Interfaces and Algorithms for a Wide-Area Event Notification Service

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

University of Colorado
Department of Computer Science
Technical Report CU-CS-888-99
October 1999
(revised May 2000)

Abstract

The components of a loosely-coupled system are typically designed to operate by generating and responding to asynchronous events. An *event notification service* is an application-independent infrastructure that supports the construction of event-based systems, whereby generators of events publish event notifications to the infrastructure and consumers of events subscribe with the infrastructure to receive relevant notifications. The two primary services that should be provided to components by the infrastructure are notification selection (i.e., determining which notifications match which subscriptions) and notification delivery (i.e, routing matching notifications from publishers to subscribers). Numerous event notification services have been developed for local-area networks, generally based on a centralized server to select and deliver event notifications. Therefore, they suffer from an inherent inability to scale to wide-area networks, such as the Internet, where the number and physical distribution of the service's clients can quickly overwhelm a centralized solution. The critical challenge in the setting of a wide-area network is to maximize the expressiveness in the selection mechanism without sacrificing scalability in the delivery mechanism.

This paper presents SIENA, an event notification service that we have designed to exhibit both expressiveness and scalability. We describe the service's interface to applications, the algorithms used by networks of servers to select and deliver event notifications, and the strategies used to optimize performance. We present results of simulation studies that examine the scalability and performance of the service. Finally, we describe a prototype implementation of SIENA.

1 Introduction

There is a clear trend among experienced software developers toward designing network-based software systems as assemblies of loosely-coupled components. A promising approach to supporting component-based systems is the so-called *event-based* or *implicit invocation* architectural style [17]. Under this style, component interactions are modeled as asynchronous occurrences of, and responses to, *events*: To inform other components about the occurrence of an internal event, such as a state change, components will emit notifications containing information about that event; upon receiving a notification, components can react by performing actions that, in turn, may result in the occurrence of other events.

In general, the primary advantage of using an event-based architecture, as opposed to an explicit invocation mechanism (such as procedure invocation), is that it reduces static dependencies among components and facilitates system evolution. This becomes particularly important in situations where components are produced by different vendors or where the system must evolve in the field by the addition, removal, or replacement of components.

Wide-area networks, such as the Internet, offer further motivations for adopting an event-based style. For one thing, the vast number of potential generators of events creates an opportunity for the development of novel applications that can effectively fuse the information associated with different events. Examples are market analysis, data mining, indexing, and security. Moreover, many existing applications that are already designed around the notion of event interaction can be increased in scale through the global connectivity provided by a wide-area network. For example, event-based workflow systems can be federated across the multiple physical locations of a large company or even across corporate boundaries. In general, the asynchrony, heterogeneity, and inherent high degree of loose coupling that characterize wide-area-network applications promote event interaction as a natural design abstraction for a growing class of software systems.

The glue that ties components together in an event-based architecture is an infrastructure that we call an *event notification service* [31]. The most primitive event notification service is one that provides facilities for notifications to be delivered to explicitly addressed components. A somewhat more sophisticated service receives notifications from components and delivers those notifications to any and all other components in the system. Yet more sophisticated is an event notification service that allows components to register interest in particular kinds of notifications and then have only the notifications of interest delivered to them. Examples of these three services are electronic mail, network news, and Internet content channels, respectively. All of these are special-purpose services, however, and thus are unsuitable as an infrastructure upon which to build arbitrary event-based systems.

We envision a ubiquitous event notification service accessible from every site on a wide-area network and suitable for supporting highly distributed applications requiring component interactions ranging in granularity from fine to coarse. Conceptually, the service is implemented as a network of servers that provide access points to clients. Clients use the access points to *advertise* the information about events that they generate and to *publish* notifications containing that information. They also use the access points to *subscribe* for notifications of interest. The service uses the access points to then *notify* clients by delivering any notifications of interest. Clearly, an event notification service complements other general-purpose middleware services, such as point-to-point and multicast communication mechanisms, by offering a many-to-many communication and integration facility.

The event notification service can carry out a *selection* process to determine which of the published notifications are of interest to which of its clients, routing and delivering notifications only to those clients that are interested. In addition to serving clients' interests, the selection process also can be used by the event notification service to optimize communication within the network. The information that drives the selection process originates with clients. More specifically, the event notification service may be asked to apply a *filter* to the contents of event notifications, such that it will deliver only notifications that contain certain specified data values. The selection process may also be asked to look for *patterns* of multiple events, such that it will deliver only sets of notifications associated with that pattern of event occurrences (where each individual event occurrence is matched by a filter).

Given that the primary purpose of an event notification service is to support notification selection and delivery, the challenge we face in a wide-area setting is maximizing *expressiveness* in the selection mechanism without sacrificing *scalability* in the delivery mechanism [6]. Expressiveness refers to the ability of the event notification service to provide a powerful data model with which to capture information about events, to express filters and patterns on notifications of interest, and to use that data model as the basis for optimizing notification delivery. In terms of scalability, we are referring not simply to the number of event generators, the number of event notifications, and the number of notification recipients, 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.

Intuitively, a simple event notification service that provides no selection mechanism can be reduced to a multicast routing and transport mechanism for which there are numerous scalable implementations. However, once the service provides a selection mechanism, then the overall efficiency of the service and its routing of notifications are affected by the power of the language used to construct notifications and to express filters and patterns. As the power of the language increases, so does the complexity of the processing. Thus, in practice, scalability and expressiveness are two conflicting goals that must be traded off.

Numerous event notification services have been developed for local-area networks, generally based on a centralized server to process event notifications (e.g., the UNIX tool `cron`, the environment Field [30], and the event-action system Yeast [22]). This architecture results in their inherent inability to scale to wide-area networks, such as the Internet, where the number and physical distribution of the service's clients can quickly overwhelm a centralized solution. Recent efforts directed at developing distributed event notification services are based on simplistic extensions to centralized event services in which the servers are connected in a hierarchical structure (e.g., see JEDI [10], or the commercial middleware product TIB/Rendezvous™ from TIBCO). We argue that this approach has fundamental shortcomings, and we investigate more appropriate alternatives.

This paper presents SIENA, an event notification service that we have designed to maximize both expressiveness and scalability. In Section 3 we describe the service's formally defined application programming interface (API), which is an extension of the familiar publish/subscribe protocol [3]. Several candidate server topologies and protocols are presented in Section 4. We then describe in Section 5 the routing algorithms used by the service to deliver event notifications to clients; these algorithms are designed for networks of peer-to-peer event servers. This is followed by a description of strategies for optimizing the performance of the notification selection process. Supported in part by the results of simulation studies, we present an analysis of the scalability of our design choices in Section 6. We conclude in Sections 7 and 8 with a discussion of related work and a brief indication of our future plans.

2 Framing the Problem and Its Solution

As discussed in Section 1, an event notification service implements two key activities, *notification selection* and *notification delivery*. A naive approach to realizing these activities is to employ a central server where all subscriptions are recorded, where all notifications are initially targeted, where notifications are evaluated against all subscriptions, and from where notifications are sent out to all relevant subscribers. This solution is logically very simple, but is impractical in the face of scale. Clearly, the service instead must be architected as a distributed system in which activities are spread across the network, hopefully exploiting some sort of locality, and hopefully exhibiting a reasonable growth in complexity.

In its most general form, a distributed event notification service is composed of interconnected *servers*, each one serving some subset of the clients of the service, as shown in Figure 1. (Some use the terms *proxy* and *broker* instead of the term *server*.) The clients are of two kinds: *objects of interest*, which are the generators of events, and *interested parties*, which are the consumers of event notifications. Of course, a client can act as both an object of interest and an interested party. Both kinds of clients interact with a locally-accessible server, which functions as an access point to the network-wide service. In practice, the service becomes a wide-area network of pattern matchers and routers, overlaid on top of 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.

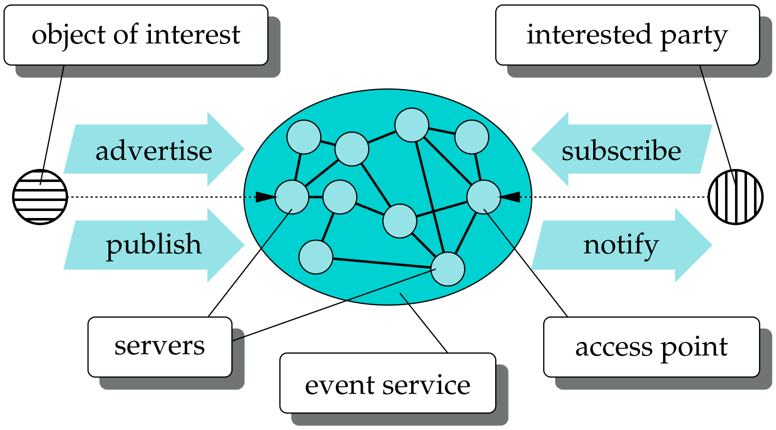

Figure 1: Distributed Event Notification Service.

- *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 wide-area event notification, leveraging previous results (both positive and negative) wherever possible.

In terms of interconnection topology, there are essentially two broad classes from which to choose: a hierarchy and a general graph. Existing distributed event notification services, such as JEDI [10] and TIBCO's product TIB/Rendezvous™, adopt a hierarchical topology. However, our analysis (presented in Section 6) shows that such a topology can exhibit significant performance problems. In SIENA we have adopted the general graph, which in common terms means that the servers are organized in a peer-to-peer relationship, as we detail in Section 4. A hybrid of the two structures—whether a hierarchy of peers, or peers of hierarchies—is also a topology to consider, but requires *a priori* knowledge of the inherent structure of the service's applications in order to make a proper subdivision among peers and hierarchies. Having such knowledge would violate the notion that the service is general purpose.

Our desire for the event notification service to be general purpose also complicates the routing problem for the service. In particular, we assume that objects of interest have no knowledge of interested parties. Therefore, event notifications cannot be addressed and routed in the same, relatively simple manner as, for example, an electronic mail message. Moreover, we cannot assume any particular locality of objects of interest and interested parties, which is a fact that bears a strong relationship to the server topology issue. At best we can only try to take advantage of any locality or structure in the message traffic as it emerges. We demonstrate below that advertisements and subscriptions serve as the basis for this.

Given these considerations, solving the routing problem can be seen as a choice among three alternatives. Common to the three alternatives is the need to broadcast some piece of information to all the servers in the network, where the broadcast is required by the lack of *a priori* knowledge of locality. The first alternative broadcasts notifications, which implies that notification matching is performed at each local server based on the subscriptions received at that server. This alternative suffers from the drawback that all notifications are delivered to all local servers, whether or not they are serving any parties interested in the notifications.

The second and third alternatives try to take advantage of emergent locality and structure. In particular, the second alternative involves a broadcast of subscriptions. A shortest-path algorithm is used to route notifications back to only the local servers of interested parties. Under the third alternative, advertisements are broadcast and subscriptions are then used to establish paths, akin to virtual circuits, by which notifications are routed to only the local servers of interested parties. Of course, both these alternatives suffer from the cost of having to store either all subscriptions or all advertisements at all servers. The drivers that trade off among the three alternatives are fairly straightforward to identify, but in the design of a general-purpose service, any choice will be suboptimal for some situation, as we discuss in Section 5.

Fortunately, we can improve the situation considerably by being intelligent about how we allocate the notification matching tasks within the network. This is the design issue that concerns the processing strategy. We observe that in practice many parties are interested in "similar" events. Put another way, it is likely that distinct subscriptions define partially, or even completely, overlapping sets of notifications. A similar observation can be made about objects of interest and their advertisements. We therefore sketch how this observation leads to a processing strategy for subscriptions and assume a corresponding strategy exists for advertisements; Section 5 presents a detailed discussion of these strategies.

Based on our observation about the likely relationship among subscriptions, the strategy works as follows. 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.

Up to this point in the discussion we have treated notifications, subscriptions, and advertisements in rather abstract terms. We now make these concepts somewhat more concrete as a basis for the material presented in the next several sections.

As mentioned in the introduction, the information associated with an event is represented by a data structure called a notification. We refer to the data model or encoding schema of notifications as the *event notification model* or simply *event model*. Most existing event notification services adopt a simple record-like structure for notifications, while some more recent frameworks define an object-oriented model (e.g., the Java™ Distributed Event Specification [34] and JEDI [10]).

Closely related to the event model is the *subscription language*, which defines the form of the expressions associated with subscriptions. Two aspects of the subscription language are crucial to the issue of expressiveness.

- *Scope* of the subscription predicates.

  This aspect is concerned with the visibility that subscriptions have into the contents of a notification. For a record-like notification structure, visibility determines which fields can be used in specifying a subscription.

- *Power* of the subscription predicates.

  This aspect is concerned with the sophistication of operators that can be used in forming subscription predicates. The predicates apply both to any possible filtering of individual notifications as well as to any possible formation of patterns of multiple notifications.

The dual of the subscription language is the *advertisement language*, which shares the issues of scope and power, but from the perspective of an object of interest, rather than an interested party. One consequence of this difference in perspective is that interested parties may subscribe for patterns of multiple notifications, whereas objects of interest advertise only individual notifications.

The following sections elaborate on these basic concepts and our approach to achieving expressiveness and scalability.

3 API and Semantics

At a minimum, an event notification service exports the functions of Table 1, which define what is usually

publish(*notification*)
subscribe(*handler*, *expression*)

Table 1: Basic Interface of an Event Notification Service.

referred to as the *publish/subscribe protocol*. Interested parties specify the events in which they are interested by means of the function *subscribe*. Objects of interest publish *notifications* via the function *publish*, and the event notification service will take care of delivering the notifications to the interested parties that subscribed for them. The *expression* given to *subscribe* determines which notifications are selected for forwarding to a *handler*. The handler specifies the means by which the interested party receives notifications, through callbacks or through messages sent via a communication protocol such as HTTP or SMTP.

Note that the interested parties and objects of interest are the external clients of the event notification service; hence, we view this protocol as also being a client/server protocol.

3.1 Interface of SIENA

SIENA extends the publish/subscribe protocol with an additional interface function called *advertise*, which an object of interest uses to advertise the notifications it publishes. SIENA also adds the functions *unsubscribe* and *unadvertise*. 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*.

Table 2 shows the interface functions of SIENA. Note that the expression given to *subscribe* and *unsub-*

publish(notification *n*)
subscribe(string *identity*, pattern *expression*)
unsubscribe(string *identity*, pattern *expression*)
advertise(string *identity*, filter *expression*)
unadvertise(string *identity*, filter *expression*)

Table 2: Interface of SIENA.

scribe is a *pattern*, while the expression given to *advertise* and *unadvertise* is a *filter*; we discuss patterns and filters in greater detail below. In all functions, the parameter *identity* specifies the identity of the object of interest or interested party. Objects of interest and interested parties must identify themselves to SIENA when they advertise or subscribe, respectively, so that they can later cancel their own advertisements or subscriptions. The only requirement that SIENA imposes on identifiers is that they be unique.

SIENA maintains a mapping between identities and handlers. Separating these two concepts at the level of clients allows for the possibility of redirecting and/or temporarily suspending the flow of notifications from objects of interest to interested parties.[1] The mapping between identities and handlers is maintained by means of three auxiliary interface functions: **map_identity**(string *identity*, string *handler*), which associates a handler with an identity; **suspend**(string *identity*), which suspends delivery of notifications; and **resume**(string *identity*), which resumes the delivery of notifications. We do not discuss these functions further, since they are simply a convenience and do not materially affect the subject of this paper.

[1] An analogous mechanism can be implemented to support mobile or temporarily disconnected clients. A more in-depth treatment of client mobility is beyond the scope of this paper and is something we are planning for future work.

3.2 Notifications

An *event notification* (or simply a *notification*) is an untyped set of typed attributes. For example, the notification displayed in Figure 2 represents a stock price change event.

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

Figure 2: Example of a Notification.

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 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 [34] and CORBA Notification Service [27], 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. As we discuss in Section 7, content-based routing has distinct advantages over the alternative schemes of channel- and subject-based routing.

