

Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service

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

[†]Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430
USA
{carzanig,alw}@cs.colorado.edu

[‡]Dept. of Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425
USA
dsr@ics.uci.edu

Abstract

This paper describes the design of *SIENA*, an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks. *SIENA* is responsible for *selecting* the notifications that are of interest to clients (as expressed in client subscriptions) and then *delivering* those notifications to the clients via access points. The key design challenge for *SIENA* is maximizing *expressiveness* in the selection mechanism without sacrificing *scalability* of the delivery mechanism. This paper focuses on those aspects of the design of *SIENA* that fundamentally impact scalability and expressiveness. In particular, we describe *SIENA*'s data model for notifications, the covering relations that formally define the semantics of the data model, the distributed architectures we have studied for *SIENA*'s implementation, and the processing strategies we developed to exploit the covering relations for optimizing the routing of notifications.

1 Introduction

There is a clear trend among experienced software developers toward designing large-scale distributed systems as assemblies of loosely-coupled autonomous components. A common approach to achieving loose coupling is an *event-based* or *implicit invocation* design style [7]. In an event-based system, component interactions are modeled as asynchronous occurrences of, and responses to, *events*. To inform other components about the occurrences of internal events (such as state changes), components emit *notifications*

containing information about the events. Upon receiving notifications, other components can react by performing actions that, in turn, may result in the occurrence of other events and the generation of additional notifications.

Wide-area networks such as the Internet, with their vast number of potential producers and consumers of notifications, create an opportunity for developing novel distributed event-based applications in such fields as market analysis, data mining, indexing, and security. In general, the asynchrony, heterogeneity, and inherent high degree of loose coupling that characterize applications for wide-area networks suggest event interaction as a natural design abstraction for a growing class of distributed systems. Yet to date there has been a lack of sufficiently powerful and scalable middleware infrastructures to support event-based interaction in a wide-area network. We refer to such a middleware infrastructure as an *event notification service* [16].

This paper describes the design of *SIENA*,¹ an Internet-scale event notification service that is representative of the capabilities we envision for scalable event notification middleware. *SIENA* is designed to be a ubiquitous service accessible from every site on a wide-area network. As shown in Figure 1, *SIENA* is implemented as a distributed network of servers that provide clients with *access points* offering an extended publish/subscribe interface. The clients are of

¹Scalable Internet Event Notification Architectures.

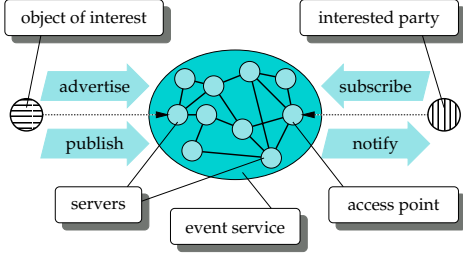


Figure 1: Distributed Event Notification Service.

two kinds: *objects of interest*, which are the generators of notifications, and *interested parties*, which are the consumers of notifications; of course, a client can act as both an object of interest and an interested party. Clients use the access points of their local servers to *advertise* the information about notifications that they generate and to *publish* the advertised notifications. Clients also use the access points to *subscribe* for individual notifications or compound patterns of notifications of interest. *SIENA* is responsible for *selecting* the notifications that are of interest to clients and then *delivering* those notifications to the clients via the access points.

The key design challenge we face in supporting event notification of this kind in a wide-area network setting is maximizing *expressiveness* in the selection mechanism without sacrificing *scalability* of the delivery mechanism. Scalability refers not only to the numbers of publishers and subscribers, and the numbers of notifications and subscriptions they request *SIENA* to process, but also to the need to discard many of the assumptions made for local-area networks, such as low latency, abundant bandwidth, homogeneous platforms, continuous and reliable connectivity, and centralized control. Expressiveness refers to the power of the data model that *SIENA* offers to publishers and subscribers of notifications. Clearly the level of expressiveness influences the algorithms used by *SIENA* to route and deliver notifications, and the extent to which those algorithms can be optimized. As the power of the data model increases, so does the complexity of the algorithms. Therefore, the expressiveness of *SIENA*'s data model ultimately influences

the scalability of *SIENA*'s implementation, and hence scalability and expressiveness are two conflicting goals that must be traded off.

