Design and Evaluation of a Wide-Area Event Notification Service

Antonio Carzaniga
University of Colorado at Boulder
David S. Rosenblum
University of California, Irvine
and
Alexander L. Wolf
University of Colorado at Boulder

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 and implemented 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 also present results of simulation studies that examine the scalability and performance of the service.

Categories and Subject Descriptors: C.2.1 [**Network Architecture and Design**]: Distributed Networks; Network Communication; Network Topology; Store and Forward Networks; C.2.2 [**Network Protocols**]: Applications; Routing Protocols; C.2.4 [**Distributed Systems**]: Client/ser-

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-94-C-0253, F30602-97-2-0021, F30602-98-2-0163 and F30602-99-C-0174; by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061; and by the National Science Foundation under Grant Number CCR-9701973. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory or the U.S. Government.
Address: Antonio Carzaniga and Alexander L. Wolf, Department of Computer Science, University of Colorado at Boulder, 430 UCB, Boulder, CO 80309-0430, USA, {carzanig,alw}@cs.colorado.edu
Address: David S. Rosenblum, Department of Information & Computer Science, University of California, Irvine, Irvine, CA 92697-3425, USA, dsr@ics.uci.edu

ver—*Distributed applications*; C.2.5 [**Local and Wide-Area Networks**]: Internet; C.2.6 [**Internetworking**]: Routers; C.4 [**Performance of Systems**]: Design Studies; Modeling Techniques; I.6.3 [**Simulation and Modeling**]: Applications; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete event*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: event notification, publish/subscribe, content-based addressing and routing

1. INTRODUCTION

The asynchrony, heterogeneity, and inherent loose coupling that characterize applications in a wide-area network promote *event interaction* as a natural design abstraction for a growing class of software systems. An emerging building block for such systems is an infrastructure called an *event notification service* [Rosenblum and Wolf 1997].

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* information about events and subsequently to *publish* multiple notifications of the kind previously advertised. Thus, an advertisement expresses the client's intent to publish a particular kind of notification. 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 [Carzaniga et al. 2000a]. 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. We recognize that there are other important attributes of an event notification service besides expressiveness and scalability, including security, reliability, and fault tolerance, but we do not address them in this paper.

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.

This paper presents *Siena*, an event notification service that we have designed and implemented 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 [Birman 1993]. 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 specifically for distributed networks of 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. In our simulation studies, we focus on two alternative service architectures, one based on a hierarchical topology, and the other based on a peer-to-peer topology. In particular, we study how these two architectures perform under various classes of applications in which the densities and distributions of clients differ. 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,

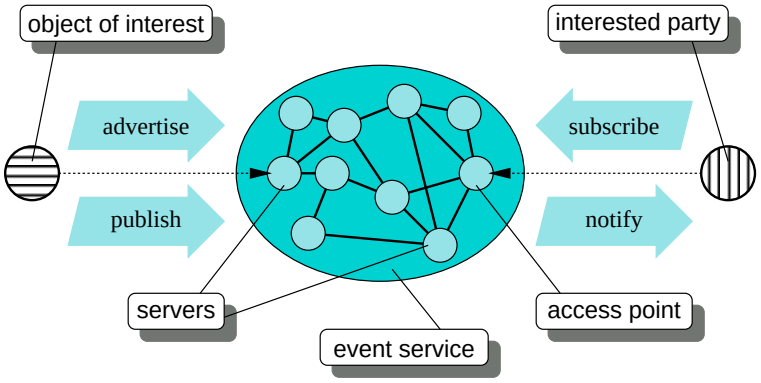

Fig. 1. Distributed Event Notification Service.

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.

—*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 [Cugola et al. 1998] 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 struc-

ture 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 [Sun Microsystems, Inc. 1998] and the CORBA Notification Service [Object Management Group 1998b]).

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 two functions that together define what is usually 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*. 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 1 shows the interface functions of SIENA. The expression given to *subscribe* and *unsubscribe* is a *pattern*, while the expression given to *advertise* and *unadvertise* is a *filter*; we discuss patterns and filters in greater detail below. The parameter

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

Table 1. Interface of SIENA.