3.3 Filters

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 (=, ≠, <, >, etc.) for all 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 $\phi = (type_\phi, name_\phi, operator_\phi, value_\phi)$ if and only if $type_\alpha = type_\phi \wedge name_\alpha = name_\phi \wedge operator_\phi(value_\alpha, value_\phi)$. We say an attribute α satisfies or *matches* an attribute constraint ϕ with the notation $\phi \sqsubset \alpha$. When α matches ϕ, we also say that ϕ *covers* α. Figure 3 shows a filter that matches price increases for stock DIS on stock exchange NYSE.

string	*class*	$>*$	*finance/exchanges/*
string	*exchange*	=	*NYSE*
string	*symbol*	=	*DIS*
float	*change*	>	*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):

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

A filter can have two or more attribute constraints with the same name, in which case the matching rule applies to all of them. Also, the notification may contain other attributes that have no correspondents in the filter. Table 3 gives some examples that illustrate the semantics of $\sqsubseteq_S^N$. The second example is not a match

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

Table 3: Examples of $\sqsubseteq_S^N$.

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.

3.4 Patterns

While a filter is matched against a single notification based on the notification's attribute values, a *pattern* is matched against one or more notifications based on both their attribute values and on the combination they form. At its most generic, a pattern might correlate events according to any 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 [23].

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 notification service. Here, we restrict a pattern to be syntactically a sequence of filters, $f_1 \cdot f_2 \cdots f_n$, 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 observe that more sophisticated forms of

string what >∗ *finance/exchanges/*
string symbol = *MSFT*
float change > *0*

•

string what >∗ *finance/exchanges/*
string symbol = *NSCP*
float change > *0*

Figure 4: Example of an Event Pattern.

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 interested party), although this is likely to induce extra network traffic. We say that a pattern is *simple* when it is composed of a single filter, and similarly we say that a subscription is *simple* when it requests a *simple* pattern.

There are many possible semantics for the filter sequence operator. In the interests of scalability, we have opted for the simplest possible semantics, which ignores out-of-order matches of notifications due to network latency (see Section 3.9). To understand the semantics we chose, consider the pattern **A**·**B** (read "**A** followed by **B**"), which we assume to be submitted as a subscription at time t_0. We represent notifications that match **A** as A_i^j, meaning the notification was generated at time t_i by the object of interest and was matched at time t_j by the server responsible for matching the pattern (and similarly for notifications B_i^j matched to **B**). Consider the following sequence of notifications (shown in match order):

$$B_4^1 \;\; A_3^2 \;\; A_1^3 \;\; B_2^4 \;\; A_5^5 \;\; B_6^6 \;\; A_7^7 \;\; B_8^8$$

According to the semantics we chose, the server matching **A**·**B** uses the first A_i^j it matches followed by the first B_k^m it matches to form the first match of the pattern, such that $i < k$ and $j < m$. It then uses the next A it matches followed by the next B it matches to form the second match of the pattern, and so on. Hence, the first match of the pattern would be the sequence $A_3^2 \cdot B_6^6$, and the second match would be the sequence $A_7^7 \cdot B_8^8$. The matcher receives B_4^1 first but discards it because it has not yet matched an A. The first A it matches is A_3^2, and so it ignores all subsequent As until it matches a B_k^m where $k > 3$. Thus it ignores A_1^3 and A_5^5 because it is waiting for a B, and it also ignores B_2^4 because it was generated before A_3^2. Hence B_6^6 is the first B that can be matched with A_3^2. Once this whole sequence has been matched, the matching of the pattern begins anew with the next A following B_6^6, which is A_7^7. The second match of the pattern is completed with B_8^8.

3.5 Advertisements

We have seen how the covering relation $\sqsubset_S^N$ defines the semantics of filters in subscriptions. We now define the semantics of advertisements by defining a similar relation $\sqsubset_A^N$. The motivation for advertisements is to inform the event notification 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:

$$a \sqsubset_A^N n \Leftrightarrow \forall \alpha \in n : \exists \phi \in a : \phi \sqsubset \alpha$$

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, 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 4 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.

3.6 Two Variants of the Semantics of SIENA

We have studied two alternative semantics for SIENA, a *subscription-based* semantics and an *advertisement-based* semantics.

Under the subscription-based semantics, the semantics of subscriptions is defined solely by the relation $\sqsubset_S^N$ and its extensions to patterns. Advertisements are not enforced on notifications—they may be used for optimization purposes, or they can be ignored completely by the implementation of the service. Thus, a notification n is delivered to an interested party X if and only if X submitted at least one subscription s such that $s \sqsubset_S^N n$.

advertisement		notification
string *what* $=$ *alarm* *string* *what* $=$ *login* *string* username any	$\sqsubset^N_A$	*string* *what* $=$ *alarm*
string *what* $=$ *alarm* *string* *what* $=$ *login* *string* username any	$\not\sqsubset^N_A$	*string* *what* $=$ *alarm* *time* *date* $=$ *02:40:03*
string *what* $=$ *alarm* *string* *what* $=$ *login* *string* username any	$\sqsubset^N_A$	*string* *what* $=$ *login* *string* *username* $=$ *carzanig*
string *what* $=$ *alarm* *string* *what* $=$ *login* *string* username any	$\not\sqsubset^N_A$	*string* *what* $=$ *logout* *string* *username* $=$ *carzanig*

Table 4: Examples of $\sqsubset^N_A$.

Under the advertisement-based semantics, both advertisements and subscriptions are used. In particular, a notification n published by object Y is delivered to interested party X if and only if Y advertised a filter a that covers n (i.e., such that $a \sqsubset^N_A n$) and X registered a subscription s that covers n (i.e., such that $s \sqsubset^N_S n$).

Under both semantics, a notification is delivered at most once to any interested party.

3.7 Other Important Covering Relations

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

- $\phi \sqsubset \alpha$: attribute α matches attribute constraint ϕ;
- $f \sqsubset^N_S n$: notification n matches filter f, where f is interpreted as a subscription filter;
- $a \sqsubset^N_A n$: notification n matches filter a, where a 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,

  $$f_1 \sqsubset^S_S f_2 \Leftrightarrow \forall n : f_2 \sqsubset^N_S n \Rightarrow f_1 \sqsubset^N_S n$$

  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:

  $$a_1 \sqsubset^A_A a_2 \Leftrightarrow \forall n : a_2 \sqsubset^N_A n \Rightarrow a_1 \sqsubset^N_A n$$

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

- $a \sqsubset^S_A f$: filter a covers filter f, where a is interpreted as an advertisement and f is interpreted as a subscription. Formally,

  $$a \sqsubset^S_A f \Leftrightarrow \exists n : a \sqsubset^N_A n \wedge f \sqsubset^N_S n$$

  which means that a defines a set of notifications that has a non-empty intersection with the set defined by f.

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

$$f_1 = f_2 \Leftrightarrow f_2 \sqsubset f_1 \wedge f_1 \sqsubset f_2$$

.

We now use the relations $\sqsubseteq_S^S$ and $\sqsubseteq_A^A$ to define the semantics of unsubscriptions and unadvertisements.

3.8 Unsubscriptions and Unadvertisements

Unsubscriptions and unadvertisements serve to cancel previous subscriptions and advertisements, respectively. Given a simple unsubscription **unsubscribe**(X, f), where X is the identity of an interested party and f is a filter, the event notification service cancels all simple subscriptions **subscribe**(X, g) submitted by the same interested party X with a subscription filter g covered by f (i.e., such that $f \sqsubseteq_S^S g$). This semantics is extended easily to patterns: An unsubscription for a pattern $P = f_1 \cdot f_2 \cdots f_k$ cancels all previous subscriptions $S = g_1 \cdot g_2 \cdots g_k$ such that $f_1 \sqsubseteq_S^S g_1 \wedge f_2 \sqsubseteq_S^S g_2 \wedge \ldots \wedge f_k \sqsubseteq_S^S g_k$. In an analogous way, unadvertisements cancel previous advertisements that are covered according to the relation $\sqsubseteq_A^A$.

Note that an unsubscription (unadvertisement) either cancels previous subscriptions (advertisements) or else has no effect. It cannot impose further constraints onto existing subscriptions. For example, subscribing with a filter [price > 100] and then unsubscribing with [price > 200] does not result in creation of a reduced subscription, [price > 100, price≤ 200]. Rather, the unsubscription simply has no effect, since it does not cover the subscription. Note also that all subscriptions covered by an unsubscription are cancelled by that unsubscription. Thus, when an interested party initially subscribes with a specific filter (say, [change > 10]), then subscribes with a more generic one (say, [change > 0]), and then finally unsubscribes with a filter that covers the more generic subscription (say, [change > 0]), the effect is to cancel all the previous subscriptions, not to revert to the more specific one [change > 10].

3.9 Timing issues

The semantics of SIENA depends on the order in which SIENA receives and processes requests (subscriptions, notifications, etc.). For instance, in the subscription-based semantics, a subscription s is effective after it is processed and until an unsubscription u that cancels s is processed.

In the most general case, a service request R, say a subscription, is generated at time R_g, received at time R_r and completely processed at time R_p (with $R_g \leq R_r \leq R_p$). SIENA guarantees the correct interpretation of R immediately after R_p. Notice that the *external delay* $R_g - R_r$ is caused by external communication mechanisms and is by no means controllable by SIENA. The *processing delay* $R_p - R_g$ is instead directly caused by computations and possibly by other communication delays internal to SIENA.

SIENA's semantics is that of a *best effort* service. This means that the implementation of SIENA must not introduce unnecessary delays in its processing, but it is not required to prevent race conditions induced by either the external delay or the processing delay. Clients of SIENA must be resilient to such race conditions; for instance, they must allow for the possibility of receiving a notification for a cancelled subscription.

SIENA associates a timestamp with each notification to indicate when it was published.[2] This allows the service to detect and account for the effects of latency on the matching of patterns, which means that within certain limits the actual order of notifications can be recognized.

4 Architectures: Server Topologies and Protocols

The previous section describes the protocol by which clients (i.e., objects of interest and interested parties) communicate with the servers that act as the clients' access points to the event notification service. As mentioned in Section 2, the servers themselves communicate in order to cooperatively distribute the selection

[2]With the advent of accurate network time protocols and the existence of the satellite-based Global Positioning System (GPS), it is reasonable to assume the existence of a global clock for creation of these timestamps, and it is hence reasonable for all but the most time-sensitive applications to rely on these timestamps.

and delivery tasks across a wide-area network. The servers must therefore be arranged into an interconnection topology and make use of a server/server communication protocol. Together, the topology and protocol define what we refer to as an *architecture* for the event notification service.

The architecture is assumed to be implemented on top of a lower-level network infrastructure. In particular, a topological connection between two servers does not necessarily imply a permanent or direct physical connection between those servers, such TCP/IP. Moreover, the server/server protocol might make use of any one of a number of network protocols, such as HTTP or SMTP, through standard encoding and/or tunneling techniques. All we assume at this point in the discussion is that a given server can communicate with some number of other specific servers by exchanging messages. This is the same assumption we make about the communication between clients and servers.

In this section we consider three basic architectures: hierarchical client/server, acyclic peer-to-peer, and general peer-to-peer. We also consider some hybrid architectures. Because it is not scalable, we ignore the degenerate case of a *centralized* architecture having a single server.

4.1 Hierarchical Client/Server Architecture

A natural way of connecting event servers is according to a hierarchical topology, as illustrated in Figure 5. In this topology, pairs of connected servers interact in an asymmetric client/server relationship. Hence, we use a directed graph to represent the topology of this architecture, and we refer to this architecture as a *hierarchical client/server* architecture (or simply a *hierarchical* architecture). A server can have any number of incoming connections from other "client" servers, but only one outgoing connection to its own "master" server. A server that has no "master" server of its own is referred to as a *root*.

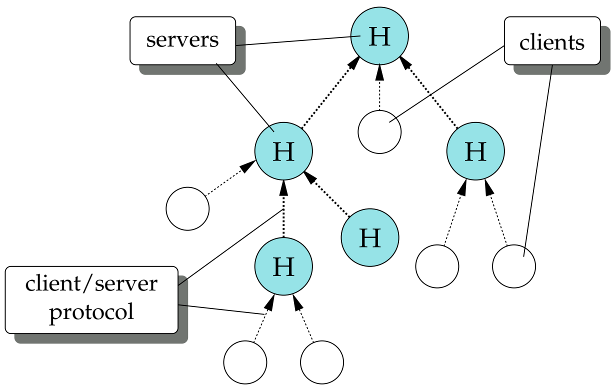

Figure 5: Hierarchical Client/Server Architecture.

The hierarchical architecture is a straightforward extension of a centralized architecture. It only requires that the basic central server be modified to propagate any information that it receives (i.e., subscriptions, etc.) on to its "master" server. In fact, the server/server protocol we use within the hierarchical architecture is exactly the same as the protocol described in Section 3.1 for communication between the servers and the external clients of the event notification service. Thus, in terms of communication, a server is not distinguished from objects of interest or interested parties. In practice, this means that a server will receive subscriptions, advertisements and notifications from its "client" servers, and will send only notifications back to those "client" servers.

As we demonstrate in Section 6.2.2, the main problem exhibited by the hierarchical architecture is the potential overloading of servers high in the hierarchy. Moreover, every server acts as a critical point of failure for the whole network. In fact, a failure in one server disconnects all the subnets reachable from its "master" server and all the "client" subnets from each other.

4.2 Acyclic Peer-to-Peer Architecture

In the *acyclic peer-to-peer* architecture, servers communicate with each other symmetrically as peers, adopting a protocol that allows a bi-directional flow of subscriptions, advertisements, and notifications. Hence we use an undirected graph to represent the topology of this architecture. (As always, the external clients of the service use the standard client/server protocol described in Section 3.1.) The configuration of the connections among servers in this architecture is restricted so that the topology forms an acyclic undirected graph. Figure 6 shows an acyclic peer-to-peer architecture of servers. The communication between servers is represented by thick undirected lines, while the communication between clients and servers is represented by dashed arrows.

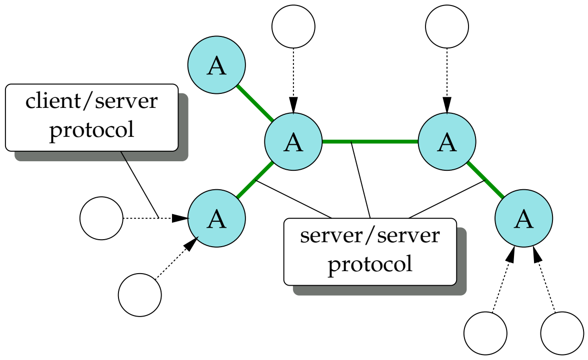

Figure 6: Acyclic Peer-to-Peer Server Architecture.

It is important that the procedures adopted to configure the connections among servers maintain the property of acyclicity, since routing algorithms might rely on the property to assume, for instance, that any two servers are connected with at most one path. However, ensuring this can be difficult and/or costly in a wide-area service in which administration is decentralized and autonomous.

As in the hierarchical architecture, the lack of redundancy in the topology constitutes a limitation in assuring connectivity, since a failure in one server S isolates all the subnets reachable from those servers directly connected to S.

4.3 General Peer-to-Peer Architecture

Removing the constraint of acyclicity from the acyclic peer-to-peer architecture, we obtain the *general peer-to-peer* architecture. Like the acyclic peer-to-peer architecture, this architecture allows bi-directional communication between two servers, but the topology can form a general undirected graph, possibly having multiple paths between servers. An example is shown in Figure 7.

The advantage of the general peer-to-peer architecture over the previous two architectures is that it requires less coordination and offers more flexibility in the configuration of connections among servers. Moreover, 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. Typically, messages will carry a "time-to-live" counter, and routes will be established according to minimal spanning trees. Consequently, the server/server protocol adopted in the general peer-to-peer architecture must accommodate this extra information.

4.4 Hybrid Architectures

A wide-area, large-scale, decentralized service such as SIENA poses different requirements at different levels of administration. In other words, we must account for intermediate levels between the local area and the

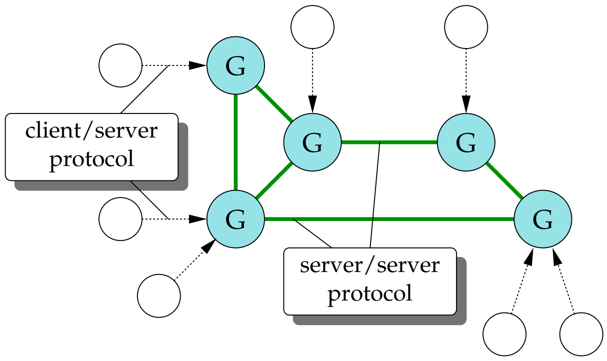

Figure 7: General Peer-to-Peer Server Architecture.

wide area. We can potentially take advantage of these intermediate levels to gain some efficiencies by considering the use of different architectures at different levels of network granularity.