In this paper, our description of *SIENA* focuses on those aspects of the design that fundamentally impact scalability and expressiveness. In designing *SIENA*, we began by designing a data model we believed to be sufficiently expressive for a wide range of applications while still allowing sufficient scalability of the delivery mechanism. Based on this data model, we designed distributed server architectures and associated delivery algorithms and processing strategies, and we evaluated and confirmed their scalability.

2 Data Model and Semantics

As mentioned above, *SIENA* extends the traditional publish/subscribe protocol with an additional interface function called *advertise*, which is used by an object of interest to inform the event service of the nature of the notifications that it might publish. *SIENA* also adds the functions *unsubscribe* and *unadvertise* to further inform the event service about the future behavior of interested parties and objects of interest, respectively. Subscriptions can be matched repeatedly until they are cancelled by a call to *unsubscribe*. Advertisements remain in effect until they are cancelled by a call to *unadvertise*.

SIENA is a *best effort* service in that it does not attempt to prevent race conditions induced by network latency. This is a pragmatic concession to the realities of Internet-scale services, but it means that clients of *SIENA* must be resilient to such race conditions. For instance, clients must allow for the possibility of receiving a notification for a cancelled subscription.

Underlying *SIENA*'s interface is a *notification data model* (or simply *data model*) that drives the semantics of the service. A notification in the model is an untyped set of typed attributes. For example, the notification shown in Figure 2 represents a stock price change event. Each individual attribute has a *type*, a *name*, and a *value*, but the notification as a whole is purely a structural value derived from its attributes. Attribute

<i>string</i>	<i>class</i> = <i>finance/exchanges/stock</i>
<i>time</i>	<i>date</i> = Mar 4 11:43:37 MST 1998
<i>string</i>	<i>exchange</i> = NYSE
<i>string</i>	<i>symbol</i> = DIS
<i>float</i>	<i>prior</i> = 105.25
<i>float</i>	<i>change</i> = -4
<i>float</i>	<i>earn</i> = 2.04

Figure 2: Example of a Notification.

names are simply character strings. The attribute types belong to a predefined set of primitive types commonly found in programming languages and database query languages, and for which a fixed set of operators is defined.

The justification for choosing this typing scheme is scalability: Typed notifications, such as one finds for example in the Java Distributed Event Specification [19] and CORBA Notification Service [14], imply a global authority for managing and verifying the type space, something which is clearly not feasible at an Internet scale. On the other hand, we define a restricted set of attribute types from which to construct (arbitrary) notifications. By having this well-defined set, we can perform efficient routing based on the content of notifications. Content-based routing is a powerful technique for selecting and delivering notifications that gives clients the ability to control the precision with which notifications are selected and gives the event service the ability to optimize the processing tasks required for notification delivery. As we discuss in Section 4 and Section 5, content-based routing has distinct advantages over the alternative schemes of channel- and subject-based routing.

In the remainder of this section we discuss two mechanisms for notification selection, namely *filters* and *patterns*, that form the essence of SIENA’s extended publish/subscribe protocol. This allows us to fully define the semantics of the interface functions, which we do in terms of what we call *covering relations*. In Section 3 we discuss the use of the covering relations to define the processing strategies that lead to optimized notification delivery.

An *event filter*, or simply a *filter*, selects event notifications by specifying a set of attributes

and constraints on the values of those attributes. Each attribute constraint is a tuple specifying a type, a name, a binary predicate operator, and a value for an attribute. The operators provided by SIENA include all the common equality and ordering relations ($=$, \neq , $<$, $>$, etc.) for each of its types, substring ($*$), prefix ($>*$), and suffix ($*<$) operators for strings, and an operator *any* that matches any value.

An attribute $\alpha = (type_\alpha, name_\alpha, value_\alpha)$ matches an attribute constraint $\phi = (type_\phi, name_\phi, operator_\phi, value_\phi)$ iff $type_\alpha = type_\phi \wedge name_\alpha = name_\phi \wedge operator_\phi(value_\alpha, value_\phi)$. We say an attribute α satisfies or *matches* an attribute constraint ϕ with the notation $\phi \sqsubset_f^n \alpha$. When α matches ϕ , we also say that ϕ *covers* α . Figure 3 shows a filter that matches price decreases for stock DIS on stock exchange NYSE.

<i>string</i>	<i>class</i> $>*$ <i>finance/exchanges/</i>
<i>string</i>	<i>exchange</i> = NYSE
<i>string</i>	<i>symbol</i> = DIS
<i>float</i>	<i>change</i> $<$ 0

Figure 3: Example of an Event Filter.

