# Challenges for Distributed Event Services: Scalability vs. Expressiveness


Antonio Carzaniga† David S. Rosenblum‡ Alexander L. Wolf†

†Dept. of Computer Science
University of Colorado
Boulder, CO 80309, USA
{carzanig,alw}@cs.colorado.edu

‡Dept. of Info. and Computer Science
University of California
Irvine, CA 92697-3425, USA
dsr@ics.uci.edu


## 1 Introduction

The event-based style is a very promising approach for the development and integration of distributed objects. An *event notification service* (or *event service*) is the glue that ties together distributed components in an event-based architecture. An event service implements what is commonly known as the publish/subscribe protocol: components publish *events* to inform other components of a change in their internal state or to request services from other components; the event service registers the interest of components expressed by means of subscriptions and consequently dispatches event notifications. In practice, the event service mediates and facilitates the interaction among applications by filtering, aggregating, and delivering events on their behalf. Because of this decoupling, an event service is particularly suitable for supporting heterogeneous distributed objects.

The functionality of an event service is characterized by two conflicting requirements: *scalability* and *expressiveness*. Scalability means that the service must be available over a wide-area network populated by numerous components each one producing and consuming many events. Expressiveness demands a rich *subscription language* that gives applications a flexible and fine-grained selection mechanism to describe precisely those events or combinations of events in which they are interested.

This tension between scalability and expressiveness is evident in all the recently proposed technologies. The ones that provide an event service facility (e.g., the CORBA Event Service [4], the Java™ Distributed Event Specification [8], iBus [7], JEDI [2], Keryx [10], Elvin [6], and TIBCO's TIB/Rendezvous™ [9]), as well as other, more mature technologies not explicitly targeted at this problem domain (e.g., the USENET news infrastructure and IP multicast), represent potential or partial solutions. One problem with some of these technologies (CORBA, Java Events, iBus, USENET News, and IP multicast) is that they offer only a limited selection capability, typically based on a predefined set of "channels" or equivalent multicast addresses, that greatly reduces their potential use as a generic event service. On the other hand, the systems that offer a better data model and better event filtering adopt either a classical centralized architecture (Elvin) or a simple extension of the centralized architecture in which the distributed components are connected in a hierarchical structure (JEDI, Keryx and TIB/Rendezvous). While this latter approach is relatively simple and effective in many cases, we argue that it has some fundamental shortcomings when scaling up to wide-area networks. In particular, it introduces unnecessary message traffic, it overloads higher nodes in the hierarchy, and it has a single point of failure in every node.

We believe that the successful integration of distributed objects by means of events depends on both the scalability and the expressiveness of the event service. Here we propose a research approach to this problem that we pursued with our SIENA project [1, 5]. In particular, we focus on how to realize scalable true content-based routing of events over a distributed event service with a generic topology.

## 2 Conceptual model for a Distributed Event Service

### 2.1 Event Model and Subscription Language

Events are represented by a data structure that we call a *notification*. The data model or the encoding schema of notifications is what we call an *event notification model* or simply *event model*. The event model defines what information can be communicated by means of events, or at least how that information must be encoded. Most of the existing event-based systems adopt a record-like structure for notifications, while others allow more sophisticated modeling by exploiting features akin to an object-oriented language.

Tightly related to the event model is the *subscription language* that defines the form of the selection expressions submitted with subscriptions. Two aspects of the subscription language are crucial to the expressiveness of an event service:

- the *scope* of the selection predicates: the part of the event model that is visible within subscription expressions. In some cases, events have an articulated structure that allows the encoding of much information, but only a limited and/or simple part of that structure can be used as a selection criteria in subscriptions.
- the *expressiveness* of the selection predicates: determines the sophistication of subscriptions. In practice, a subscription language is expressive if it has various basic selection predicates and the ability to combine predicates for the selection of one single event at a time as well as for grouping events into higher-level abstractions.

