml+oil (march 2001) reference desciption. W3C Note, W3C, December 2001.
37. Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language 1.0 reference. W3C Working Draft, W3C, March 2003.
38. T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible markup language (xml) 1.0. W3C Recommendation, W3C, February 1998.

39. D.C. Fallside. XML Schema Part 0: Primer. W3C Recommendation, W3C, May 2001.
40. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the IEEE Intl Conference on Data Engineering (ICDE)*, 1995.
41. S. Abiteboul, S. Cluet, and T. Milo. Correspondence and Translation for Heterogeneous Data. In *Intl. Conf. on Database Theory*, 1997.
42. A. Deutsch, M. Fernandez, and D. Suciu. Storing Semistructured Data in Relations. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-standard Data Formats*, Jerusalem, Israel, 1999.
43. O. Lassila and R.R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, W3C, February 1999.
44. D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Working Draft, W3C, March 2002. `http://www.w3.org/TR/rdf-schema`.
45. C. Bornhövd. *Semantic Metadata for the Integration of Heterogeneous Internet Data (in German)*. PhD thesis, Darmstadt University of Technology, 2000.
46. C. Bornhövd and A.P. Buchmann. A Prototype for Metadata-Based Integration of Internet Sources. In *Proc. of CAiSE*, volume 1626 of *LNCS*, 1999.
47. N. Paton, editor. *Active Rules in Database Systems*. Springer, 1999.
48. A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler. The WISE approach to electronic commerce. *International Journal of Computer Systems Science and Engineering*, 15(5), 2000.
49. J. Lopes de Oliveira, C. Bauzer Medeiros, and M. Cilia. Active Customization of GIS User Interfaces. In *Proceedings of the IEEE Intl Conference on Data Engineering (ICDE)*, pages 487–496, Birmingham, U.K., April 1997. IEEE Computer Society Press.
50. M. Cilia, P. Hasselmeyer, and A.P. Buchmann. Profiling and Internet Connectivity in Automotive Environments. In *Proceedings of the Intl Conference on Very Lage Data Bases (VLDB)*, pages 1071–1074, Hong-Kong, China, August 2002. Morgan-Kaufmann.
51. S. Chakravarthy, J. Jacob, N. Pandrangi, and A. Sanka. Webvigil: An approach to just-in-time information propagation in large network-centric environments. In *2nd International Workshop on Web Dynamics (In conjunction with WWW2002)*, 2002.
52. G. Alonso, C. Hagen, and A. Lazcano. Processes in electronic commerce. In *ICDCS Workshop on Electronic Commerce and Web-Based Applications (ICDCS 99)*, May 1999.
53. Martin West. The state of business rules. *MiddlewareSpectra*, 15(11), November 2001.
54. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Data Management Systems. Morgan Kaufmann Publishers, 1996.
55. A. Bonifati, S. Ceri, and S. Paraboschi. Active rules for XML: A new paradigm for E-services. *The VLDB Journal*, 10(1):39–47, 2001.
56. James Bailey, Alexandra Poulovassilis, and Peter T. Wood. Analysis and optimisation of event-condition-action rules on XML. *Computer Networks*, 39(3):239–259, 2002.
57. S. Gatziu, A. Koschel, G. v. Buetzingsloewen, and H. Fritschi. Unbundling Active Functionality. *ACM SIGMOD Record*, 27(1):35–40, March 1998.

58. A. Koschel and P. Lockemann. Distributed Events in Active Database Systems - Letting the Genie out of the Bottle. *Data & Knowledge Engineering*, 25(1-2):29–53, March 1998.
59. H. Fritschi, S. Gatziu, and K. Dittrich. FRAMBOISE - an Approach to Framework-based Active Data Management System Construction. In *Proceedings of the seventh on Information and Knowledge Management (CIKM 98)*, pages 364–370, Maryland, November 1998.
60. C. Collet. The NODS Project: Networked Open Database Services. In K. Dittrich et.al., editor, *Object and Databases 2000*, number 1944 in LNCS, pages 153–169. Springer, 2000.
61. A.P. Buchmann. *Architecture of Active Database Systems*, chapter 2, pages 29–48. In Paton [47], 1999. In Paton, N. 1999.
62. A. Buchmann and C. Liebig. Distributed, Object-Oriented, Active, Real-Time DBMSs: We Want It All – Do We Need Them (At) All? *Annual Reviews in Control*, 25, January 2001.
63. P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.
64. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
65. G. Weikum and G. Vossen. *Transactional Information Processing: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2001.
66. A.P. Buchmann, A. Deutsch, J. Zimmermann, and M. Higa. The REACH Active OODBMS. In *Proceedings of the ACM Intl Conference on Management of Data (SIGMOD)*, page 476. ACM Press, 1995.
67. Object Management Group (OMG). Transaction service v1.1. Technical Report OMG Document formal/2000-06-28, OMG, Famingham, MA, May 2000.
68. The Open Group. Distributed Transaction Processing: Reference Model, Version 3 . Technical Report G504, The Open Group, February 1996.
69. A.K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.
70. P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing for Systems Professionals.* Morgan Kaufmann Publishers, 1996.
71. Philip A. Bernstein, Meichun Hsu, and Bruce Mann. Implementing recoverable requests using queues. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the ACM Intl Conference on Management of Data (SIGMOD)*, pages 112–122. ACM Press, 1990.
72. A. Chan. Transactional Publish/Subscribe: The Proactive Multicast of Database Changes. In *Proceedings of the ACM Intl Conference on Management of Data (SIGMOD).* ACM Press, June 1998.
73. S. Tai and I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proceedings of the IFIP/ACM Intl Conference on Distributed Systems Platforms and Open Distribued Processing(Middleware)*, New York, USA, April 2000. Springer-Verlag.
74. Christoph Liebig and Stefan Tai. Middleware mediated transactions. In *Proc. of DOA'00*, 2001.
75. Object Management Group. Management of event domains. Version 1.0, Formal Specification, 2001. formal/01-06-03.