When a filter is used in a subscription, multiple constraints for the same attribute are interpreted as a conjunction—all such constraints must be matched. Thus, we say that a notification n *matches* a filter f , or equivalently that f *covers* n ($f \sqsubset_S^N n$ for short):

$$f \sqsubset_S^N n \Leftrightarrow \forall \phi \in f : \exists \alpha \in n : \phi \sqsubset_f^n \alpha$$

Notice that the notification may contain other attributes that have no correspondents in the filter. Table 1 gives some examples that illustrate the semantics of \sqsubset_S^N . The second example is not a match because the notification is missing a value for attribute *level*. The third example is not a match because the constraints specified for attribute *level* in the subscription are not matched by the value for *level* in the notification.

While a filter is matched against a single notification based on the notification’s attribute data, a *pattern* is matched against one or more notifications based on both their attribute data and on

<i>subscription</i>		<i>notification</i>
<i>string</i> what = alarm	\sqsubset_S^N	<i>string</i> what = alarm <i>time</i> date = 02:40:03
<i>string</i> what = alarm <i>integer</i> level > 3	$\not\sqsubset_S^N$	<i>string</i> what = alarm <i>time</i> date = 02:40:03
<i>string</i> what = alarm <i>integer</i> level > 3 <i>integer</i> level < 7	$\not\sqsubset_S^N$	<i>string</i> what = alarm <i>integer</i> level = 10
<i>string</i> what = alarm <i>integer</i> level > 3 <i>integer</i> level < 7	\sqsubset_S^N	<i>string</i> what = alarm <i>integer</i> level = 5

Table 1: Examples of \sqsubset_S^N .

the combination they form. At its most generic, a pattern might correlate events according to any compound relation. For example, the programmer of a stock market analysis tool might be interested in receiving price change notifications for the stock of one company only if the price of a related stock has changed by a certain amount. Rich languages and logics exist that allow one to express event patterns [12].

In SIENA we do not attempt to provide a complete pattern language. Our goal is rather to study pattern operators that can be exploited to optimize the selection of notifications within the event service. Here, we restrict a pattern to be syntactically a sequence of filters that is matched by a temporally ordered sequence of notifications, each one matching the corresponding filter. An example of a pattern is shown in Figure 4, which matches an increase in the price of stock MSFT followed by a subsequent increase in the price of stock NSCP. In general, we ob-

<i>string</i> what > * finance/exchanges/ <i>string</i> symbol = MSFT <i>float</i> change > 0
.
<i>string</i> what > * finance/exchanges/ <i>string</i> symbol = NSCP <i>float</i> change > 0

Figure 4: Example of an Event Pattern.

serve that more sophisticated forms of patterns can always be split into a set of simple subscriptions and then matched externally to SIENA (i.e., at the access point of the subscriber), although this is likely to induce extra network traffic.

We have shown how the covering relation \sqsubset_S^N defines the semantics of filters in subscriptions. We now define the semantics of advertisements with a similar relation \sqsubset_A^N . The motivation for advertisements is to inform the event service about which kind of notifications will be generated by which objects of interest, so that it can best direct the propagation of subscriptions. The idea is that, while a subscription defines the set of interesting notifications for an interested party, an advertisement defines the set of notifications potentially generated by an object of interest. Therefore, the advertisement is relevant to the subscription only if these two sets of notifications have a non-empty intersection.

The relation \sqsubset_A^N defines the set of notifications covered by an advertisement as follows:

$$a \sqsubset_A^N n \Leftrightarrow \forall \alpha_n \in n : \exists \phi_a \in a : \phi_a \sqsubset_f^n \alpha_n$$

This expression says that an advertisement covers a notification if and only if it covers each individual attribute in the notification. Note that this is the dual of subscriptions, which define the minimal set of attributes that a notification must contain. In contrast to subscriptions, when a filter is used as an advertisement, then multiple constraints for the same attribute are interpreted as a disjunction rather than as a conjunction; only one of the constraints need be satisfied. Table 2 shows some examples of the relation \sqsubset_A^N . The second example is not a match because the attribute *date* of the notification is not defined in the advertisement. The fourth example is not a match because the value of attribute *what* in the notification does not match any of the constraints defined for *what* in the advertisement.