For example, in the case of a multi-national corporation, it might be reasonable to assume a high degree of control and coordination in the administration of the cluster of subnets of the corporation's intranet. The administrators of this intranet might very well be able to design and manage the whole network of event servers deployed on their subnets, and thus it might be a good idea to adopt a hierarchical architecture within the intranet. Of course, the intranet would connect to other networks outside of the influence of the administrators. Thus, what could arise is a general peer-to-peer architecture at the global level, serving to interconnect different corporate intranets, each having a hierarchical architecture. This is illustrated in Figure 8.

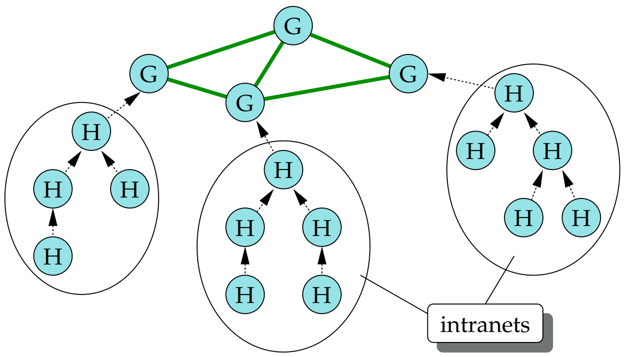

Figure 8: Hierarchical/General Hybrid Server Architecture.

In other cases, we might want to invert the structure, as illustrated in Figure 9. For example, suppose that some clusters of subnets carry a high degree of event-service message traffic, and for some specific applications or perhaps for security reasons, only a small fraction of that traffic is visible outside the cluster. In this case, for efficiency reasons a general peer-to-peer architecture might be preferable within the clusters, while the high-level architecture could be acyclic peer-to-peer. For every cluster, there would be a gateway server that should be able to filter the messages used for the protocol inside the cluster, and adapt them to the protocol used between clusters. For example, if a protocol is used locally within a cluster to discover minimal spanning trees, then the messages associated with that protocol should not be propagated outside the cluster.

Hybrid architectures such as these are somewhat more complicated than the three basic architectures.

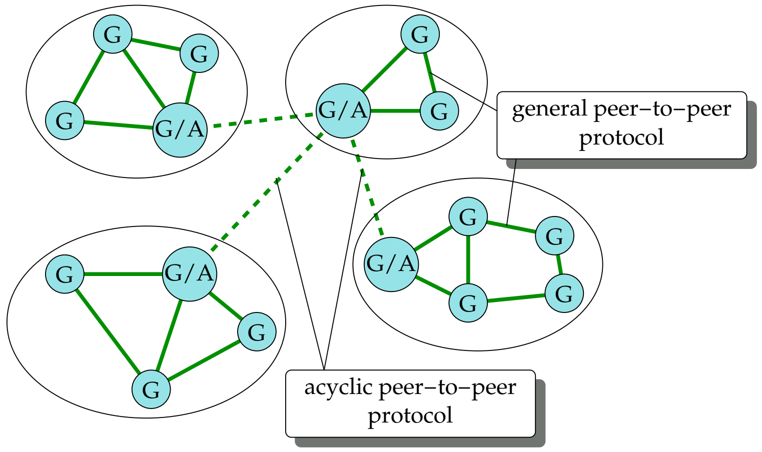

Figure 9: General/Acyclic Hybrid Server Architecture.

Nevertheless, they offer the opportunity to tailor the server/server topologies and protocols in such a way that localities can be exploited.

5 Routing Algorithms and Processing Strategies

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

5.1 Routing Strategies in SIENA

To devise more efficient routing algorithms, we employ principles found in IP multicast routing protocols [14]. Similar to these protocols, the main idea behind the routing strategy of SIENA is to send a notification only toward 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. This principle is illustrated in Figure 10.

upstream evaluation: Filters are applied and patterns are assembled upstream, that is, as close as possible to the sources of (patterns of) notifications. This principle is illustrated in Figure 11.

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:

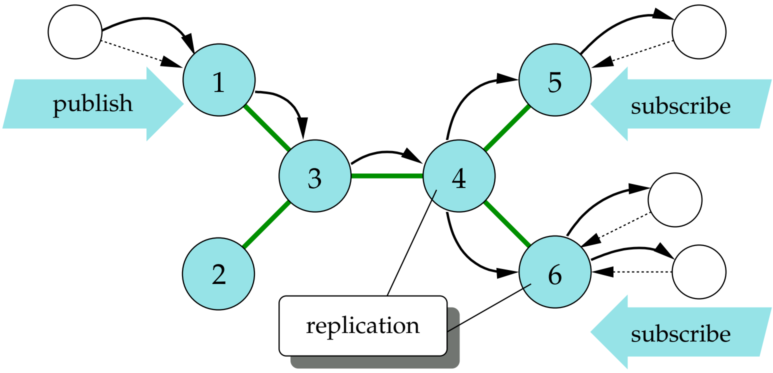

Figure 10: Downstream Replication of Notifications.

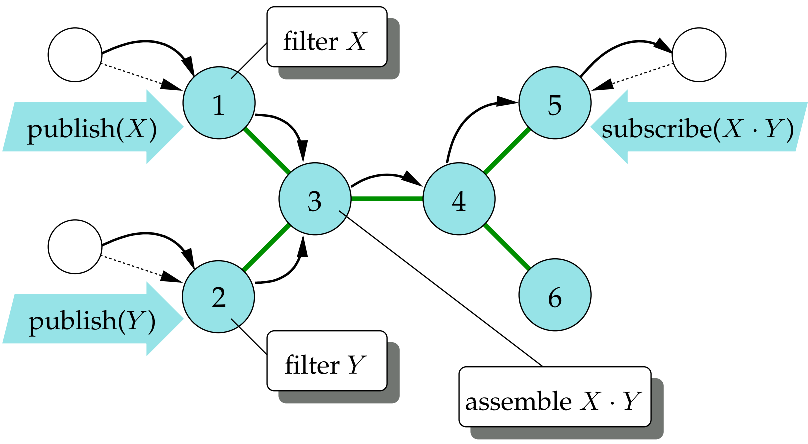

Figure 11: Upstream Evaluation of Filters and Patterns.

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 that are relevant to that subscription. Thus, advertisements set the paths for subscriptions, which in turn set the paths for notifications. Every advertisement is propagated throughout the network, thereby forming a tree that reaches every server. When a server receives a subscription, it propagates the subscription in reverse, along the paths to all advertisers that submitted relevant advertisements, thereby *activating* those paths. Notifications are then forwarded only through the activated paths.

In the process of forwarding subscriptions, SIENA exploits commonalities among the subscriptions. In particular, SIENA prunes the propagation trees by propagating along only those paths that have not been covered by previous requests. The derived covering relation $\sqsubset_S^S$ is used to determine whether a new subscription is covered by a previous one that has already been forwarded. Advertisements are treated similarly using the relation $\sqsubset_A^A$. And although not discussed in detail here, unsubscriptions and unadvertisements are handled in a similar way as well.

Subscription forwarding algorithms realize a *subscription-based* semantics, while advertisement forwarding algorithms realize an *advertisement-based* semantics. As we show in Section 5.4, advertisement forwarding algorithms are needed in order to implement the upstream evaluation principle for event patterns.

5.2 Putting Algorithms and Topologies Together

In this section we describe in detail how subscription forwarding and advertisement forwarding algorithms are implemented over the hierarchical and peer-to-peer architectures. In particular, we describe the principal data structures maintained by servers and the main algorithms that process the various requests coming from clients or other servers. Here we consider only simple subscriptions; Section 5.4 deals with patterns.

5.2.1 The Filters *Poset*

In order to keep track of previous requests, their relationships, where they came from, and where they have been forwarded, event servers maintain a data structure that is common to the different algorithms and topologies. This data structure represents a partially ordered set (*poset*) of filters. The partial order is defined by the covering relations $\sqsubset_S^S$ for subscription filters, and $\sqsubset_A^A$ for advertisement filters. We denote with P_S a poset defined by $\sqsubset_S^S$, and denote with P_A a poset defined by $\sqsubset_A^A$. Figure 12 shows an example of a poset of subscriptions.

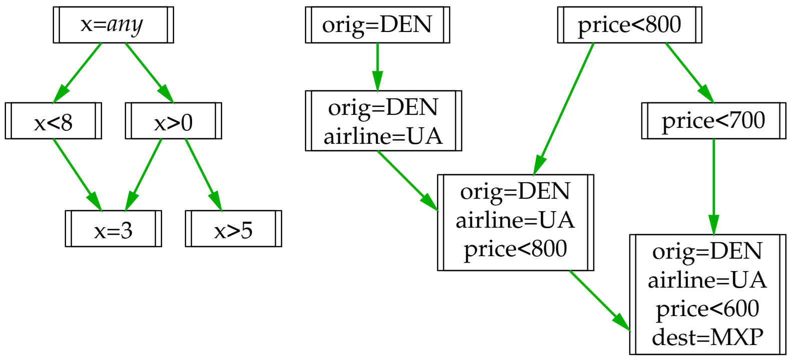

Figure 12: Example of a Poset of Simple Subscriptions. Arrows represent the *immediate* relation $\sqsubset_S^S$.

Note that $\sqsubset^S_S$ and $\sqsubset^A_A$ are transitive relations, while the diagram and its representation in memory store *immediate* relationships only. In a poset P_S, ordered according to $\sqsubset^S_S$, a filter f_1 is an *immediate successor* of another filter f_2 and f_2 is an *immediate predecessor* of f_1 if and only if $f_2 \sqsubset^S_S f_1$ and there is no other filter f_3 in P_S such that $f_2 \sqsubset^S_S f_3 \sqsubset^S_S f_1$. Filters that have no immediate predecessors are called *root* filters.

When inserting a new filter f into a poset, three different cases apply that are of special interest for the forwarding algorithms:

- f is added as a *root* filter;
- f exists already in the poset; or
- f is inserted somewhere in the poset with a non-empty set of predecessors.

As we detail below, only root filters produce network traffic, due to the propagation of subscriptions (or advertisements). Thus the "shape" of a subscription (or advertisement) poset roughly indicates the effectiveness of our optimization strategies. In particular, a poset that extends "vertically" indicates that subscriptions are very much interdependent and that there are just a few subscriptions summarizing all the other ones. Conversely, a poset that extends "horizontally" indicates that there are few similarities among subscriptions and that there are thus few opportunities to reduce network traffic.

5.2.2 Hierarchical Client/Server Architecture

A hierarchical server stores its subscriptions in a poset P_S. Each subscription s in P_S has an associated set *subscribers*(s) containing the identities of the subscribers of that filter. Every server also has a variable *master*, possibly null, containing the identity of its "master" server.

Subscriptions Upon receiving a simple subscription **subscribe**(X, f), a server E walks through its subscription poset P_S, starting from each root subscription, looking for a filter f' that covers the new filter f and that contains X in its subscribers set: $f' \sqsubset^S_S f \land X \in$ *subscribers*(f'). If the server finds such a subscription f' in P_S, it simply terminates the search without any effect. This happens when the same interested party (X) has already subscribed for a more generic filter (f').

In case the server does not find such a subscription, the search process terminates producing two possibly empty sets $\overline{f}$ and $\underline{f}$, representing the immediate predecessors and the immediate successors of f, respectively. If $\overline{f} = \underline{f} = \{f\}$, that is, if filter f already exists in P_S, then the server simply inserts X in *subscribers*(f). Otherwise, f is inserted in P_S between $\overline{f}$ and $\underline{f}$, and X is inserted in its subscribers set.

Only if $\overline{f} = \emptyset$, that is, only if f is inserted as a root subscription, does the server then forward the same subscription to its master server. In particular, if *master* is not null, the server (E) sends a subscription **subscribe**(E, f) to *master*.

If $\underline{f} \neq \emptyset$, the server removes X from the sets of subscribers of all the subscriptions covered by f. This is done by recursively walking breadth first through the poset P_S starting from the subscriptions in $\underline{f}$. The recursion is stopped whenever X is found in a subscription (and removed). Note that, in this process, some subscriptions might be left with no associated interested parties; such subscriptions are removed from P_S.

We illustrate the processing of subscriptions in the hierarchical architecture with the scenario depicted in Figures 13 through 15. Figure 13 depicts a hierarchical server (1) that has two clients (a and b) and a master server (2). The server receives and processes a subscription [airline=UA] from client a. The right side of the figure shows the subscription poset P_S of server 1. The new subscription is inserted as a root subscription, so server 1 forwards it to its master server (2).

Figure 14 continues the example of Figure 13. Here server 1 receives another subscription [airline=UA, dest=DEN] from client b. Since this new subscription is already covered by the previously forwarded subscription (it is not made a root subscription in P_S), server 1 does not forward it to its master.

In Figure 15 server 1 processes another subscription [airline=*any*] from client a. This is a root subscription and so it is forwarded to server 2. In this case, server 1 eliminates a from the subscribers of all the subscriptions covered by the new one. In particular, it removes a from the first subscription [airline=UA]; because there are no other subscribers for that subscription, the subscription itself is also removed.

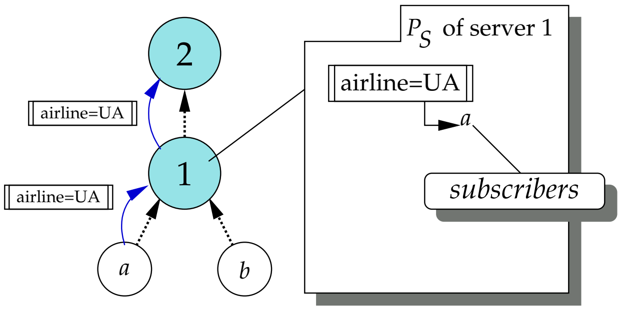

Figure 13: Subscription Scenario in the Hierarchical Architecture (step 1).

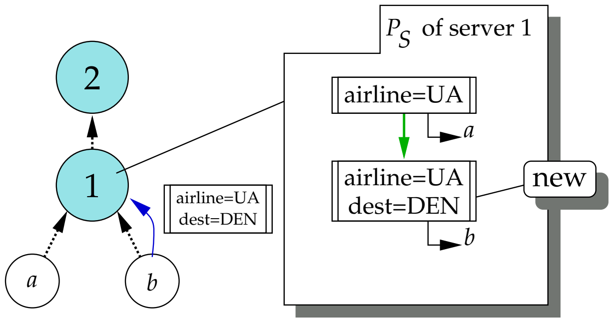

Figure 14: Subscription Scenario in the Hierarchical Architecture (step 2).

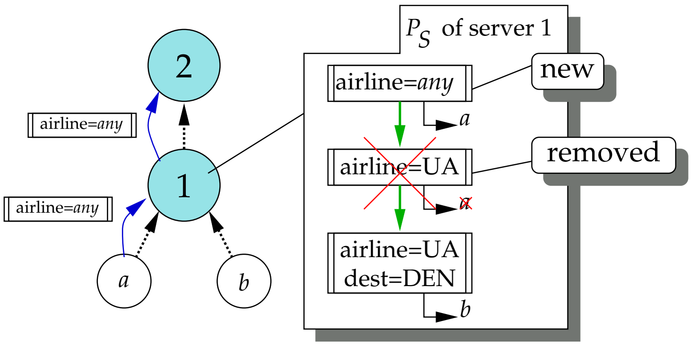

Figure 15: Subscription Scenario in the Hierarchical Architecture (step 3).

Notifications When a server receives a notification n, it walks through its subscriptions poset P_S breadth first looking for all the subscriptions matching n. In particular, the server initializes a queue Q with its root subscriptions. Then, the server iterates through each element s in Q. If $s \sqsubseteq_S^N n$, the server appends to Q all the immediate successors of s that have not yet been visited. Otherwise, if $s \not\sqsubseteq_S^N n$, the server removes s from the queue.

When this process terminates, Q contains all the subscriptions that cover n. The server then sends a copy of n to each subscriber of the subscriptions in Q. Independently of the matching of subscriptions, if the server has a master server and the master server was not the sender of n, then the server also sends a copy of n to its master server.