In terms of *scope*, most existing technologies limit the selection to a single well known element of a notification usually called a "channel" or "subject". A few systems allow filtering based on the content of the whole notification. In terms of *expressiveness*, the simplest models allow a single equality test (channel), while the most sophisticated ones allow for other predicates and conjunctions of predicates.

| | | *scope of subscriptions* | |
| --- | --- | --- | --- |
| | | *one field (not structured)* | *multiple fields (structured)* |
| *expressiveness* | *simple equality* | channel-based | simple content-based |
| | *other predicates and expressions* | subject-based | content-based |
| | *multiple events* | subject-based + patterns | content-based + patterns |

Table 1: Classes of Subscription Languages.

Table 1 gives a classification of subscription languages. Note that the difference between "content-based" and "subject-based" is that a channel allows only a straight equality test (e.g., $channel = X$) whereas the subject subsumes richer predicates like wild-card string matching (e.g., *subject*="A∗B"). In both cases, the filter applies to one single (unstructured) element.

### 2.2 Architecture of an event service

Usually, an event service is realized with one or more components called *event servers* (or brokers or dispatchers). The implementation of an event server can be anything from a library to an operating system service to a separate process on the same host or on a remote host. At this point we are not interested in distinguishing these cases. The architecture of an event service is determined by the number of servers, by the topology of connections among them, and by the kind of server-to-server communication protocol. By "communication protocol", we refer to the type and amount of information that event servers exchange. This protocol is obviously implemented on top of some communication mechanism that could range from shared memory to application-level network protocols such as SMTP or HTTP. At this level, standard encoding and/or tunneling techniques can be used, so we do not discuss the details here.

Most existing technologies that have a distributed architecture adopt a hierarchical topology to connect their servers. In this topology every server may be connected as a common client to a "master" server. The protocol that connects two servers is thus the same one that connects clients and servers. So, except for notifications, which can flow from servers to clients and from servers to other lower-level servers, any other information may flow upward in the hierarchy.

Other technologies, such as IP multicast, have an underlying peer-to-peer network with a generic topology. In this architecture, two connected routers (event servers) exchange routing information (subscriptions) and data (notifications) as peers in both directions.

### 2.3 Classification Framework

We can use the subscription language, which determines expressiveness, and the architecture, which influences scalability, as our classification metrics. Values for the subscription language are: "channel", "subject", "content", and "content + patterns". For the architecture, we have the values "centralized", "hierarchical", and "generic peer-to-peer". Table 2 positions several technologies that are related to event-based infrastructures, including our system SIENA, with respect to these two metrics.

| | | *architecture* | | |
|---|---|---|---|---|
| | | *centralized* | *hierarchical* | *generic peer-to-peer* |
| *subscription language* | *channel* | CORBA, Java Field | CORBA, Java | IP multicast, iBus |
| | *subject* | ToolTalk, | NNTP, JEDI, TIBCO | |
| | *content* | Elvin | Keryx | |
| | *content+patterns* | Yeast, GEM, active database | | SIENA |

Table 2: Classification of Event-Based Infrastructures.

## 3 SIENA: Multicast Routing Revisited

### 3.1 Nature of the Event Service: A Routing Problem

In IP multicast [3], a datagram may be addressed to a *host group*—a "virtual" address that refers to a set of "physical" addresses. Hosts can send a datagram with the usual *IP_send* primitive. Hosts can also join (or leave) a group at any time using the special control primitive *JoinHostGroup* (or *LeaveHostGroup*). The job of multicast-enabled routers is to forward every incoming datagram to one or more of their neighbor routers according to (1) destination (and source) address of the datagram and (2) the group membership information, i.e., whether or not a group has members in one of the attached networks. In IP multicast, a special group membership protocol disseminates group membership information among routers.

It is quite evident that, in a distributed event service, the task of an event dispatcher is substantially equivalent to the one of a multicast router. Subscribing corresponds to joining a group and sending a datagram corresponds to publishing an event. Notice how in a channel-based event service these operations are exactly isomorphic. Depending on the type of event service, however, there might be some fundamental differences.