So far we have defined a number of relations that express the semantics of subscriptions and advertisements:

- $\phi \sqsubset_f^n \alpha$: attribute α matches attribute constraint ϕ ;

<i>advertisement</i>		<i>notification</i>
<i>string what = alarm</i> <i>string what = login</i> <i>string user any</i>	\sqsubset_A^N	<i>string what = alarm</i>
<i>string what = alarm</i> <i>string what = login</i> <i>string user any</i>	$\not\sqsubset_A^N$	<i>string what = alarm</i> <i>time date = 02:40:03</i>
<i>string what = alarm</i> <i>string what = login</i> <i>string user any</i>	\sqsubset_A^N	<i>string what = login</i> <i>string user = carzanig</i>
<i>string what = alarm</i> <i>string what = login</i> <i>string user any</i>	$\not\sqsubset_S^N$	<i>string what = logout</i> <i>string user = carzanig</i>

Table 2: Examples of \sqsubset_A^N .

- $f \sqsubset_S^N n$: notification n matches filter f (where f is interpreted as a subscription filter);
- $a \sqsubset_A^N n$: notification n matches filter a (where f is interpreted as an advertisement filter);

From these, other relations can be derived:

- $f_1 \sqsubset_S^S f_2$: filter f_1 covers filter f_2 (where f_1 and f_2 are interpreted as subscriptions). Formally,

$$f_1 \sqsubset_S^S f_2 \Leftrightarrow \forall n : f_2 \sqsubset_S^N n \Rightarrow f_1 \sqsubset_S^N n$$

which means that f_1 defines a superset of the notifications defined by f_2 .

- $a_1 \sqsubset_A^A a_2$: filter a_1 covers filter a_2 (where a_1 and a_2 are interpreted as advertisements). Formally:

$$a_1 \sqsubset_A^A a_2 \Leftrightarrow \forall n : a_2 \sqsubset_A^N n \Rightarrow a_1 \sqsubset_A^N n$$

which means that a_1 defines a superset of the notifications defined by a_2 .

The relations \sqsubset_S^S and \sqsubset_A^A can also define the equality relation between filters with its intuitive meaning:

$$f = g \Leftrightarrow g \sqsubset f \wedge f \sqsubset g$$

In the next section we describe how we exploit these derived relations in *SIENA*'s processing strategies.

Unsubscriptions and unadvertisements cancel previous subscriptions and advertisements, respectively. These operations must be performed in the context of the covering relations so that the proper subscriptions and advertisements remain in place. The details of this are complex and are described elsewhere [2, 3].

3 Architectures and Processing Strategies

As shown in Figure 1, the implementation of *SIENA* comprises a number of interconnected servers,² each serving some subset of the clients of the service. In effect, *SIENA* is a wide-area network of pattern matchers and routers overlaid atop some other wide-area communication facility, such as the Internet. One reasonable allocation of such servers might be to place a server at each administrative domain within the low-level, wide-area communication network.

Creating a network of servers to provide a distributed service of any sort gives rise to three critical design issues:

- *Interconnection topology*. In what configuration should the servers be connected?
- *Routing algorithm*. What information should be communicated between the servers to allow the correct and efficient delivery of messages?
- *Processing strategy*. Where in the network, and according to what heuristics, should message data be processed in order to optimize message traffic?

These three design issues have been studied extensively for many years and in many contexts. Our challenge is to find a solution in the particular domain of Internet-scale event notification, leveraging previous results (both positive and negative) wherever possible.

²Note that some authors use the term *proxy* or *broker* instead of *server* for this concept.

A pair of interconnected servers use a server/server communication protocol that determines what kinds of information they can exchange, and in which direction. This protocol might make use of any one of a number of lower-level network protocols, such as SMTP or HTTP, through standard encoding and/or tunneling techniques. An interconnection topology and a protocol together define what we refer to as an *architecture* for SIENA. We have studied three basic architectures for SIENA: hierarchical client/server, acyclic peer-to-peer, and general peer-to-peer. We also have studied some hybrids of these three architectures. Because it is not scalable, we ignore the degenerate case of a *centralized* architecture having a single server.