Unsubscription Unsubscriptions cancel previous subscriptions, but they are not exactly the inverse of subscriptions. They are slightly more complex to handle and sometimes more expensive in terms of communication. One reason is that a single unsubscription might cancel more than one previous subscription. The other reason is that an unsubscription might cancel one or more root subscriptions, which in turn might uncover other more specific subscriptions (which in turn become new root subscriptions). In this case, the server must forward the unsubscription to its master server, but it must also forward the new root subscriptions as well.

More specifically, when a server receives an unsubscription **unsubscribe**(X, f), it removes X from the subscribers set of all the subscriptions in P_S that are covered by f. The algorithm used by the server in this case is a simple variation of the algorithm that computes the set of matching subscriptions for a notification. The only difference is that the relation $\sqsubseteq_S^S$ is used to fill the queue instead of $\sqsubseteq_S^N$.

As a consequence of removing X, some subscriptions might remain with an empty set of subscribers. Let S_X be the set of such subscriptions and let S_X^r ($S_X^r \subset S_X$) be the set of those that are also root subscriptions in P_S. The server computes $\underline{S_X^r}$ as the union of all the immediate successors of each subscription in S_X^r. With all this, the server:

1. removes all the subscriptions in S_X from P_S;
2. forwards the unsubscription for f to its master server; and
3. sends all the subscriptions in $\underline{S_X^r}$ to its master server.

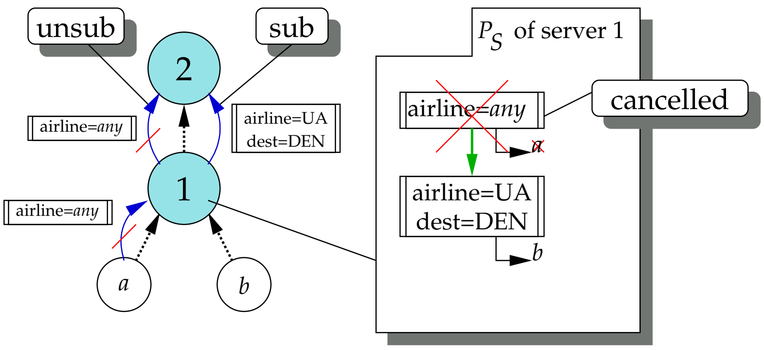

Figure 16: Unsubscription in the Hierarchical Architecture.

Figure 16 continues from the example of Figure 15. Server 1 receives an unsubscription for [airline=*any*] from client a. As a consequence, it removes a from the subscribers of subscription [airline=*any*], which is the only subscription from a covered by the unsubscription (in this case, the two filters coincide). The subscription contains no more subscribers, and so the server removes it. But since it was also a root subscription, the server forwards the unsubscription to its master along with the new root subscription, [airline=UA, dest=DEN].

Advertisements The advertisement forwarding technique does not apply to the hierarchical architecture. Although it would be possible to propagate advertisements from a server to its master, this would be useless, since the master server would never respond by sending back subscriptions. In fact, a hierarchical server considers all its clients as "normal" clients (i.e., outside the event notification service), so it would not forward subscriptions to them. In practice, advertisements and unadvertisements are silently dropped.

5.2.3 Peer-to-Peer Architectures with Subscription Forwarding

In peer-to-peer architectures, each server maintains a set *neighbors* containing the identifiers of the peer servers to which the server is connected. A peer-to-peer server also maintains its subscriptions in a poset P_S that is an extension of the subscription poset of a hierarchical server. As in the hierarchical server, a peer-to-peer server associates a set *subscribers*(s) with each subscription s, and it associates an additional set with s called *forwards*(s), which contains the subset of neighbors to which s has been forwarded.

General vs. Acyclic Architectures A subscription or notification is propagated from its origin to its destination following a minimal spanning tree. In an acyclic peer-to-peer architecture the path that connects any two servers (if it exists) is unique, and any such spanning tree coincides with the whole network of servers. Thus, when propagating a message m, say a subscription, a server simply sends it to all of its neighbors excluding the sender of m. Any server that propagates m is considered to be a sender of m, but the *origin* of a message is the (unique) event notification service access point to which the message is originally posted.

In a general peer-to-peer architecture, two servers might be connected by two or more different paths. So when a server receives a message that must be forwarded throughout the network of servers, the first server must make sure to forward it only through the links of the minimal spanning tree rooted in the origin of that message. This is similar to the well-known problem of broadcasting information over a packet-switched network. Several distributed algorithms have been designed and implemented to solve this problem [12], so we do not discuss this aspect further in the context of SIENA.[3] In order to simplify the description of the algorithms we focus only on acyclic peer-to-peer architectures; algorithms for the general peer-to-peer architectures can be found elsewhere [5].

Peer Connection Setup A server E_1 connects to a server E_2 by sending a **peer_connect**(E_1) request to E_2. E_2 can either accept or refuse the connection. In case E_2 accepts E_1 as a peer, E_2 sends a confirmation message back to E_1 so that both servers add each other's address to their neighbors set. Then the accepting server E_2 forwards every root subscription in its subscriptions poset P_S to the requesting server E_1, adding E_1 to the corresponding forwards set. Servers can also be dynamically disconnected with a **peer_disconnect**(E_1) request. When a server E_2 receives a **peer_disconnect**(E_1), it removes E_1 from its neighbors set, unsubscribes E_1 for all its root subscriptions, and finally removes E_1 from all its forwards sets.

Subscriptions The algorithm by which a peer-to-peer server processes subscriptions is an extension of the algorithm of the hierarchical server. When a server receives a subscription **subscribe**(X, f), it searches its subscriptions poset P_S for either

1. a subscription f' that covers f and has X among its subscribers: $f' \sqsubset_S^S f \wedge X \in$ *subscribers*(f'). In this case, the search terminates with no effect; or

2. a subscription f' that is equal to f and does not have X among its subscribers: $f' \sqsubset_S^S f \wedge f \sqsubset_S^S f'$. Here the server adds X to *subscribers*(f'); or

3. two possibly empty sets $\overline{f}$ and $\underline{f}$, representing the immediate predecessors and the immediate successors of f respectively. Here the server inserts f as a new subscription between $\overline{f}$ and $\underline{f}$, and adds X to *subscribers*(f).

[3] Assuming that the communication layer underlying the event notification service is a packet-switched network (like the Internet), and that the topology of servers is configured to match the topology of the communication layer, if nothing else it would be possible to use the routing information already maintained by the underlying network to control the forwarding process.

In cases 2 and 3, the server also removes X from all the subscriptions in P_S that are covered by f, and then removes from P_S those subscriptions that have no other subscribers.

This procedure differs from the corresponding procedure of the hierarchical server in how the peer-to-peer server forwards the subscription to its neighbors. Formally, given a subscription f in P_S, let *forwards*(f) be defined as follows:

$$forwards(f) = neighbors - NST(f) - \bigcup_{f' \in P_S \wedge f' \sqsubset_S^S f} forwards(f') \quad (1)$$

In other words, f is forwarded to all neighbors of the server except those not downstream from the server along any spanning tree rooted at an original subscriber of f (the second term in the formula), and those to which subscriptions f' covering f have been forwarded already by this server (the last term in the formula).

The second term in the formula (whose functor stands for "Not on any Spanning Tree") accounts for the fact that there may be multiple paths connecting a subscriber to potential publishers, and that therefore the propagation of a subscription f must follow only the computed spanning trees rooted at the original subscribers of f. Viewing a spanning tree rooted at f as a directed graph, we may refer to paths traveling away from f as going "downstream" with the edges, and those traveling toward f as going "upstream" against the edges. In practice, the propagation process excludes those neighbors that are *not* downstream from the server of interest along any spanning tree rooted at a subscriber of f. $NST(f)$ is trivially computed for the topology of the acyclic architecture, since every spanning tree in the topology coincides with the whole topology itself. For the topology of the general architecture its computation is more complicated; however, the necessary techniques, such as link-state or distance-vector routing algorithms, are well-known and widely deployed. An alternative approach to propagating subscriptions is to use Dalal and Metcalfe's broadcasting algorithm [12].

The last term in the formula represents an important optimization that the server makes in the situation where more generic subscriptions have been propagated already to some neighbors.

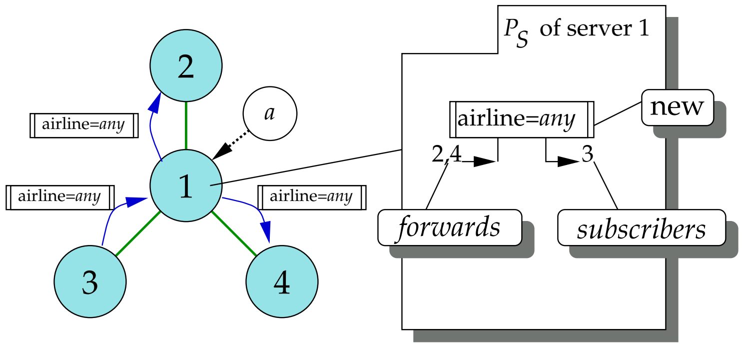

Figure 17: Subscription Scenario in the Acyclic Peer-to-Peer Architecture (step 1).

We illustrate the processing of subscriptions in the acyclic peer-to-peer architecture with the scenario depicted in Figures 17 through 19. Figure 17 shows a fragment of a peer-to-peer event notification service. In this example, server 1 is connected to servers 2, 3, and 4. Server 1 also has a local client a. Server 3 sends a subscription [airline=*any*] to server 1. The poset shown on the right side of the figure represents the subscription poset P_S of server 1. As shown in the figure, the new subscription is inserted as a root subscription in P_S and then forwarded to servers 2 and 4 but not to server 3, which is in the NST set of the subscription. In this figure and the following ones, for each subscription in P_S, subscribers are denoted with an outgoing arrow from the subscription, while forwards are denoted with an incoming arrow. Intuitively, arrows indicate the direction of notifications.

Figure 18 shows the effect of a second subscription [airline=UA, orig=DEN] sent to server 1 by server 2. This subscription is inserted in P_S as an immediate successor of the previous (root) subscription and is

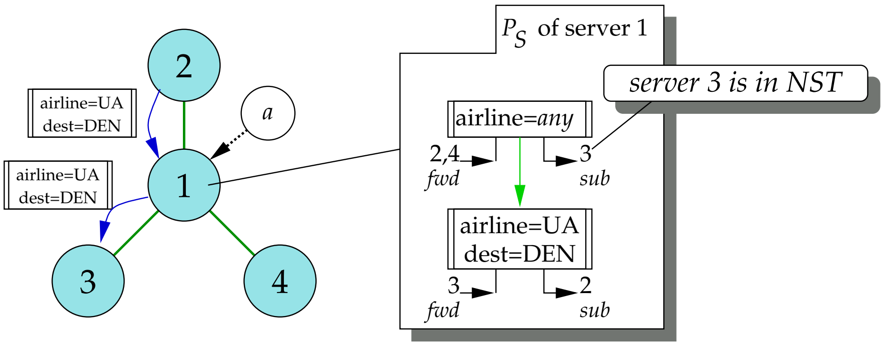

Figure 18: Subscription Scenario in the Acyclic Peer-to-Peer Architecture (step 2).

forwarded to server 3, which is the only neighbor that is not in the forwards set for the covering subscription [airline=*any*].

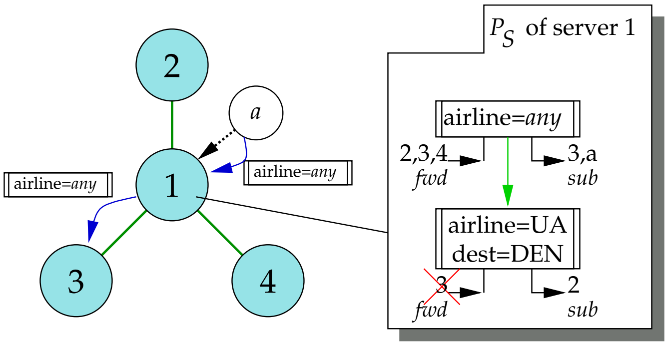

Figure 19: Subscription Scenario in the Acyclic Peer-to-Peer Architecture (step 3).

In Figure 19 client a subscribes for [airline=*any*]. In this case, the subscription is found in P_S and a is simply added to its subscribers set. Because the NST set for that subscription is now empty, the subscription is then forwarded to server 3. Every time the server forwards a subscription f to a neighbor server E_2, it adds E_2 to the forwards set of f and consequently removes E_2 from the forwards sets of all the subscriptions covered by f. In the example, the server removes 3 from the forwards set of subscription [airline=UA, orig=DEN].

Unsubscriptions An unsubscription has the effect of removing a subscriber from a number of subscriptions in P_S. More specifically, when a server E receives an unsubscription **unsubscribe**(X, f), it removes X from the subscribers set of every subscription covered by f.

As a consequence of these cancellations, some subscriptions might remain with an empty subscribers set; such subscriptions are removed from P_S. The removal of X from some subscriptions might also affect the NST set of those subscriptions. In particular, removing a subscriber for a subscription means removing its distribution spanning tree, which in turn might add some neighbor servers to NST for those paths that are not on the spanning tree of any other subscriber (see equation 1 on page 21). In order to reduce the forwards set of those subscriptions according to equation 1, the server forwards the corresponding unsubscriptions to every neighbor server added to NST.

The reduced forwards sets of some subscriptions might affect the forwards sets of other covered subscriptions. This effect is produced by the last term of equation 1 for the covered subscriptions. Intuitively, this means that after unsubscribing for some more generic subscriptions it might be necessary to (re)forward some more specific subscriptions whose propagation was blocked by the existence of the more generic subscriptions.

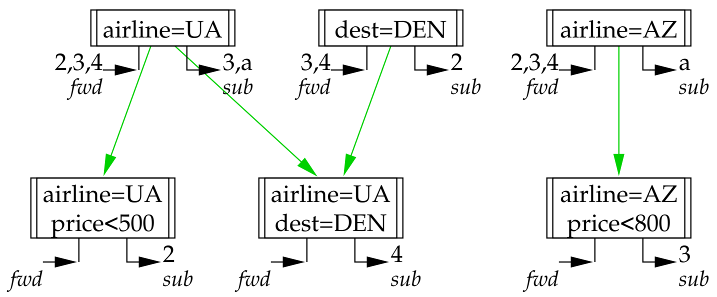

Figure 20: Unsubscription Scenario in the Acyclic Peer-to-Peer Architecture (initial state).

We illustrate the processing of unsubscriptions in the acyclic peer-to-peer architecture with the scenario depicted in Figures 20 through 22. Figure 20 depicts the subscriptions poset of server 1 from Figure 19 after it has received some subscriptions from local clients and neighbor servers. This is the initial state of server 1 just before it receives an unsubscription filter [airline=*any*] from client a.

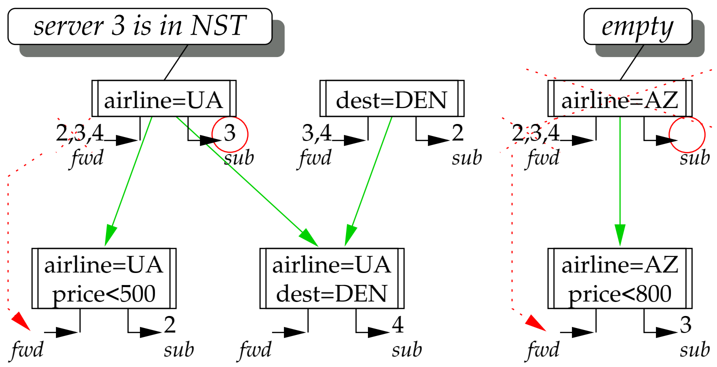

Figure 21: Unsubscription Scenario in the Acyclic Peer-to-Peer Architecture (step 1).

As a first step in processing the unsubscription from client a, server 1 removes the subscriber (i.e., a) from all the subscriptions covered by the unsubscription filter [airline=*any*]. Figure 21 shows the subscriptions poset P_S in this state. Two (root) subscriptions are affected: subscription [airline=UA] changes its NST set (which is initially empty) to include server 3, while subscription [airline=AZ] remains with an empty subscribers set. As a consequence, server 1 forwards the first unsubscription [airline=UA] to the neighbor server added to the NST set (i.e., 3) and forwards the second unsubscription [airline=AZ] to all the previous forwards 2, 3, and 4.