### 3.2 A Fundamental Difference: Content-Based Addressing

In order to understand the new challenges of an event service, we must examine the routing problem in a bit more detail. In very simplistic terms, routing a datagram $D$ means computing the function $\textit{next-hops} = r(\textit{destination}_D, \textit{routing-info})$. Similarly, managing the routing information in response to a control request $C$ (e.g., a *JoinHostGroup*) is done by updating the routing table $\textit{routing-info}' = c(\textit{group}_C, \textit{host}_C, \textit{routing-info})$, possibly forwarding that information to other neighbor routers.

In the case of IP multicast, *routing-info* can be as simple as a table that associates *next-hops* (interfaces) to group addresses. So, $r(\textit{destination}_D, \textit{routing-info})$ is simply a table lookup $\textit{routing-info}(\textit{destination}_D)$. Group membership maintenance is also easy because $\textit{group}_C$ is a key in the *routing-info* table, so when a host joins a group $\textit{group}_C$, either $\textit{group}_C$ is the routing table or it is not. In this latter case, the router propagates the new membership information, while in the first case the propagation is stopped.

This simplification is possible thanks to the fact that, in IP multicast as well as in a channel-based event service, there is a one-to-one mapping (in fact, the identity function) between destination addresses ($\textit{destination}_D$) and group addresses ($\textit{group}_C$). In other words, a datagram/event is explicitly addressed to one specific group/channel.

Content-based addressing is rather different. The correspondence between the "destination address" of a notification, which is in fact its entire content, and a "group address", determined by a subscription, is not as simple to compute and, more importantly, it is not a one-to-one relation, since a notification might well match more than one subscription and vice-versa. Similarly, when propagating new subscriptions (membership information) in content-based addressing, we can no longer rely on the fact that a subscription is a *key* in the subscriptions table since two different subscriptions might define partially overlapping sets of notifications. In this case, it is crucial to be able to compare the new subscription against the old ones to see if there exist one that *covers* the new one completely so that the new one will not need to be propagated.

### 3.3 The SIENA Event Service

In SIENA we combine a content-based subscription language with a distributed realization based on a generic topology of servers.

The event model is a record-like structure consisting of a set of named attributes, similar to a `struct` in C. A *simple* subscription is a conjunction of filters, each one specifying a condition for an attribute. For example, $\textit{stock} = \textit{"DIS"}, \textit{gain} > 10, \textit{gain} < 20$ would select all the events having an attribute named "stock" whose value is "DIS" and an attribute named "gain" whose value is between 10 and 20. SIENA is also capable of observing *compound* events (or *patterns*), i.e., sequences of events. A compound subscription is simply an expression whose elementary terms are simple subscriptions.

SIENA extends the well-known publish/subscribe protocol by introducing another primitive called *advertise*. An *advertisement* is a meta-publication in the sense that it announces the classes of events that an object intends to publish. Advertisements are the dual of subscriptions in that subscriptions declare the intention of receiving notifications, and thus they define the "destination address" of notifications in the routing tables, while advertisements define the "source address" of notifications. Advertisements do not just serve to make the interface complete and symmetrical. The information provided by advertisements can be used to disseminate routing directions more efficiently, thus making the event service more scalable.

### 3.4 Content-Based Routing in SIENA

The routing of notifications in SIENA is based on a generalization of the correspondence between virtual addresses as defined by notifications, subscriptions, and advertisements. We call these correspondences *covering relations*.

A subscription covers a notification when its filter condition is satisfied by the notification. This relation in SIENA embodies the semantics of subscriptions described above and thus involves the evaluation of a conjunction of simple predicates. The covering relation between subscriptions and notifications is used in the routing function. In particular, notifications are forwarded along the paths put in place by subscriptions.