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.

We note that SIENA maintains a mapping between the identities of interested parties and their *handlers*. A handler specifies the means by which an interested party receives notifications, either through callbacks or through messages sent via a communication protocol such as HTTP or SMTP. 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, and also supports the mobility of clients. Detailed discussion of handlers is beyond the scope of this paper.

3.1 Notifications

An *event notification* (or simply a *notification*) is a set of typed attributes. For example, the notification displayed in Figure 2 represents a stock price change event. Each individual attribute has a *type*, a *name*, and a *value*, but the notification as a

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*

Fig. 2. Example of a Notification.

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. In other systems, such as one finds for example in the Java Distributed Event Specification [Sun Microsystems, Inc. 1998] and CORBA Notification Service [Object Management Group 1998b], a notification is a value of some named, explicit notification type.

This implies 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.

3.2 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 ($=$, $\neq$, $<$, $>$, 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 constraint $\phi = (type_\phi, name_\phi, operator_\phi, value_\phi)$ 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 $\alpha \prec \phi$. 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*

Fig. 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 ($n \prec_S^N f$ for short):

$$n \prec_S^N f \Leftrightarrow \forall \phi \in f : \exists \alpha \in n : \alpha \prec \phi$$

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 2 gives some examples that illustrate the semantics of $\prec_S^N$. The second example is not a match because the notification is missing a value for attribute *level*. The third example is not a match because the constraints specified for attribute *level* in the subscription are not matched by the value for *level* in the notification.

3.3 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,

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

Table 2. Examples of $\prec_S^N$.

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 [Mansouri-Samani and Sloman 1997].

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

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

•

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

Fig. 4. Example of an Event Pattern.

general, we observe that more sophisticated forms of patterns can always be split into a set of simple subscriptions and then matched externally to SIENA (i.e., at the access point of the 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 3.8). 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.

3.4 Advertisements

We have seen how the covering relation $\prec_S^N$ defines the semantics of filters in subscriptions. We now define the semantics of advertisements by defining a similar relation $\prec_A^N$. 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 $\prec_A^N$ defines the set of notifications covered by an advertisement:

$$n \prec_A^N a \Leftrightarrow \forall \alpha \in n : \exists \phi \in a : \alpha \prec \phi$$

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 3 shows some examples of the relation $\prec_A^N$. 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.

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

Table 3. Examples of $\prec_A^N$.

3.5 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 $\prec_S^N$ 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 $n \prec_S^N s$.

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 $n \prec_A^N a$) and X registered a subscription s that covers n (i.e., such that $n \prec_S^N s$).

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

3.6 Other Important Covering Relations

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

—$\alpha \prec \phi$: attribute α matches attribute constraint ϕ;

—$n \prec_S^N f$: notification n matches filter f, where f is interpreted as a subscription filter;

—$n \prec_A^N a$: notification n matches filter a, where a is interpreted as an advertisement filter;

From these, other relations can be derived:

—$f_2 \prec_S^S f_1$: filter f_1 covers filter f_2, where f_1 and f_2 are interpreted as subscriptions. Formally,

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

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

—$a_2 \prec_A^A a_1$: filter a_1 covers filter a_2, where a_1 and a_2 are interpreted as advertisements. Formally:

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

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

—$f \prec_A^S a$: filter a covers filter f, where a is interpreted as an advertisement and f is interpreted as a subscription. Formally,

$$f \prec_A^S a \Leftrightarrow \exists n : n \prec_A^N a \wedge n \prec_S^N f$$

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

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

$$f_1 = f_2 \Leftrightarrow f_1 \prec f_2 \wedge f_2 \prec f_1$$

.

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

3.7 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 $g \prec_S^S f$). 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 $g_1 \prec_S^S f_1 \wedge g_2 \prec_S^S f_2 \wedge \ldots \wedge g_k \prec_S^S f_k$. In an analogous way, unadvertisements cancel previous advertisements that are covered according to the relation $\prec_A^A$.

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$\leq$ 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].

3.8 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.[1] 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 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,

[1] 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.

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*.