Eventually, server 1 processes the immediate successors of the cancelled subscriptions since their forwards might have changed as a consequence of the previous unsubscriptions. Figure 22 shows the state of the subscription poset at this time. The subscription [airline=UA, price<500] must be forwarded to server 3 because its (only) predecessor has not been forwarded to server 3 (see equation 1). Subscription [airline=UA, dest=DEN] does not need to be propagated because all the neighbor servers have received either one of its predecessors. Subscription [airline=AZ, price<800] has now become a root subscription and thus must be forwarded to every neighbor server except those in its NST (i.e., server 3).

Notifications The algorithm for peer-to-peer architectures processes notifications exactly like the one that operates on the hierarchical architecture. So, a subscription n is forwarded to every subscriber of s for every s that covers n.

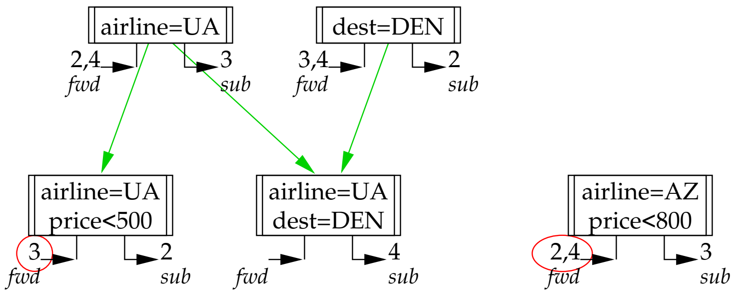

Figure 22: Unsubscription Scenario in the Acyclic Peer-to-Peer Architecture (step 2).

5.2.4 Advertisement Forwarding

With the subscription forwarding algorithm presented in the previous sections, we have described almost everything needed to implement an advertisement forwarding algorithm. In fact, we can exploit the duality between subscriptions and advertisements to transpose the subscription forwarding algorithm to advertisement forwarding. To some extent, if we read the description of the subscription forwarding algorithm, replacing the terms regarding subscriptions with the corresponding terms regarding advertisements, and replacing the terms regarding notifications with the corresponding terms regarding subscriptions, we obtain an almost exact description of the advertisement forwarding algorithm.

The main difference with respect to the subscription forwarding structure is that there are actually two interacting computations; one realizes the forwarding of advertisements while the other realizes the forwarding of subscriptions. Both computations have similar data structures and similar algorithms, equivalent to the ones described above. In particular, the server has a poset of advertisements P_A, ordered according to the relation $\sqsubset_A^A$, as well as a poset of subscriptions P_S. In P_A, each advertisement a has an associated set of identities *advertisers*(a) and another set of identities *forwards*(a).

These two computations interact in the sense that advertisement forwarding constrains subscription forwarding. For instance, in maintaining P_S, when processing a subscription s, the server does not use the global set *neighbors*, but instead uses a subset *neighbors*$_s \subseteq$ *neighbors* that is specific to s. *neighbors*$_s$ is defined as the set of advertisers listed in P_A for all the advertisements covering s. Formally:

$$\mathit{neighbors}_s = \bigcup_{a \in T_A : a \sqsubset_A^S s} \mathit{advertisers}(a) \cap \mathit{neighbors}$$

Note that one effect of this constraint is that new advertisements and unadvertisements are viewed by the subscription forwarding computation as new peer connections or dropped peer connections, respectively. Thus, if the server receives a new advertisement that covers a set of subscriptions $s_1, s_2, \ldots, s_k$, then the server reacts by forwarding $s_1, s_2, \ldots, s_k$ immediately to the sender of the advertisement.

5.3 Other Optimization Strategies

In addition to the main principles discussed in the previous sections, SIENA event servers may perform other types of optimizations. These techniques are variants of the algorithms discussed so far. Note that we have simulated these optimizations in our simulation studies (described in Section 6.2), but we have not yet implemented them in our prototype implementation (described in Section 6.3).

5.3.1 Batching and Merging Subscriptions and Advertisements

There are cases in which a server must forward a whole set of subscriptions or advertisements to another server. For example, in the subscription forwarding algorithm, when a server accepts a connection request from a peer server, it immediately forwards all its root subscriptions to that peer server.

In these cases, the forwarding servers might try to merge two or more subscriptions into a more generic one. For example, suppose a server E_1 receives subscriptions from its local clients—one interested party subscribes for [alarm=1], another one subscribes for [alarm>2], and another one subscribes for [alarm=2]. Now, if a peer server E_2 sends a connection request to E_1, then E_1 can send one single subscription [alarm>0] back to E_2 together with the reply accepting the connection.

This mechanism can also be implemented among servers that exchange a very high number of subscriptions and unsubscriptions by batching requests. This means that instead of forwarding requests immediately as they are received, servers can implement a deferred forwarding by buffering requests and sending batches of requests periodically. When flushing the buffer of requests to construct the batch, the server can apply the merge reductions.

5.3.2 Space vs. Processing vs. Communication Tradeoffs

All the algorithms we present are geared toward the optimization of network resources. However, there are tradeoffs between the need for reducing communications, and the processing and memory usage on each event server.

In some cases, the data structures on servers can be compacted, thus saving space as well as processing power for maintaining and searching those data structures. Compaction can be achieved by merging subscriptions in subscription posets much the same way they can be batched and merged in the transmissions to other servers. In particular, it might be preferable to compact the subscription posets (or advertisement posets) a little more than what is induced exactly by the covering relations. This saves space and processing power, at the expense of forwarding or receiving unnecessary messages. For example, a server E can merge two subscriptions s_1 and s_2, sent by the same peer server, into another subscription $\overline{s}$ that defines a set of subscriptions that is larger than the union of the notifications defined by s_1 and s_2 alone. For example, having s_1 = [x>200] and s_2 = [x<100], the server could store $\overline{s}$ = [x=*any*]. This operation might cause the server to forward unnecessary notifications, namely those that match $\overline{s}$ but match neither s_1 nor s_2 (e.g., [x=150]). The advantage is that E now stores only the more generic subscription $\overline{s}$ as opposed to the two subscriptions s_1 and s_2. This compaction can be done only with subscriptions forwarded by peer servers since spurious notifications must go through additional filtering and eventually must be dropped before they reach an interested party.

5.3.3 Evaluation of the Covering Relations

The evaluation of the covering relations is clearly a crucial step in every algorithm presented here. For example, when a server receives a new notification n it looks for all the subscriptions s that cover n. In general, this implies computing the relation $s \sqsubseteq_S^N n$ for each subscription s stored by the server. However, the server can arrange its subscriptions in a structure that makes this evaluation more efficient. Aguilera et al. describe one such method [1].

The subscription poset described in Section 5.2.1 can also be used to optimize the evaluation of $\sqsubseteq_S^N$. In fact, because of the definition of $\sqsubseteq_S^S$ and its relation to $\sqsubseteq_S^N$, elements in a poset P_S are arranged in such a way that for every element s of P_S, all its immediate successors define subsets of the set of notifications defined by s. Thus, a notification n that is not covered by s is not covered by any other subscription s' covered by s (i.e., $s \not\sqsubseteq_S^N n \wedge s \sqsubseteq_S^S s' \Rightarrow s' \not\sqsubseteq_S^N n$). This allows the server to skip the evaluation of the subset of P_S covered by s.

Also, the definition of $\sqsubseteq_S^N$ requires a notification to contain every attribute defined by the subscription. Thus, the evaluation of $\sqsubseteq_S^N$ can be accelerated by sorting the attributes of notifications and subscriptions in alphabetic order by their name. Note that the sort need be performed only once when notifications, subscriptions and advertisements enter the event notification service for the first time (i.e., at an access point). Then all the subsequent forwards in server/server communications can preserve the order.

5.4 Matching Patterns

So far we have seen how simple subscriptions and simple notifications are handled by event servers. A major additional functionality provided by SIENA is the matching of patterns of notifications. This functionality is implemented with distributed monitoring following the upstream evaluation principle set forth in Section 5.1.

To match patterns, servers assemble sequences of notifications from smaller subsequences or from single notifications. Thanks to advertisements, every server knows which notifications and which subpatterns may be sent from each of its neighbors, which is why this technique requires an advertisement-based semantics. In addition to the notifications and patterns available from its neighbors, a server might further use patterns from previous subscriptions that the server itself is already set up to recognize.

We use the term *pattern factoring* to refer to the process by which the server breaks a compound subscription into smaller compound and simple subscriptions. After a subscription has been factored into its elementary components, the server attempts to group those factors into compound subscriptions to forward to some of its neighbors. This process is called *pattern delegation*.

5.4.1 Available Patterns Table

Every server maintains a table T_P of available patterns. This table is simply the advertisements poset P_A that, in addition to the usual advertisements, contains also those patterns that the server has already processed. Each pattern p in T_P has an associated set of identities *providers*(p) that contains all the peer servers from which p is available. Table 5 shows an example of a table of available patterns. The table

	pattern	*providers*
a_1	*string* *alarm* = *"failed-login"* *integer* *attempts* > *0*	3
a_2	*string* *file* *any* *string* *operation* = *"file-change"*	2,3

Table 5: Example of a Table of Available Patterns.

says that notifications matching filter a_1—notifications that signal a failed login with an integer attribute named "attempts"—are available from server 2, and that notifications matching filter a_2—file modification notifications—are available from servers 2 and 3.

5.4.2 Pattern Factoring

Let us suppose a server E receives a compound subscription **subscribe**(X, s), where $s = f_1 \cdot f_2 \cdot \ldots \cdot f_k$. Now, the server scans s trying to match each f_i with a pattern p_i, or trying to match a sequence of filters $f_i \cdot f_{i+1} \cdot \ldots \cdot f_{i+k_i}$ with a single compound pattern $p_{i\ldots i+k_i}$ using patterns p that are contained in T_P.

For example, assuming the table of available patterns shown in Table 5, suppose server 1 receives a subscription $s = f \cdot g \cdot h$ for a sequence of two "failed login" alarms with one and two attempts respectively (f = [alarm=failed-login, attempts=1], and g = [alarm=failed-login, attempts=2]), followed by a file modification event on file "/etc/passwd" (h = [file=/etc/passwd, operation=file-change]). In response to s, the server factors s, matching the three filters of s with the sequence of available patterns $a_1 \cdot a_1 \cdot a_2$. Table 6 shows the subscription and the factoring computed by the server. Because the only operator in SIENA for combining subpatterns is the sequence operator, the output of the factoring process is always a sequence.

5.4.3 Pattern Delegation

Once a compound subscription is divided into available parts, the server must (1) send out the necessary subscriptions to collect the required subpatterns and (2) set up a *monitor* that will receive all the notifications matching the subpatterns and will observe and distribute the occurrence of the whole pattern. In deciding which subscriptions to send out, the server tries to reassemble the elementary factors in groups that can be delegated to other servers, thereby implementing the upstream evaluation principle. The selection

requested	*available*	
string *alarm* = *"failed-login"* *integer attempts* = *1*	*string* *alarm* = *"failed-login"* *integer attempts* > *0*	(a_1)
string *alarm* = *"failed-login"* *integer attempts* = *2*	*string* *alarm* = *"failed-login"* *integer attempts* > *0*	(a_1)
string *file* = *"/etc/passwd"* *string operation* = *"file-change"*	*string* *file* *any* *string operation* = *"file-change"*	(a_2)

Table 6: Example of a Factored Compound Subscription.

of subpatterns that are eligible for delegation follows some intuitive criteria. For example, only contiguous subpatterns available from a single source can be grouped and delegated to that source. A complete discussion of these criteria is presented elsewhere [5].

In the example of Table 6, server 1 would group the first two filters $a_1 \cdot a_1$ and delegate the subpattern defined by the corresponding two subscriptions $(f \cdot g)$ to server 2. Thus, it would send a subscription **subscribe**(E, s_1) with pattern

$$s_1 = \boxed{\begin{array}{l} \textit{string} \quad \textit{alarm} = \textit{"failed-login"} \\ \textit{integer attempts} = \textit{1} \end{array}} \bullet \boxed{\begin{array}{l} \textit{string} \quad \textit{alarm} = \textit{"failed-login"} \\ \textit{integer attempts} = \textit{2} \end{array}}$$

to server 2, and would send the remaining filter h using a simple subscription **subscribe**(E, s_2) with

$$s_2 = \boxed{\begin{array}{l} \textit{string} \quad \textit{file} = \textit{"/etc/passwd"} \\ \textit{string operation} = \textit{"file-change"} \end{array}}$$

to servers 2 and 3. Server 1 will then start up a monitor that recognizes the sequence $(s_1 = f \cdot g) \cdot (s_2 = h)$.

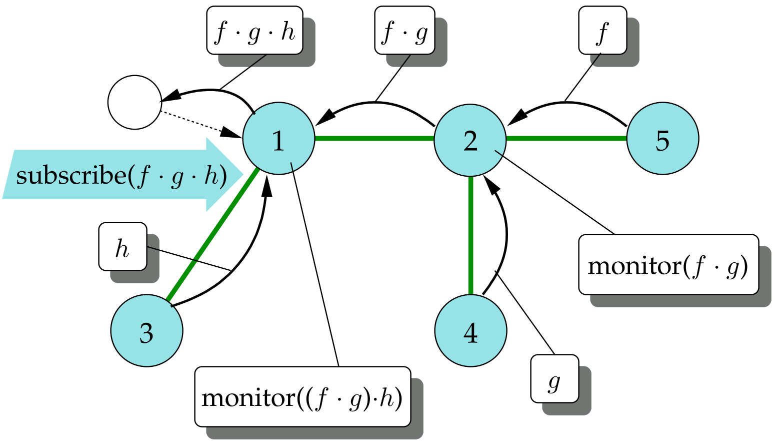

Figure 23: Pattern Monitoring and Delegation.

Figure 23 depicts an example that corresponds to the tables and subscriptions discussed above. In particular, server 1 delegates $f \cdot g$ to server 2, subscribes for h, and monitors $(f \cdot g) \cdot h$. The diagram also shows how server 2 handles the delegated subscription. Assuming that f is available from server 5 and g is available from server 4, server 2 sends the two corresponding subscriptions to 4 and 5 and then starts up a monitor for $f \cdot g$.

6 Evaluation

Substantiating claims of scalability for an event notification service is a difficult challenge. In particular, how does one demonstrate its ability to scale, when fully doing so would require the deployment of an implementation of the service to thousands of computers across the world? Conceding to pragmatics, our approach is to build an argument based on (1) reasoning qualitatively about the rationale for the expressiveness of the notification selection mechanism, (2) performing simulation studies to determine the relative performance of the various architectures under certain hypothetical usage scenarios, and (3) constructing a prototype implementation of the service as a proof of concept.

This section presents each of these three elements of the argument. Our conclusion is that, while more study is required to fully validate the design, the early evidence strongly suggests that we have achieved our goal of developing an event notification service suitable for use at the scale of a wide-area network.

6.1 Rationale for Chosen Expressiveness

The interface to the SIENA event notification service is a tailored application of the basic publish/subscribe protocol. Certain factors affecting the scalability of our design (such as network latency and data structure size) are intrinsic to the service and its use, and hence are beyond our control. The key factors we can control are the definitions of notifications, filters, and patterns, and the complexity of computing the covering relations.

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" 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 $\sqsubseteq_S^S$, 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 database query 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 determining whether a given subscription and a given notification are related by $\sqsubseteq_S^N$ is $O(n+m)$, where n is the number of attribute constraints in the subscription filter and m is the number of attributes in the notification. The complexity of computing $\sqsubseteq_S^N$ reflects the computation of an intersection between the attribute values in a notification and constraints on those values appearing in a subscription. The complexity of each individual comparison is $O(1)$ for all the predefined types we have included in SIENA. The only exception is the string type, but efficient comparison algorithms are well known.

The complexities of computing $\sqsubseteq_S^S$, $\sqsubseteq_A^A$, and $\sqsubseteq_A^S$ 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. Patterns, as we discuss in Section 3.4, are a simple sequence of filters. The computational complexity of matching 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 covering relations exhibit a complexity that is quite reasonable for a scalable event notification service. In fact, the factors n and m are, in practice, likely are to be relatively small (typically less than 10), making the computations negligible compared to the network costs they are attempting to reduce. This is all achieved with an expressiveness that approximates SQL.