76. Edward Curry, Desmond Chambers, and Gerard Lyons. Reflective channel hierarchies. In *The 2nd Workshop on Reflective and Adaptive Middleware, Middleware 2003*, Rio de Janeiro, Brazil, 2003.
77. William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In Andreas Paepcke, editor, *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, Washington, DC, USA, 1993.
78. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
79. Karl O'Connell, Tom Dinneen, Stephen Collins, Brendan Tangney, Neville Harris, and Vinny Cahill. Techniques for handling scale and distribution in virtual worlds. In Andrew Herbert and Andrew S. Tanenbaum, editors, *Proceedings of the* 7^{th} *ACM SIGOPS European Workshop*, pages 17–24, Connemara, Ireland, September 1996.
80. R. Gruber, B. Krishnamurthy, and E. Panagos. The Architecture of the READY Event Notification Service. In *Proceedings of the 19th IEEE Intl. Conf. on Distributed Computing Systems Middleware Workshop*, Austin, Texas, May 1999. IEEE.
81. L. Fiege, M. Mezini, G. Mühl, and A.P. Buchmann. Engineering Event-Based Systems with Scopes. In *Proc. of ECOOP'02*, volume 2374 of *LNCS*, 2002.
82. P. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *In Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, July 2002.
83. M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), 2002.
84. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.
85. Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions of Software Engineering and Methodology*, 1(3):229–269, July 1992.
86. Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.
87. G. Mühl, L. Fiege, F.C. Gärtner, and A.P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proc. IEEE/ACM MASCOTS'02*, 2002.
88. Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt University of Technology, 2002.
89. Ludger Fiege, Gero Mühl, and Felix C. Gärtner. A modular approach to build structured event-based systems. In *Proc. of ACM SAC'02*, 2002.
90. Edsger W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
91. M. Cilia. *An Active Functionality Service for Open Distributed Heterogeneous Environments*. PhD thesis, Darmstadt University of Technology, 2002.
92. D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the IEEE Intl Conference on Data Engineering (ICDE)*, 1999.
93. Object Management Group. Enhanced View of Time Service, Version 1.1. Tecnical Report formal/02-05-07, (OMG), 2002.

94. M. Cilia, C. Bornhövd, and A. P. Buchmann. Moving Active functionality from Centralized to Open Distributed Heterogeneous Environments. In *Proceedings of the Intl Conference on Cooperative Information Systems (CoopIS)*, volume 2172 of *LNCS*, 2001.
95. C. Liebig, M. Malva, and A.P. Buchmann. Integrating Notifications and Transactions: Concepts and X^2TS Prototype. In *Proc. of EDO*, volume 1999 of *LNCS*, 2000.
96. L. Fiege, M. Mezini, G. Mühl, and A.P. Buchmann. Engineering event-based systems with scopes. In *Proc. of ECOOP'02*, volume 2374 of *LNCS*, 2002.
97. R.E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the READY event notification service. In *Proc. Middleware Workshop ICDCS*, 1999.
98. Oracle, Inc. *Oracle Advanced Queuing (AQ)*, 2002.
99. Gero Mühl. Generic constraints for content-based publish/subscribe systems. In *Proceedings of the Intl Conference on Cooperative Information Systems (CoopIS)*, volume 2172 of *LNCS*, 2001.
100. C. Bornhövd, M. Cilia, C. Liebig, and A.P Buchmann. An Infrastructure for Meta-Auctions. In *Proc. of WECWIS'00*, 2000.
101. M. Cilia and A.P. Buchmann. An active functionality service for e-business applications. *ACM SIGMOD Record*, 31(1):24–30, 2002.