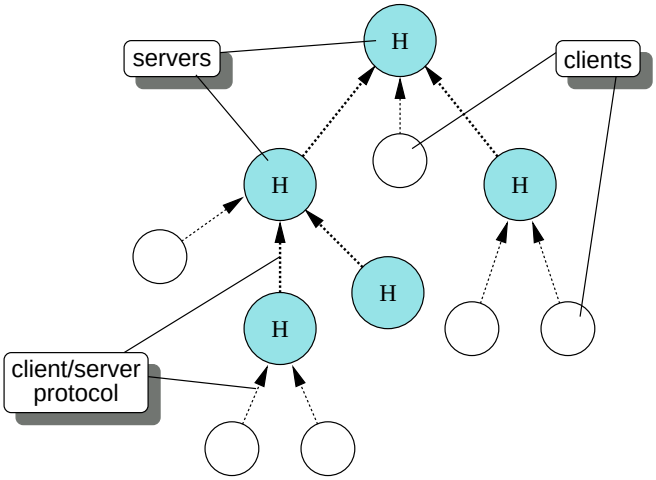

Fig. 5. Hierarchical Client/Server Architecture.

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 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 6.2.2.3, 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.) 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.

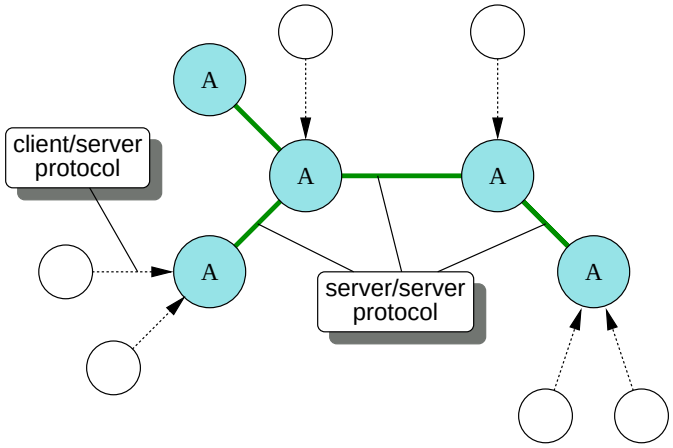

Fig. 6. Acyclic Peer-to-Peer Server Architecture.

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.

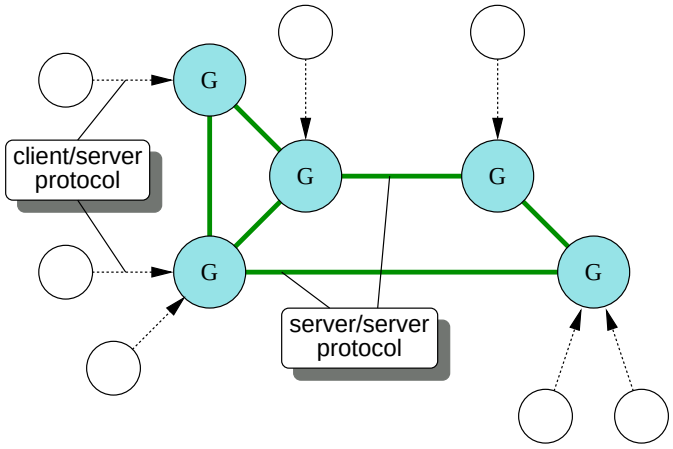

Fig. 7. General Peer-to-Peer Server Architecture.

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 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.

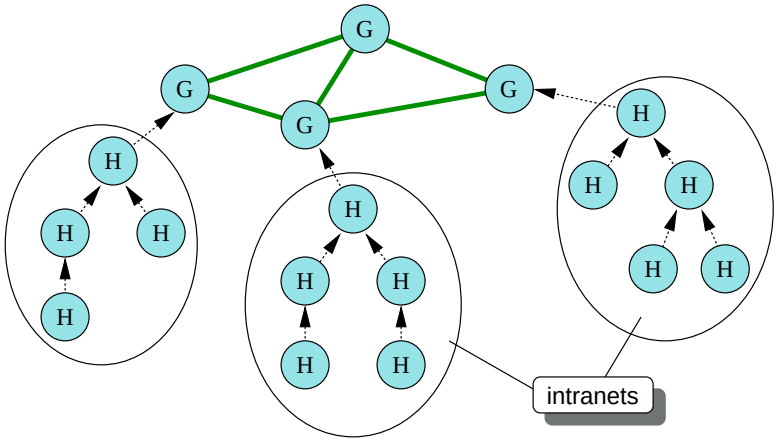

Fig. 8. Hierarchical/General Hybrid Server Architectures.