6.2 Simulation Studies

A common practice in the field of computer systems is to perform simulation studies to gain feedback about a design before incurring the costs of implementation and deployment. For a wide-area event notification service, this means performing studies to determine the properties of the design under different network configurations and application behaviors. In a properly conducted study, these hypothetical usage scenarios will correspond to likely situations to be encountered by an actual implementation.

There are many questions that one could ask about a wide-area event notification service. In our initial studies we have concentrated on the particular question of scalability with respect to the architectures and algorithms described in the previous sections.

6.2.1 Simulation Framework

The simulation framework we use consists of two parts: (1) a configuration of servers and clients mapped onto the sites of a wide-area network and (2) an assignment of application behaviors to objects of interest and interested parties. The configuration of servers reflects the choice of the event notification service architecture, while the application behaviors involve the basic service requests of advertise/unadvertise, subscribe/unsubscribe, and publish.

The primary measurement of interest is an abstract quantity we refer to as *cost*. We assign a relative cost to each site-to-site communication in the network and then calculate the effect on this cost of varying a number of simulation parameters. In other words, we evaluate the architectures and algorithms in terms of the communication induced by the application behaviors, since we are interested in characterizing the degree to which each architecture/algorithm combination can or cannot absorb increased communication costs in the face of increasing application demands.

Network Configuration Figure 24 shows the layered structure of a network configuration in our simulation framework. At the bottom level is a model of a wide-area network topology. This model defines *sites* (depicted as cubes) and *links* (depicted as heavy lines between cubes). Sites are individual computers or local-area networks in which communication costs are considered to be negligible relative to the larger network. Links are the abstract representation of connections between two sites and have an associated cost for carrying the communication. To develop realistic network topologies that approximate the relative costs of real wide-area networks, we use a publicly available generator of random network topologies [29] that implements the Transit-Stub model [39]. (A discussion of this and other models for generating network topologies can be found elsewhere [40].) Although we studied networks of between 500 and 1000 sites, the results we present here were performed on networks of 100 sites.

At the top level of the network configuration is a model of an event notification service topology. This model defines the servers (depicted as shaded ovals) and clients (depicted as white ovals), with the interconnection among servers resulting from a choice of architecture (depicted as heavy lines between servers), the assignment of a client to a server (depicted as a dotted arrow), and the mapping of clients and servers onto sites in a wide-area network (depicted as dashed lines from ovals to cubes).

As a simplification, the simulations we present here involve only homogeneous architectures, and not the hybrid architectures that are also possible (see Section 4.4). Moreover, each client represents only either an object of interest or an interested party, although in general it is possible for a client to be both. Finally, we allocate one server per site, we configure every server to connect to the servers residing at its neighbor

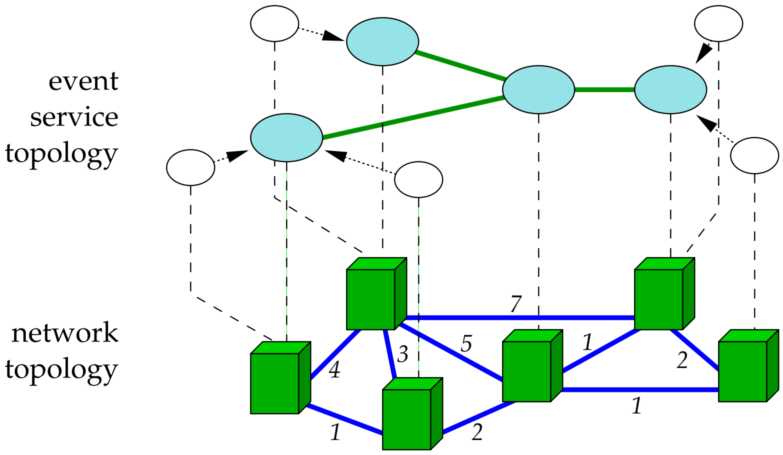

Figure 24: Layers in a Network Configuration.

sites in the network topology, and we configure every client to use a server at its (local) site. In other words, we assume that the locations and interconnections among servers are an image of the underlying network topology. This assumption significantly reduces the parameter space in the simulation. Nonetheless, this is a reasonable assumption, since it reflects the structure of domains that characterize the Internet [9].

In addition to simulating the various multi-server architectures, we simulate a single-server, centralized architecture. The centralized architecture serves as a reference architecture that we use as a baseline for our comparisons; where the centralized architecture performs as well as or better than the others, it should be chosen simply because of its simplicity. Of course, the centralized architecture requires no forwarding algorithm, since it comprises a single server.

Application Behavior The behavior of an application using the event notification service involves the collective behaviors of its objects of interest and interested parties. These individual behaviors are specified as sequences of service requests. In particular, an object of interest executes m sequences

$$\textit{advertise}, \overbrace{\textit{publish}, \textit{publish}, \ldots}^{n \text{ times}}, \textit{unadvertise}$$

running through m cycles of advertising, publishing, and then unadvertising, and within each cycle publishing n notifications. In addition, an average delay between publish requests, $\bar{t}$, can be specified (with the delays generated according to a Poisson distribution). In a corresponding manner, an interested party executes p sequences

$$\textit{subscribe}, \overbrace{\textit{recv\_notif}, \textit{recv\_notif}, \ldots}^{q \text{ times}}, \textit{unsubscribe}$$

where *recv_notif* represents the operation of waiting for and then receiving a notification.

Scenario Generation Input to our simulation tool is generated in a two-step process. In the first step a (random) network topology is generated, as discussed above. In the second step the generated topology is combined with a scenario parameter file that specifies both the event notification service topology and the application behavior.

Figure 25 shows an example scenario parameter file. The tag `servers` defines the architecture and algorithm to be used in the scenario, in this case the acyclic peer-to-peer architecture combined with the advertisement forwarding algorithm. A server is generated for each of the sites in the network topology and connected according to the specified architecture.

```
servers = acyclic adv_fwd;

objects {
   event = "A";
   number = 1000;
   pub/adv = 10;
   time/pub = 2000~2500;
   cycles = 10;
};

parties {
   pattern = "A";
   number = 10000;
   notif/sub = 10;
   cycles = 10;
};
```

Figure 25: Example Scenario Parameter File.

The sections `objects` and `parties` specify the number and behavior of objects of interest and interested parties, respectively. A single scenario parameter file can specify different kinds of behaviors for different objects of interest and interested parties, although the example in Figure 25, and the simulations presented here, use uniform behavior for every one. Objects of interest and interested parties are distributed randomly among all the sites.

The tag `event` gives the name of a type of event generated by the objects of interest. The corresponding tag `pattern` describes the pattern of events in which the interested party is interested. Because we are concerned here with evaluating the communication costs incurred by the different architectures and not the computational costs of the algorithms, we simplify the simulations by assuming trivial events, filters and patterns.

The tags `pub/adv` and `notif/sub` give the ratio of publication requests to advertisement requests (the value n mentioned above) and notification receipts to subscription requests (the value q mentioned above), respectively. The tag `cycles` gives the number of times the behavioral sequences are repeated (the values m and p mentioned above). The tag `time/pub` specifies a distribution for the average inter-publication delay for each client (the value $\bar{t}$ mentioned above). It is expressed as a pair of values *min* $\sim$ *max*, indicating a uniform distribution of values between *min* and *max*. Thus, the delay averages are distributed uniformly across clients, while each average itself is interpreted as the average from a Poisson distribution. Note that the total number of publications need not equal the total number of receipts; it is possible, for example, that a notification is published at a time when there are no subscribers for that notification, in which case the notification is ignored.

6.2.2 Results

The space of studies made possible by our simulation framework is quite extensive. Here we explore a portion of that space, focusing on usage scenarios that distinguish four basic architecture/algorithm combinations: centralized, hierarchical client/server with subscription forwarding, acyclic peer-to-peer with subscription forwarding, and acyclic peer-to-peer with advertisement forwarding. To reveal their scaling properties, our approach is to keep the behaviors of objects of interest and interested parties constant while varying the number of objects of interest from 1 to 1000 and the number of interested parties from 1 to 10000. In all cases, the number of network sites is 100. The behaviors are those specified in Figure 25. Notice that we are only simulating the objects of interest and interested parties associated with one particular kind of event. Simulating additional kinds of events (with associated objects of interest and interested parties) does not change the relative characteristics of the architectures and algorithms.

We ran our scenarios with artificially low ratios of publications-per-advertisement and notifications-per-subscription in order to produce conservative simulation results. Applications ultimately benefit from delivery of notifications, and so advertisements and subscriptions can be considered a necessary overhead to obtain that benefit. Such low ratios then serve to exaggerate this overhead. In real applications, we would expect the service to deliver a much higher volume of notifications (with correspondingly lower overhead) than is represented by these ratios.

The results we present are all shown as plots whose data points represent the average of 10 simulation runs for the same parameter values. For the majority of the plots, the horizontal axis gives the number of interested parties in a logarithmic scale ranging from 1 to 10000, while the vertical axis gives a linear measure of cost. As mentioned above, the cost values are derived from an assignment of relative costs for communicating over network links. Therefore, the absolute value of a data point's cost is meaningless, but its relative value gives a useful characterization.

In the figures below we use the following aliases for the event notification service architecture/algorithm combinations: *ce*=centralized, *hs*=hierarchical client/server with subscription forwarding, *as*=acyclic peer-to-peer with subscription forwarding, and *aa*=acyclic peer-to-peer with advertisement forwarding.

Total Cost A basic metric for the event notification service is the total cost of providing the service. The total cost is calculated by summing the costs of all site-to-site message traffic. Figure 26 compares the total costs incurred by each of the four architecture/algorithm combinations considered here, first with 10 objects of interest and then with 1000 objects of interest. The plot on the left represents a scenario in which the

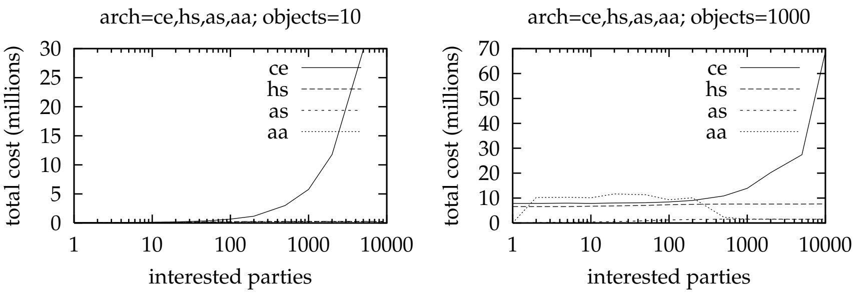

Figure 26: Comparison of Total Costs Incurred by Centralized and Distributed Architectures.

behavior is dominated by the number of subscriptions, whereas the plot on the right represents a scenario in which the behavior is dominated by the number of notifications. What we see, not surprisingly, is that in both cases, as the number of interested parties increases, the centralized architecture performs increasingly worse than the various distributed architectures. Although the centralized architecture exhibits linear cost growth, we would prefer to see the sublinear growth indicative of a powerful network effect that amortizes the costs of adding interested parties. The distributed architectures all show sublinear growth in total cost.

A more illuminating view of the total cost is given in Figure 27, where only the results for the distributed architectures are shown. Here we add plots for 10 and 100 objects of interest in addition to those for 1 and 1000 objects of interest. There are several interesting observations we can make about these plots.

First, all the architectures scale sublinearly when there are fewer than about 100 interested parties. This means that adding new subscribers for the same event notifications adds no cost. However, it is likely that the object of interest and interested party are not at the same site, so there is still a nonzero cost to deliver a notification.

Second, when there are more than 100 interested parties, the total cost becomes essentially constant, meaning that the notifications no longer add cost. We call this the *saturation point*, since there is high likelihood that there is an object of interest at every site, and thus an object of interest near every interested party. (Recall that all objects of interest are publishing the same notifications.)

Third, as the number of objects of interest increases, the hierarchical client/server architecture with subscription forwarding performs worse by an increasingly large constant factor as compared to the acyclic peer-to-peer architecture with subscription forwarding. This can be attributed to the fact that, while the acyclic peer-to-peer architecture is penalized by its broadcast of subscriptions, the hierarchical client/server architecture, which propagates subscriptions only towards the root of the hierarchy, is forced to do so

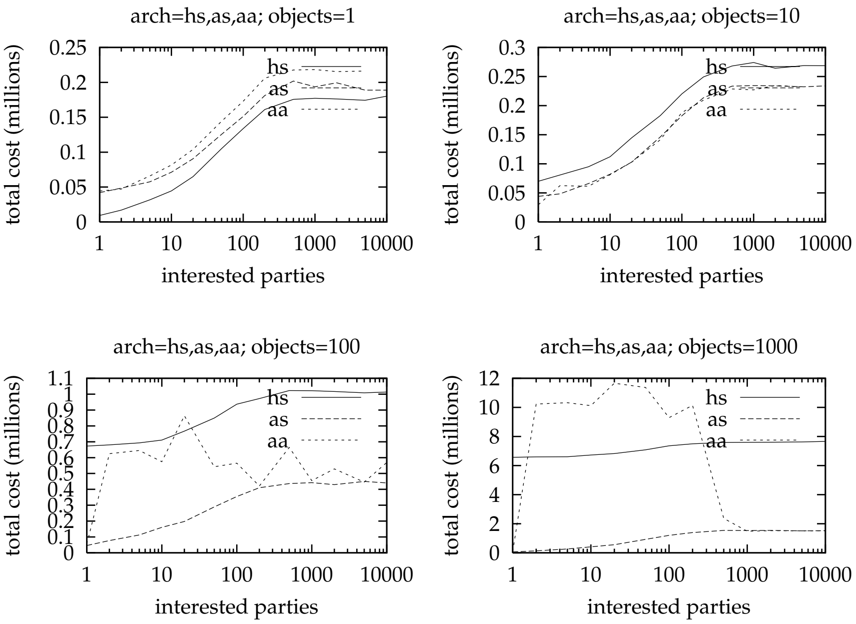

Figure 27: Comparison of Total Costs Incurred by Distributed Architectures.

whether or not interested parties exist on the other side of the root of the network. This generates a potentially significant traffic in unnecessary notifications.

Finally, the acyclic peer-to-peer architecture with advertisement forwarding displays a strikingly unstable cost profile for low densities of interested parties. On the other hand, its costs essentially follow those of the acyclic peer-to-peer architecture with subscription forwarding once the saturation point is passed. This effect becomes more evident as the number of objects of interest increase. We can attribute this to our conservative choice of behavior for objects of interest in these studies. In particular, an object of interest unadvertises and readvertises quite frequently, compared to the number publications it generates at each iteration.

A different view of the same data is shown in Figure 28. In this figure, each plot represents a single architecture/algorithm combination, with its total costs for different numbers of objects of interest compared. In all cases, the centralized architecture, shown in the upper left plot, explodes in total cost as the number of interested parties increases. The upper right plot shows that the hierarchical client/server architecture with subscription forwarding performs reasonably well with low numbers of objects of interest, but explodes in fixed total cost for high numbers of objects of interest. The bottom right plot shows the extreme instability of advertisement forwarding under high numbers of objects of interest below the saturation point for interested parties. The acyclic peer-to-peer architecture with subscription forwarding, shown in the bottom left plot, appears to scale well and predictably under all circumstances, and thus is likely to represent a good choice to cover a wide variety of scenarios.

Cost per Service Request An event notification service is efficient if it can amortize the cost of satisfying newer client requests over the cost of satisfying previous client requests. This is another manifestation of the network effect. The average per-service cost is calculated by dividing the total cost, as introduced above, by the total number of client requests. A low value for this ratio indicates low overhead. Recall that for these studies we configure the network so that clients are connected to servers at their local sites and, therefore, the client-to-server communication cost is treated as zero. The per-service cost thus purely reflects the choice of architecture/algorithm combination.