In the *hierarchical client/server* architecture, the servers form a hierarchical topology, with each server (except the root server) behaving like a SIENA client of the server one level up the hierarchy. As we have demonstrated elsewhere [3], the main problems exhibited by the hierarchical architecture are the potential overloading of servers high in the hierarchy and the fact that each server is a single point of failure. In the *acyclic peer-to-peer* architecture, servers communicate with each other symmetrically as peers in an acyclic undirected graph, adopting a protocol that allows a bi-directional flow of subscriptions, advertisements, and notifications. Removing the constraint of acyclicity from the acyclic peer-to-peer architecture, we obtain the *general peer-to-peer* architecture, which can have multiple paths of bi-directional communication between servers. Allowing redundant connections makes it more robust with respect to failures of single servers. The drawback of having redundant connections is that special algorithms must be implemented to avoid cycles and to choose the best paths. These three basic architectures can be combined to form hybrid architectures, such as an acyclic peer-to-peer topology of subnets, each subnet being a hierarchy.

Once a topology of servers is defined, they must establish appropriate routing paths to ensure that notifications published by an object of interest are correctly delivered to all the inter-

ested parties that subscribed for them. In general, we observe that notifications must “meet” subscriptions somewhere in the network so that the notifications can be selected according to the subscriptions and then dispatched to the subscribers. This common principle can be realized according to a spectrum of possible routing algorithms. One possibility is to maintain subscriptions at their access point and to broadcast notifications throughout the whole network; when a notification meets and matches a subscription, the subscriber is immediately notified locally. However, since we expect the number of notifications to far exceed the number of subscriptions or advertisements, this strategy appears to offer the least possible efficiency, and so we consider it no further for SIENA.

To devise more efficient routing algorithms, we employ principles found in IP multicast routing protocols [6]. Similar to these protocols, the main idea behind the routing algorithms of SIENA is to send a notification only towards event servers that have clients that are interested in that notification, possibly using the shortest path. The same principle applies to patterns of notifications as well. More specifically, we formulate two generic principles that become requirements for our routing algorithms:

downstream replication: A notification should be routed in one copy as far as possible and should be replicated only downstream, that is, as close as possible to the parties that are interested in it.

upstream evaluation: Filters are applied and patterns are assembled upstream, that is, as close as possible to the sources of (patterns of) notifications.

These principles are implemented by two classes of routing algorithms, the first of which involves broadcasting subscriptions and the second of which involves broadcasting advertisements:

subscription forwarding: In an implementation that does not use advertisements, the routing paths for notifications are set by subscriptions, which are propagated

throughout the network so as to form a tree that connects the subscribers to all the servers in the network. When an object publishes a notification that matches that subscription, the notification is routed towards the subscriber, following the reverse path put in place by the subscription.

advertisement forwarding: In an implementation that uses advertisements, it is safe to send a subscription only towards those objects of interest that intend to generate notifications relevant to that subscription. Thus, advertisements set the paths for subscriptions, which in turn set the paths for notifications. Every advertisement is propagated through the network, thereby forming a tree that reaches every server. When a server receives a subscription, it propagates the subscription in reverse, along the path to the advertiser, thereby *activating* that path. Notifications are then forwarded only through the activated paths.

In the simplest implementation of the subscription forwarding (advertisement forwarding) algorithms, all subscriptions (advertisements) would be stored at all servers. However, we can optimize their implementation by exploiting the covering relations defined in Section 2. When a subscription reaches a server (either from a client or from another server), the server propagates that subscription only if it defines new selectable notifications that are not in the set of selectable notifications defined by any previously propagated subscription. Three benefits accrue from this approach: First, we reduce network costs by pruning the propagation of new subscriptions. Second, we reduce the storage requirements for servers. Third, by reducing the number of subscriptions held at each server, we reduce the computational resources needed to match notifications at that server. We use a similar strategy for propagation of advertisements.

Hence, in the process of forwarding subscriptions or advertisements, *SIENA* exploits commonalities among subscriptions or advertisements. In practice, *SIENA* prunes the propagation trees

by following only those paths that have not been covered by previous requests. The derived covering relations \sqsubset_S^S and \sqsubset_A^A defined in Section 2 are used to determine whether a new subscription or a new advertisement is covered by a previous one that has already been forwarded.

Unsubscriptions and unadvertisements are handled in a similar way to undo the effect of the affected subscriptions or advertisements. To match patterns, servers assemble sequences of notifications from small sub-sequences or from single notifications according to the advertised paths along which notifications will be propagated. For this reason, advertisement forwarding algorithms are necessary to implement the *upstream evaluation* principle for event patterns.