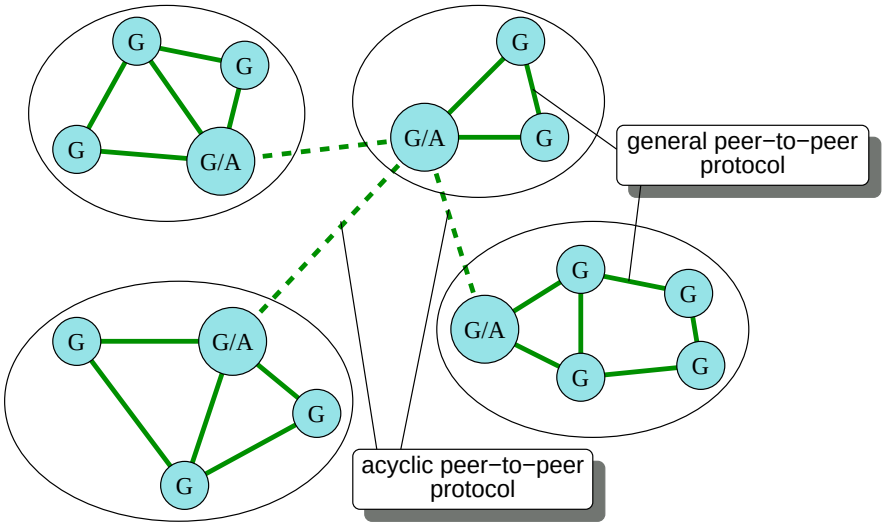

Fig. 9. General/Acyclic Hybrid Server Architectures.

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. 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 [Deering and Cheriton 1990]. 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 broad-

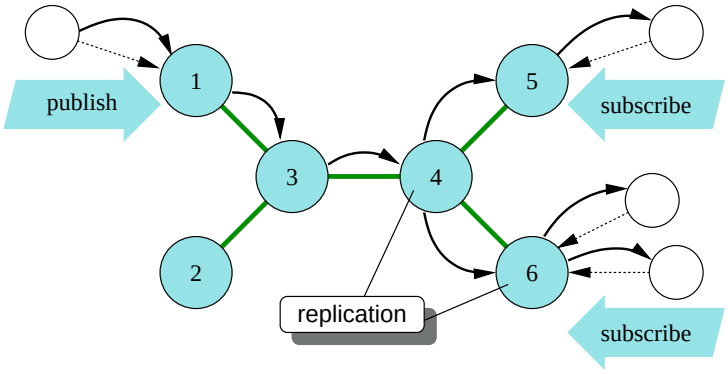

Fig. 10. Downstream Notification Replication.

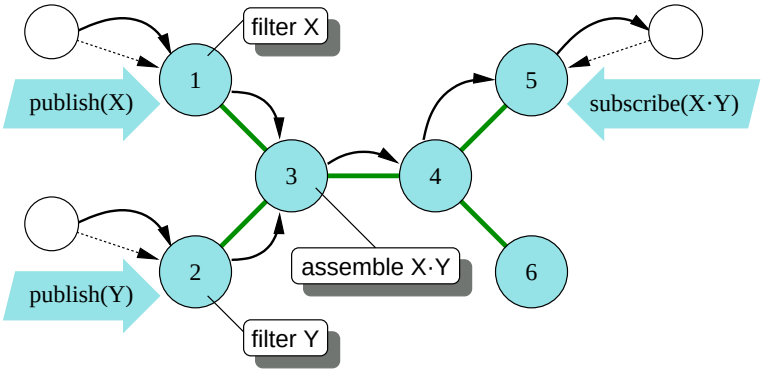

Fig. 11. Upstream Filter and Pattern Evaluation.