We can see several interesting things in the results of the data analysis presented in Figure 29. (For completeness, we again invert the presentation of the data, presenting in Figure 30 separate plots for each

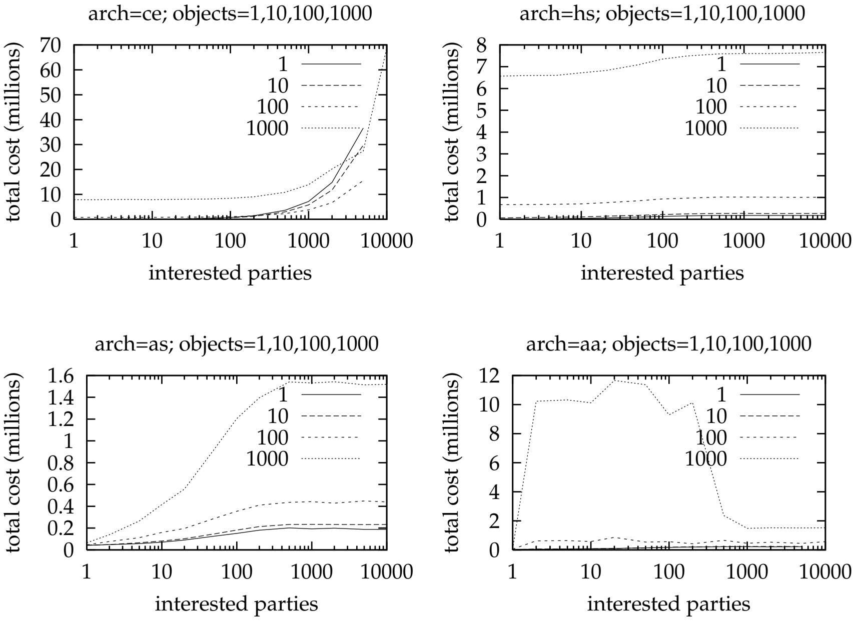

Figure 28: Comparison of Total Costs For Each Architecture/Algorithm Combination under Varying Numbers of Objects of Interest.

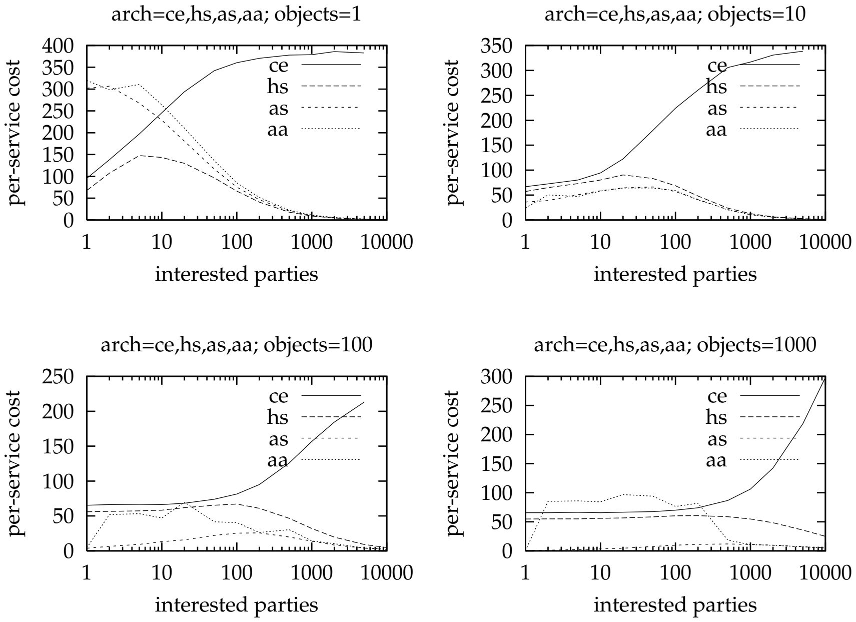

Figure 29: Comparison of Per-Service Costs.

architecture/algorithm combination.) First, the centralized architecture is unreasonable in essentially all scenarios as compared to the other architectures. Second, advertisement forwarding again shows itself unstable for high numbers of objects of interest until the saturation point in interested parties is reached. Third, for low numbers of objects of interest and low numbers of interested parties, the costs are dominated by message-passing costs internal to SIENA, since there are relatively few notifications generated in the network, there are few parties interested in receiving those notifications, and there is a significant internal cost incurred in setting up routing paths from objects of interest to interested parties. The hierarchical client/server architecture with subscription forwarding does well in this situation because subscriptions are forwarded only towards the root server, resulting in lower setup costs. However, as the number of objects of interest or the number of interested parties increases, its advantage quickly disappears, recovering only beyond the saturation point for interested parties. Finally, the acyclic peer-to-peer architecture with subscription forwarding does extremely well when there is a high number of objects of interest, independent of the number of interested parties. This effect is explained in the next series of plots.

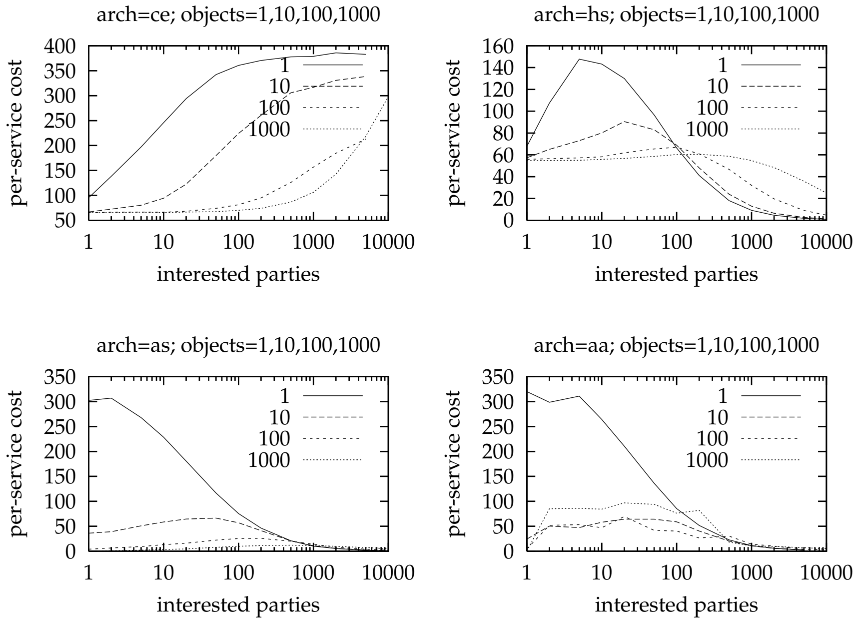

Figure 30: Comparison of Per-Service Costs For Each Architecture/Algorithm Combination under Varying Numbers of Objects of Interest.

Cost per Subscription and per Notification Based on the results of studying the total and per-service cost incurred by each of the four architecture/algorithm combinations, the hierarchical client/server architecture and acyclic peer-to-peer architecture, both with subscription forwarding, appear to be the two most promising choices. However, they are clearly distinguished if we examine which kind of service request each one favors for its optimizations.

The average per-subscription cost is calculated by dividing the total cost of all subscription-related messages by the number of subscriptions processed. The graph of Figure 31 shows the per-subscription cost incurred by the hierarchical client/server architecture and acyclic peer-to-peer architecture with a single object of interest. In trying to understand the different cost drivers of the two architectures, we simulated several scenarios with a single object of interest while varying only the behavioral parameters. In these cases, we observed no significant variation in cost. However, additional simulations, in which we varied the density of interested parties, highlight the difference between the two architectures. The results of these simulations are presented in Figure 31 and reveal that the costs are primarily dependent on the density of interested parties. In particular, the per-subscription cost is evidently higher for the acyclic peer-to-peer architecture than the hierarchical client/server for low densities of interested parties, while both architectures benefit from increasing densities of interested parties.

The main difference is in the way each architecture forwards subscriptions. In the acyclic peer-to-peer architecture, a subscription must be propagated throughout the network; in a network of N sites, a subscription goes through $O(N)$ hops and, therefore, the cost is $O(N)$. The hierarchical client/server architecture, on the other hand, requires that a subscription be forwarded only upward towards the root server; in this case the number of hops, and hence the cost, are both $O(\log N)$.

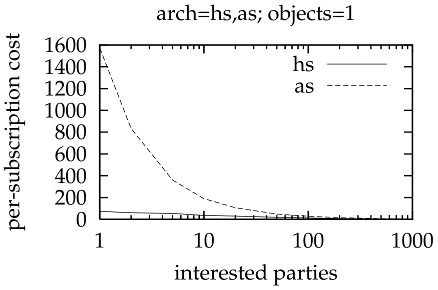

Figure 31: Comparison of Per-Subscription Costs For Hierarchical Client/Server and Acyclic Peer-to-Peer Architectures under Varying Numbers of Interested Parties.

The acyclic peer-to-peer architecture recoups its greater setup costs for subscriptions by reducing the average cost of notifications. Figure 32 compares the per-notification costs incurred by the acyclic peer-to-peer architecture and the hierarchical client/server architecture with a single object of interest. In the particular scenario of Figure 32, the difference between the per-notification costs in the two architectures is constant with respect to the number of interested parties. The same difference is clearly visible from the global per-service costs shown in of Figure 29.

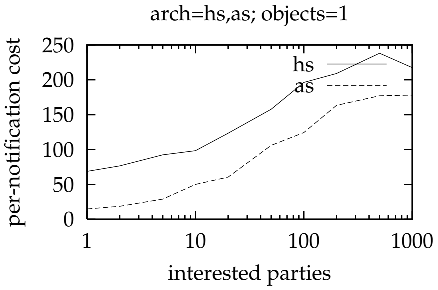

Figure 32: Comparison of Per-Notification Costs For Hierarchical Client/Server and Acyclic Peer-to-Peer Architectures under Varying Numbers of Interested Parties.

We observe that this constant bracket depends on the number of ignored notifications. In many of the scenarios we simulated, the total number of notifications produced by objects of interest exceeds the number of notifications consumed by interested parties. For example, in the scenario defined by the parameters of Figure 25, a total of 100000 notifications are produced, while each interested party consumes 100 notifications before terminating, leaving 99900 ignored notifications.

The cost of ignored notifications is clearly shown by the degenerate scenarios of Figure 33, in which per-notification costs are plotted against the number of notifications published. The average per-notification cost is calculated by dividing the total cost of all notification-related messages by the number of notification processed. The first scenario has one object of interest that emits a varying number of notifications and no interested parties at all. Here the hierarchical client/server architecture incurs a constant cost due to the fact that every notification must be propagated toward the root of the hierarchy, whereas the acyclic peer-to-peer architecture incurs no cost at all, since every notification remains local to its access server. The second scenario has one interested party that consumes exactly one notification and then terminates. Again, in the hierarchical client/server architecture, the per-notification cost is constant, while the acyclic peer-to-peer architecture incurs an initial cost for the first notification that is subsequently amortized by the zero cost of the subsequent ignored notifications.

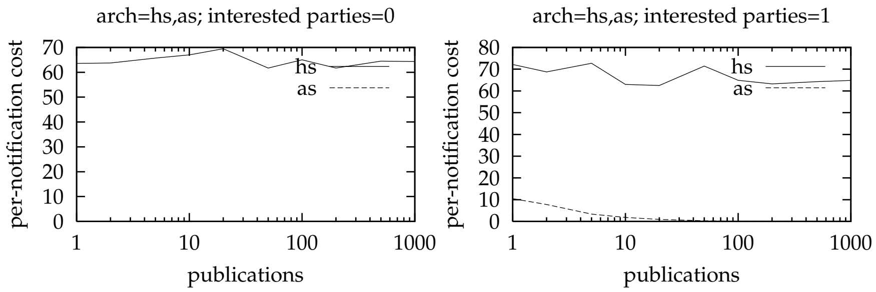

Figure 33: Comparison of Per-Notification Costs For Hierarchical Client/Server and Acyclic Peer-to-Peer Architectures under Varying Numbers of Publications in Two Degenerate Cases.

We can summarize the differences between the hierarchical client/server and acyclic peer-to-peer architectures as follows:

- The hierarchical client/server architecture has a lower ($O(\log N)$) per-subscription cost than the acyclic peer-to-peer ($O(N)$). This cost does not depend on the behavior of objects of interest or interested parties.
- In both architectures, the subscription cost is amortized for increased densities of interested parties. The cost difference between the two architectures is also significantly reduced for high densities of interested parties.
- The cost of delivering a notification to interested parties is more or less the same for the two architectures. However the acyclic peer-to-peer architecture has no cost for ignored notifications while the hierarchical peer-to-peer architecture pays a fixed cost ($O(\log N)$).

In practice, the hierarchical client/server architecture should be used where there are low densities of interested parties that subscribe (and unsubscribe) very frequently. The acyclic peer-to-peer architecture is more suitable to scenarios where the total cost is dominated by notifications, and especially where the total number of notifications exceeds the number of notifications consumed by interested parties, that is, in the presence of ignored notifications.

6.3 Prototype

We have implemented a prototype of SIENA that realizes the subscription-based event notification service.[4] The current implementation of SIENA offers two application programming interfaces, one for C++ and the other for Java. Both interfaces provide nearly the complete data model and subscription language described in Section 3. The *time* data type is the only one that has not yet been implemented.

Two event servers are also provided in 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 subscription forwarding. These two servers have been used together to form a hybrid topology.

[4] A binary version of the prototype is available at `http://www.cs.colorado.edu/serl/dot/siena.html`.

For the client/server and server/server communication in SIENA, we have developed a simple event notification protocol that we have implemented on top of TCP/IP connections. We have also encapsulated application-level protocols such as HTTP and SMTP.

7 Related Work

In this section we briefly review related work in event notification services. A more complete discussion of these topics is presented elsewhere [5].

7.1 Classification Framework

In order to understand and classify technologies that are related to SIENA, 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 7 presents such a comparison in terms of the architectures described in Section 4 and in terms of a classification of subscription languages shown in Table 8.

		Architecture		
		centralized	*hierarchical client/server*	*peer-to-peer*
Subscription Language	*channel-based*	Field [30] CORBA Event Service [26] Java Dist. Event Spec. [34] Java Message Service [35]	CORBA Event Service [26]	IP multicast [14] SoftWired's iBus
	subject-based	ToolTalk [20]	NNTP [21] JEDI [11] TIBCO's TIB/Rendezvous	Talarian's SmartSockets Vitria's BusinessWare
	content-based	Elvin [32]	Keryx [37] Yu et al. [38]	Gryphon [2]
	content-based with patterns	GEM [23] Yeast [22] CORBA Notification Service [27] object-oriented active databases† [8]	SIENA	SIENA

†Allows user-defined operators in subscription predicates; all others support only predefined operators.

Table 7: A Classification of Related Technologies.

		Scope		
		single notification single field	*single notification multiple fields*	*multiple notifications multiple fields*
Power	*simple equality*	channel-based	—	—
	expressions with predefined operators	restricted subject-based	restricted content-based	restricted content-based with patterns
	expressions with user-defined operators	general subject-based	general content-based	general content-based with patterns

Table 8: Typical Features of Subscription Languages.

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 predicates, ranging from a simple equality predicate, to expressions involving only predefined operators, to expressions involving user-defined operators. As we point out in Section 6, user-defined operators suffer from the disadvantage of having arbitrary, unknown, and potentially unbounded complexity.

From Table 8 we derive the four classes of subscription languages used in Table 7. In a *channel-based* language, a client subscribes for all notifications sent across an explicitly-identified channel, which is a discrete communication path. In a *subject-based* language, a client subscribes for all notifications that the publisher has identified as being relevant to a particular subject, which is selected from a predefined set of available subjects. The difference between channel-based and subject-based is that a channel typically allows only a straight equality test (e.g., *channel*=314 or *channel*="CNN") whereas a subject often subsumes richer predicates, such as wild-card string expressions on subject identifiers (e.g., *subject*="comp.os.∗"). In both cases, the filter applies to a single well-known field. In a *content-based* language, a client subscribes for all notifications whose content matches client-specified predicates that are evaluated on the content; evaluation of these predicates can be limited either to individual notifications (a simple content-based language) or to patterns of multiple notifications (a content-based language with patterns). We observe that subscription languages with user-defined predicates are rare; in Table 7 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.

In the remainder of this section, we discuss the relationship between SIENA and the other technologies mentioned in Table 7 in greater detail.

7.2 Message-Based Integrated Environments

The idea of integrating different components by means of *messages* was pioneered in a research system called Field [30]. As in several commercial products that followed (e.g., HP SoftBench [4], DEC FUSE [19], and Sun ToolTalk [20]), Field implements an environment in which several software development tools can cooperate by exchanging messages. Messages are the means by which one tool can request services to be carried out by other tools, or by which a tool announces a change of state—such as the termination of an operation—so that other tools can proceed with a dependent task.