The interested reader is referred to our other publications [2, 3] for details on how we apply these processing strategies to the different architectures. These publications also present results of simulation studies showing how the different architectures and processing strategies perform over an extensive range of scenarios having different client distributions and processing loads.

4 Analysis

Consider two extremes of expressiveness. In a *channel-based* model of event notification, notifications are fed into what amounts to a discrete communications pipe. Subscriptions are made by simply identifying the pipe (i.e., channel) from which notifications are expected to flow; the notion of “filtering” then reduces to channel selection. Since the contents of notifications are not used in routing, it is not necessary to define any service-visible structure within notifications. The covering relations become an equality check on the identifier of the channel, thus making the routing of notifications very efficient. However, the resulting notification selection mechanism is simplistic, and too weak for some applications.

At the opposite extreme, the structure of notifications, the types of attributes within notifications, and the operators that can be applied to those attributes are all application defined, perhaps employing the full expressive power of

a Turing-complete language. However, the operators, which are used by the service to perform notification selection, would then be of an arbitrary, unknown, and potentially unbounded complexity. Moreover, the computation of the covering relations that allow the pruning of propagation trees, such as \sqsubset_S^S , might be undecidable.

These considerations led us to a level of expressiveness in *SIENA* at which notification structure, attribute types, and attribute operators approximate those of the well-understood and widely-used language SQL. In particular, *SIENA* supports the definition of filters that essentially implement a significant subset of the SQL *select* query.

The covering relations are well behaved and predictable in the sense that they exhibit an arguably reasonable computational complexity deriving from the expressiveness of filters: Assuming a brute-force and unoptimized algorithm, the complexity of computing \sqsubset_S^N on a given subscription and a given notification is $O(n + m)$, where n is the number of attribute constraints in the subscription filter, m is the number of attributes in the notification. The complexity of each individual comparison is $O(1)$ for all the predefined types we have included in *SIENA*. The only exception is for the string type, but efficient comparison algorithms are well known. The complexity of computing \sqsubset_S^N reflects the computation of an intersection between the attribute values in a notification and constraints on those values appearing in a subscription.

The complexities of computing \sqsubset_S^S , \sqsubset_A^A , and \sqsubset_A^S are all $O(nm)$, where n and m represent the number of attribute constraints appearing in the respective subscription and/or advertisement filters. This complexity represents a comparison between each attribute constraint in one filter and any corresponding attribute constraints in the other filter. Checking a covering relation between filters amounts to a universal quantification. But given our choice of types and operators, comparing a pair of attribute constraints can be reduced to evaluating an appropriate predicate on the two constant values of the constraints, with a complexity $O(1)$. For example,

to see if $[x > k_1]$ covers $[x > k_2]$ we can simply verify that $k_2 \geq k_1$.

We also restricted the expressiveness of patterns in *SIENA* in the interests of efficiency. As we discuss in Section 2, patterns are simple sequences of filters. The computational complexity of recognizing a pattern is $O(l(n + m))$, where l is the length of the pattern. This means that it is linear in the number of filters, whose covering relation \sqsubset_S^N has complexity $O(n + m)$.

Our conclusion from this analysis is that the optimizations presented in Section 3 are effective, since they derive from the reasonable complexity of the covering relation computations. In fact, the factors n and m are, in practice, likely to be relatively small, making the computations negligible compared to the network costs they are attempting to reduce. This is all achieved with an expressiveness that approximates SQL.

5 Conclusion

In this section we briefly review related work in event notification services and discuss our prototype implementation of *SIENA*. A more complete discussion of these topics is presented elsewhere [2, 3].

We can compare related technologies from the perspective of their server architectures, which affects scalability, and from the perspective of their subscription language, which affects expressiveness. Table 3 presents such a comparison in terms of the architectures described in Section 3 and in terms of a classification of subscription languages shown in Table 4.