A subscription $x$ covers another subscription $y$ when every notification that is covered by $y$ is also covered by $x$. This relation is used in propagating subscriptions to set up the appropriate routing information. When a server receives a new subscription $y$ it looks for a previously registered subscription $x$ that covers $y$. If such a subscription does not exist, the server propagates an equivalent subscription to its neighbors thereby setting up a forwarding path for future notifications. The covering relation between subscriptions is sensibly more complex than the covering between subscriptions and notifications since it includes a universal quantifier over the set of notifications. However, because SIENA allows only a fixed set of "well-behaved" operators (including the usual relational operators such as $=$, $<$, and $\leq$, plus some simple wild-card string match operations), it is still quite efficient to compute.

Similar covering relations exist between advertisements and subscriptions, as well as between different advertisements. These relations can be used as the basis for a dual routing strategy that floods the network with advertisements and forwards subscriptions only along the paths set up by advertisements. Propagating advertisements is also necessary to realize a distributed observation of patterns of events.

### 3.5 Trading Expressiveness for Scalability

As we have seen, the covering relations play a fundamental role in the observation and dispatching of notifications. Many optimization strategies that can be applied to the dispatching algorithms rely on the covering relations as well [1]. Note that since they are an essential part of any basic routing operation, their relevance goes beyond the implementation of SIENA and extends to any event service.

For the sake of scalability it is therefore necessary that these relations be efficient to compute. On the other hand, the features of the subscription language heavily affect the complexity of the covering relations. As we have seen, in IP multicast they are reduced to equality tests between 32-bit numbers, while in SIENA they entail the evaluation of simple predicates and simple first-order logic expressions.

It is easy to show that adding only a little more expressive power to the subscription language makes some of the relations not computable at all. For example, if we allowed user-defined operators in subscriptions, we would still be able to match notifications against subscriptions, but we would lose the ability to reason about the implications among subscriptions. Thus, we would not be able to verify the covering between subscriptions and, as a consequence, we would be forced to broadcast every new subscription.

## 4 Conclusions

We envision a wide-area event service as an effective platform for the integration of distributed heterogeneous objects. However, in the realization of such an infrastructure we see two major conflicting challenges, namely scalability and expressiveness. The fact that these two are conflicting features is shown by a pattern in current event-based technologies: some of them offer rich selection mechanisms, but with a centralized architecture, while others adopt a more scalable distributed architecture, but they give scarce accuracy in filtering events. We know of no event service besides SIENA that features a scalable architecture *and* a fine-grained selection and aggregation mechanism.

By analyzing the functionality of an event service and by comparing it to the well-known problem of routing, we found that the trade off between scalability and expressiveness is not really specific to any implementation, but rather it is intrinsic to the problem domain. In this paper we also sketched some design solutions that we adopted for SIENA. In particular, we have formulated an event service that combines an expressive API with a generic distributed architecture. The dispatching algorithms that implement the SIENA event service are based on the generalization provided by our analysis.

## Acknowledgments

We would like to thank Gianpaolo Cugola, Elisabetta Di Nitto, Alfonso Fuggetta, Richard Hall, Dennis Heimbigner, and André van der Hoek for their considerable contributions in discussing and shaping many of the ideas presented in this paper.

## References


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

[2] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, Apr. 1998.

[3] S. E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD thesis, Stanford University, Dec. 1991.

[4] Object Management Group. CORBAservices: Common object service specification. Technical report, Object Management Group, July 1998.

[5] D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In *Proceedings of the Sixth European Software Engineering Conference*, Zurich, Switzerland, Sept. 1997. Springer–Verlag.

[6] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, Brisbane, Queensland, Australia, Sept. 3–5 1997.

[7] SoftWired AG, Zurich, Switzerland. *iBus Programmer's Manual*, Nov. 1998. http://www.softwired.ch/ibus.htm.

[8] Sun Microsystems, Inc., Mountain View CA, U.S.A. *Java Distributed Event Specification*, 1998.

[9] TIBCO Inc. Rendezvous information bus. http://www.rv.tibco.com/rvwhitepaper.html, 1996.

[10] M. Wray and R. Hawkes. Distributed virtual environments and VRML: an event-based architecture. In *Proceedings of the Seventh International WWW Conference (WWW7)*, Brisbane, Australia, 1998.