Integrated software development environments embody a primitive event notification service, since registering tools to handle service requests is equivalent to subscribing for those requests. However, the domain of event notifications and subscriptions in these systems is limited. Tools can generate a fixed set of messages, and in some cases (e.g., in DEC FUSE), the set of messages is statically mapped into a set of callback procedures hard wired into the tool.

7.3 Event-Action Systems and Active Databases

Yeast [22] is a system supporting the definition of *rules* that specify the actions to be taken upon the occurrence of particular events. Unlike message-based integrated environments, Yeast is a general-purpose event notification service. Yeast defines a rich event pattern language that supports predefined operating system events (such as file modifications, user logins, and host load changes), events involving time, and user-defined events. The action part of a specification is a UNIX shell script. GEM [23] is a more recent language that allows one to specify event-action rules similarly to Yeast.

A different, more specialized kind of system that is conceptually equivalent to event-action systems are active databases [8]. In active databases, primitive events are operations on database objects, and event-action rules are called *triggers*. Events can be combined and correlated in the trigger. A trigger can also impose an additional condition or *guard*, to be evaluated once the requested sequence of events has occurred. If the guard is satisfied, the database executes the action part of the trigger.

The main difference between an event notification service like SIENA and an event-action system like Yeast is that an event notification service only dispatches event notifications, possibly to multiple recipients, so that responses to events (i.e., the actions) are executed by interested parties externally to the service. An event-action system, on the other hand, is also responsible for executing the actions taken in response to event notifications. Thus, it must have a logically centralized architecture, because actions are executed by the system itself within its environment. In other words, in an event-action system, there is a strict coupling between the observation of events and the reaction to those events, whereas SIENA decouples these elements of the problem.

7.4 Internet Technology

A number of Internet technologies are worth discussing because they realize services on a wide-area scale. Thus, even if none of them is really designed to realize an event notification service, they employ a variety of relevant techniques to achieve scalability.

7.4.1 Domain Name Service

DNS [24, 25] maps symbolic host or domain names into IP addresses. The current implementation of DNS has proved to be extremely scalable, especially considering the recent explosion of domains caused by the commercial exploitation of the Internet. DNS is realized with a distributed architecture. In particular, DNS servers form a hierarchical structure. The reason why a hierarchical architecture works so well for DNS is that the hierarchical structure of servers can be naturally laid out according to the structure of the address data that they manage. In fact, the space of host names and the space of IP addresses are hierarchical themselves, and the mapping between them preserves many of the hierarchical properties. In other words, because host names are partitioned into domains (e.g., `.edu`, `.it`, `.com`, etc.), which in turn are partitioned into sub-domains (e.g., `.colorado.edu`, `.uci.edu`, etc.) and so on, the physical architecture of DNS can be set up so that requests that pertain to one domain are handed off to a server dedicated to that domain, and no other host outside that domain can affect the mapping realized within the domain.

Although we can adopt techniques that are inspired by DNS, the same hierarchical partition does not appear to be valid for a general-purpose event notification service. In fact, the space of event notifications does not exhibit any hierarchical structure and, even if we decided to force this type of structure—which some systems do by defining a "subject" attribute for notifications and by partitioning its possible values—this would not naturally map onto a hierarchical location of objects. In other words, we cannot assume that events of a particular kind (or subject) occur only within a particular group of sites or are requested only by a particular set of interested parties located in a specific subnet.

Another differentiator of DNS with respect to an event notification service is the essential read-only nature of the DNS service. DNS is able to resolve names very efficiently because the mappings maintained by DNS are relatively stable in the sense that they are read much more frequently then they are modified. This allows DNS to employ caching to increase its performance and scalability. In general, this stability and read-only character are not true of the information exchanged in general-purpose event notification services.

7.4.2 USENET News

The USENET News system, with its main protocol NNTP [21], is perhaps the best example of a scalable, user-level, many-to-many communication facility. USENET News articles are modeled after e-mail messages, yet they provide additional information in headers that can be used by NNTP to direct their distribution. The infrastructure that supports the propagation of articles comprises a network of news servers. New servers can join the infrastructure by connecting as *slave* to another (*master*) server that is already part of the infrastructure and that is willing to share articles. The structure of servers thus formed is a tree in which articles are flooded from master servers to slave servers. Besides receiving the feed of articles from its master server, a slave server can also send locally-posted articles to its master server.

Articles are posted to *newsgroups*, each group roughly representing a discussion topic. Groups are organized in a hierarchical name/subject space. NNTP provides some primitive filtering capabilities on articles. Articles can be selected by means of simple expressions denoting sets of group names and also based on the dates of postings. For example, a slave server can request all the groups in `comp.os.*` that have been posted after a given date.

The USENET News infrastructure is intended for the exchange of generic information and for discussions rather than for dispatching event notifications. Also, the information is not simply passed through the infrastructure, but is replicated and persistently stored within it.

The main problem with the USENET News infrastructure and with NNTP that limits their applicability as a general-purpose event notification service is that the selection mechanisms are not very sophisticated. Although group names and sub-names reflect the general subject and content of messages, the filter that they realize is too coarse-grained for most users and definitely inadequate for a general-purpose event notification service. This is also proved by the fact that most news readers (the client programs of USENET News) allow users to perform additional sophisticated filtering to discard uninteresting messages once the messages have been transferred from the server. The limited expressiveness of the USENET News system may result in unnecessary transfers of entire groups of messages over the network. The service is scalable but still quite heavyweight, and in fact the time frame of news propagation ranges from hours to days, which is inadequate for an event notification service.

7.4.3 IP Multicast

IP multicast [13] is a network-level infrastructure that extends the Internet protocol in order to realize an efficient one-to-many communication service. IP multicast is an extension of the usual unicast routing mechanism realized over the Internet. The network that realizes this extension is also referred to as the MBone. In the MBone, a multicast address (or host group address) is a virtual IP address that corresponds to a group of hosts possibly residing on different subnets. IP datagrams that are addressed to a host group are routed to every host belonging to the group. Hosts can join or leave a group at any time using a special group membership protocol [16].

We consider the IP multicast infrastructure and its routing algorithms to be the most important technology related to SIENA. As a first observation, we note that IP multicast can be used as an underlying transport mechanism for notifications. But the most important aspect is that an event notification service can be thought of as a multicast communication infrastructure in which addresses are not explicit host addresses but rather arbitrary expressions of interest. With this model, the ideas developed for routing multicast datagrams can be adapted to solve the problem of forwarding notifications in an event notification service.

Unfortunately, the IP multicast infrastructure has some major limitations when used as a generic event notification service. The first issue is the lack of scalability when attempting to map expressions of interest into IP group addresses. The second issue, related to the first, is the limited expressiveness of IP addresses. In fact, even assuming that we could encode subscriptions as IP multicast addresses, that there exist enough multicast addresses, and that a separate service, perhaps similar to DNS, is available for managing and resolving that mapping, the addressing scheme itself still poses major limitations when attempting to observe combinations of events or when combining different subscriptions into more generic ones. Because IP multicast never relates two different IP groups, it would not be possible to exploit similarities between subscriptions mapped to IP group addresses. Different notifications matching more than one subscription or matching partially overlapping patterns would have to be sent to several separate multicast addresses, each one being routed in parallel and independently by the IP multicast network.

7.4.4 Active Networks

An *active network* is a network with programmable switches [36]. Certain packets sent through the network carry program code that can alter the routing behavior of the switches (including modification of routed packets). In a sense, the programmability feature of active networks is a form of content-based routing, since the content of the packets can govern packet routing in a very expressive manner and can be used to achieve routing optimizations. But while active networks themselves are not a general-purpose event notification service like SIENA, they could nevertheless possibly be used as an implementation platform.

7.5 Event-Based Infrastructures

Some technologies specifically realize an event notification service. Among them are standards proposals such as the CORBA Event Service [26], and research systems such as IBM's Gryphon [1, 2], Elvin [32], JEDI [10], Keryx [37], and the recent work of Yu et al. [38]. All of these systems support some form of publish/subscribe service, which is also the favored style of commercial middleware products such as SoftWired's iBus, TIBCO's TIB/Rendezvous™, Talarian SmartSockets™, Hewlett-Packard's E-speak™, and the messaging system in Vitria's BusinessWare™.

Gryphon is a distributed content-based message brokering system similar to SIENA. The focus of the Gryphon project is on the efficiency of matching and distributing messages based on their content. The techniques developed in Gryphon are complementary to the ones of SIENA. In particular, Gryphon uses a fast algorithm to match a notification to a large set of subscriptions [1]. This algorithm, similar to the one described by Gough and Smith [18], exploits commonalities among subscriptions. That is, whenever two or more subscriptions specify a constraint on the same attribute, the algorithm organizes them in order to test the value of that attribute in each notification once for all the subscriptions. The main difference with SIENA is that Gryphon propagates every subscription everywhere in the network, whereas SIENA propagates only the most generic subscriptions.

Yu et al. propose an event notification service implemented using a peer-to-peer architecture of proxy servers, with one server being the distinguished "root" server. In a sense their architecture is an amalgam of SIENA's hierarchical and acyclic peer-to-peer architectures. They believe a hierarchical arrangement of servers to have superior scalability to a non-hierarchical one. However, they have not yet simulated or implemented their architecture, so this architecture's scalability properties are yet to be determined.

Elvin is a centralized event dispatcher that has a quite rich event filtering language that allows complex expressions of interest. The centralized architecture facilitates efficient event filtering, although it poses severe limitations to its scalability. Keryx also provides structured event notifications and filtering capabilities extended to the whole structure of events. Keryx has a distributed architecture similar to that of USENET News servers. The architectures of TIB/Rendezvous and JEDI are also hierarchical, although their subscriptions are based on a simplified regular expression applied to a single string—the "subject" in TIB/Rendezvous or the entire notification in JEDI.

Even simpler is the selection mechanism offered by iBus and by the CORBA Event Service. They both adopt channel-based addressing, whereby parties can subscribe or listen to a channel and applications can explicitly post event notifications to channels. Note that the channel abstraction is exactly equivalent to the one given by a mailing list or an IP multicast group address, so channel-based event notification services are functionally identical to reliable multicast with a one-to-one mapping between channels and multicast addresses. In fact, iBus uses a transport mechanism based on IP multicast in which the mapping is the identity function. Some implementations of CORBA recognize the shortcomings of the channel-based selection and thus allow additional filtering based on notification contents. The importance of these improvements has been recently recognized by the OMG, which has formalized them in the specification of the CORBA Notification Service [28]. However, this additional filtering capability is simply retrofitted into the channel schema of the CORBA Event Service, which remains the main publish/subscribe interface to applications. The implementations of the CORBA Event Service that we know of have a centralized architecture; to create federations of channels, application builders must program the federations themselves.

In addition to the technologies described above, there is a class of infrastructures that do not realize an event notification service, although they are publicized as such. These infrastructures are component frameworks of virtual classes (or interfaces) in an object-oriented language supporting an event-based interaction among software components. The most significant example is the Java™ Distributed Event Specification [34]. This framework defines the Java interfaces of roles such as *event generators*, *event listener*, and *event notifications*. Typically, an event generator exports a *register* method that allows event listeners to declare their interest in the events emitted by the event generator. The listener interface defines a method called *notify* that will be called by the event generator whenever an event occurs. This framework of classes allows a distributed interaction because both event generators and event listeners are *remote* objects and their interaction is handled by means of RMI calls [15, 33].

Events in the Java™ Distributed Event Specification are objects implementing the *RemoteEvent* interface, so the modeling capability for events is good. However, the filtering capability is rather limited. Event notifications have an event *id* of type *long*, and a listener can select events based only on their *id*. Recently, Sun Microsystems has published the specification of the Java™ Message Service [35] which is an evolution of the Distributed Event Specification with significant enhancements to the expressiveness of their "message selectors" (filters). The message model and the filtering capabilities of Java Message Service are practically identical to those of SIENA.

It should be clear that, regardless of the expressive power of notifications and subscriptions, frameworks like Java Message Service and CORBA Notification Service specify interfaces only. They do not provide nor recommend any architecture or algorithm for the implementation.

8 Conclusions

In this paper, we have described our work on SIENA, a distributed, Internet-scale event notification service. We have described the design of the interface to the service, its semantics, the topological arrangements of event servers, and the routing algorithms that realize the service over a network of servers. The simulations that we performed confirm our intuitions about the scalability of the topologies and algorithms that we have studied. We plan on continuing to explore the parameter space of our simulations in several directions. In particular, we are simulating different ranges of behavioral parameters to see which algorithms are most sensitive to different classes of applications.

We plan on extending our design and prototype implementation of SIENA in a number of ways. For instance, we plan to enhance the design of the interface and algorithms to support mobility of clients. We also plan to implement the advertisement forwarding algorithm in the prototype, which will also allow us to apply the pattern matching optimizations that we discussed. Finally, we plan to work on extensions that support the expression of *quality of service* parameters especially suited to the integration of software components. These new features would allow the implementation of grouping mechanisms, such as transactions for notifications.

A significant new direction we intend to explore in the future is a realization of content-based routing as a fundamental network service provided within the physical network fabric itself [7]. This essentially involves replacing the architecture described at the beginning of Section 4, where we assume an event notification service such as SIENA to be implemented on top of a lower-level network protocol such as TCP/IP. Viewed differently, this involves embedding the content processing and routing capabilities of the upper layer of Figure 24 within the network topology of the lower layer of that figure. Content-based routing supported in this fashion could portend even more efficient and scalable even notification services, supported by a new class of networks built from high-speed content-based routers.

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] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Eighteenth ACM Symposium on Principles of Distributed Computing (PODC '99)*, Atlanta GA, USA, May 4–6 1999.

[2] 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 19^{th} IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, Austin, TX USA, May 1999.

[3] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, Dec. 1993.

[4] M. R. Cagan. The HP SoftBench environment: an architecture for a new generation of software tools. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 41(3):36–47, June 1990.

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

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)*, Portland, OR, July 2000.

[7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, Jan. 2000.

[8] S. Ceri and J. Widom. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Mateo, 1996.

[9] D. Clark. Policy routing in internet protocols. Internet Requests For Comments (RFC) 1102, May 1989.

[10] 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.

[11] 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.

[12] Y. K. Dalal and R. M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, Dec. 1978.

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

[14] 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.

[15] J. Farley. *Java Distributed Computing*. The Java Series. O'Reilly & Associates Inc., 1997.

[16] W. Fenner. Internet group management protocol, version 2. Internet Requests For Comments (RFC) 2236, Nov. 1997.

[17] 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.

[18] J. Gough and G. Smith. Efficient recognition of events in a distributed system. In *Proceedings of the 18^{th} Australasian Computer Science Conference*, Adelaide, Australia, Feb. 1995.

[19] R. O. Hart and G. Lupton. DEC FUSE: Building a graphical software development environment from UNIX tools. *Digital Technical Journal of Digital Equipment Corporation*, 7(2):5–19, Spring 1995.

[20] A. M. Julienne and B. Holtz. *ToolTalk and open protocols, inter-application communication*. Prentice–Hall, Englewood Cliffs, New Jersey, 1994.

[21] 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.

[22] B. Krishnamurthy and D. S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857, Oct. 1995.

[23] 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.

[24] P. Mockapetris. Domain Names - Concepts And Facilities. Internet Requests For Comments (RFC) 1034, Nov. 1987.

[25] P. Mockapetris. Domain names - implementation and specification. Internet Requests For Comments (RFC) 1035, Nov. 1987.

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

[27] Object Management Group. Notification service. Technical report, Object Management Group, Nov. 1998.

[28] Object Management Group. Notification service. Technical report, Object Management Group, Aug. 1999.

[29] G. I. of Technology. College of Computing. Georgia tech internet topology models (gt-itm). http://www.cc.gatech.edu/projects/gtitm.

[30] S. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–66, July 1990.

[31] 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.

[32] 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.

[33] Sun Microsystems, Inc., Mountain View CA, U.S.A. *Remote Method Invocation Specification*, 1997.

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

[35] Sun Microsystems, Inc., Mountain View CA, U.S.A. *Java Message Service*, Nov. 1999.

[36] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[37] 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.

[38] 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.

[39] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM '96*, San Framcisco CA, U.S.A., Apr. 1996.

[40] E. W. Zegura, K. L. Calvert, and M. J. Donahoo. A quantitative comparison of graph-based models for internet topology. *IEEE/ACM Transactions on Networking*, 5(6), Dec. 1997.