We classify subscription languages based on their *scope* and *expressive power*. Scope has two aspects: (1) whether a subscription is limited to considering a single notification (thus reducing the language to that of filters) or whether it can consider multiple notifications (thus involving both filters and patterns); and (2) whether a subscription is limited to considering a single, designated field in a notification or whether it can consider multiple fields. Expressive power is concerned with the sophistication of operators that can be used in forming subscription predi-

		Architecture		
		<i>centralized</i>	<i>hierarchical client/server</i>	<i>general peer-to-peer</i>
Subscription Language	<i>channel-based</i>	Field [15] CORBA Event Service [13] Java Dist. Event Spec. [19]	CORBA Event Service [13]	IP multicast [6] iBus [18]
	<i>subject-based</i>	ToolTalk [9]	NNTP [10] JEDI [5] TIB/Rendezvous [20]	(none)
	<i>content-based</i>	Elvin [17]	Keryx [21] Yu et al. [22]	Gryphon [1]
	<i>content-based with patterns</i>	GEM [12] Yeast [11] CORBA Notification Service [14] object-oriented active databases [4]	SIENA	SIENA

Table 3: A Classification of Related Technologies.

		Scope		
		<i>Single Notification Single Field</i>	<i>Single Notification Multiple Fields</i>	<i>Multiple Notifications Multiple Fields</i>
Power	<i>Simple Equality</i>	channel-based	—	—
	<i>Expressions with Predefined Operators</i>	restricted subject-based	restricted content-based	restricted content-based with patterns
	<i>Expressions with User-defined Operators</i>	general subject-based	general content-based	general content-based with patterns

Table 4: Typical Features of Subscription Languages.

icates, ranging from a simple equality predicate to expressions involving only predefined operators to expressions involving user-defined operators. As we point out in Section 4, user-defined operators suffer from the disadvantage of having arbitrary, unknown, and potentially unbounded complexity. In fact, we have observed that subscription languages with user-defined predicates are rare; in Table 3 we have combined the language classes corresponding to predefined and user-defined predicates because only a single entry, object-oriented active databases, makes use of user-defined predicates.

We have implemented a prototype of *SIENA*³ and used it as the event service of an agent-based, wide-area software deployment system called the SoftwareDock [8]. The current implementation of *SIENA* offers two APIs, one for C++ and the other for Java. Both interfaces provide the data model and subscription lan-

guage described in Section 2. Two event servers are also provided by the current implementation. One (written in Java) is based on the hierarchical client/server algorithm, while the other one (written in C++) is based on the acyclic peer-to-peer architecture with the subscription forwarding algorithm. These two types of servers have been used together (thus forming a hybrid topology) in the SoftwareDock.

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

³<http://www.cs.colorado.edu/ser1/dot/siena.html> paper.

References

- [1] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, Austin, TX USA, May 1999.
- [2] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998.
- [3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, Oct. 1999.
- [4] S. Ceri and J. Widom. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Mateo, 1996.
- [5] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, To appear.
- [6] S. E. Deering and D. R. Cheriton. Multicast routing in datagram networks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–111, May 1990.
- [7] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of VDM '91: 4th International Symposium of VDM Europe on Formal Software Development Methods*, pages 31–44, Noordwijkerhout, The Netherlands, Oct. 1991. Springer-Verlag.
- [8] R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide-area network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore MD, U.S.A., May 1997.
- [9] A. M. Julienne and B. Holtz. *ToolTalk and open protocols, inter-application communication*. Prentice-Hall, Englewood Cliffs, New Jersey, 1994.
- [10] B. Kantor and P. Lapsley. Network news transfer protocol—a proposed standard for the stream-based transmission of news. Internet Requests For Comments (RFC) 977, Feb. 1986.
- [11] B. Krishnamurthy and D. S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857, Oct. 1995.
- [12] M. Mansouri-Samani and M. Sloman. GEM: A generalized event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.
- [13] Object Management Group. CORBA services: Common object service specification. Technical report, Object Management Group, July 1998.
- [14] Object Management Group. Notification service. Technical report, Object Management Group, Nov. 1998.
- [15] S. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–66, July 1990.
- [16] 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*, number 1301 in Lecture Notes in Computer Science, pages 344–360. Springer-Verlag, 1997.
- [17] 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.
- [18] SoftWired AG, Zurich, Switzerland. *iBus Programmer's Manual*, Nov. 1998. <http://www.softwired.ch/ibus.htm>.
- [19] Sun Microsystems, Inc., Mountain View CA, U.S.A. *Java Distributed Event Specification*, 1998.
- [20] TIBCO Inc. Rendezvous information bus. <http://www.rv.tibco.com/rvwhitepaper.html>, 1996.
- [21] 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.
- [22] H. Yu, D. Estrin, and R. Govindan. A hierarchical proxy architecture for Internet-scale event services. In *Proceedings of WETICE '99*, Stanford, CA, June 1999.