casting advertisements:

subscription forwarding: In an implementation that does not use advertisements, the routing paths for notifications are set by subscriptions, which are propagated throughout the network so as to form a tree that connects the subscribers to all the servers in the network. When an object publishes a notification that matches that subscription, the notification is routed towards the subscriber following the reverse path put in place by the subscription.

advertisement forwarding: In an implementation that uses advertisements, it is safe to send a subscription only towards those objects of interest that intend to generate notifications 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 $\prec_S^S$ 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 $\prec_A^A$. 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 5.3, advertisement forwarding algorithms are needed in order to implement the upstream evaluation principle for event patterns.

In addition to the principles introduced above, we have also devised several other strategies that lead to optimizations in resource usage. These are discussed elsewhere [Carzaniga 1998; Carzaniga et al. 1999].

5.2 Putting Algorithms and Topologies Together

We now describe in detail how subscription forwarding and advertisement forwarding algorithms are implemented within 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 5.3 deals with patterns.

At a high level and in general terms, the algorithms for the acyclic and peer-to-peer architectures attempt to reduce communication, storage, and computation costs by pruning spanning trees over a network of SIENA servers. More specifically, the subscription-forwarding algorithms operate by broadcasting subscriptions along spanning trees rooted at interested parties. When a server receives a new subscription, it can terminate the further propagation of that subscription if it has already propagated a more general subscription that covers the new one. In this way servers prune spanning trees along which new subscriptions are propagated. The

advertisement-forwarding algorithms operate in an analogous fashion by pruning the spanning trees rooted in objects of interest. Computation of spanning trees in a network is a solved problem [Dalal and Metcalfe 1978] and, therefore, we do not discuss their construction. Instead, our focus is on the details of pruning. The algorithms for the hierarchical architectures are simpler, because subscriptions and advertisements are not propagated along spanning trees, but are merely propagated along unique paths to the root of the hierarchy.

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 $\prec_S^S$ for subscription filters, and $\prec_A^A$ for advertisement filters. We denote with P_S a poset defined by $\prec_S^S$, and denote with P_A a poset defined by $\prec_A^A$. Figure 12 shows an example of a poset of subscriptions.

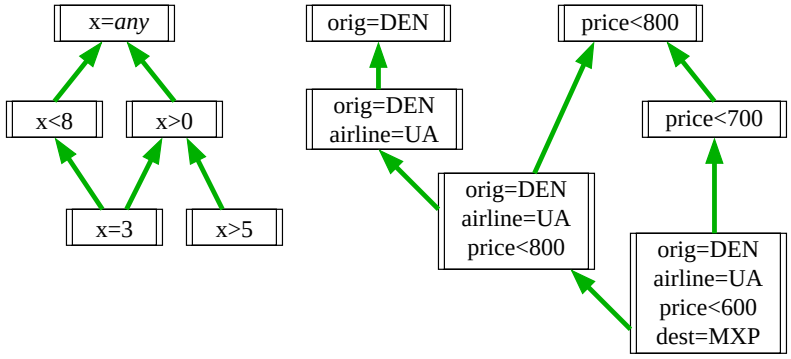

Fig. 12. Example of a Poset of Simple Subscriptions. Arrows represent the *immediate* relation $\prec_S^S$.

Note that $\prec_S^S$ and $\prec_A^A$ are transitive relations, while the diagram and its representation in memory store *immediate* relationships only. In a poset P_S, ordered according to $\prec_S^S$, a filter f_1 is an *immediate predecessor* of another filter f_2 and f_2 is an *immediate successor* of f_1 if and only if $f_1 \prec_S^S f_2$ and there is no other filter f_3 in P_S such that $f_1 \prec_S^S f_3 \prec_S^S f_2$. The "top-level" filters, which we refer to as *roots*, are those that have no successors in the poset.

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 successors.

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 reflects the degree of opportunity presented to our processing 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 maintains its subscriptions in a poset P_S. Each subscription s in P_S has an associated set called *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.

5.2.2.1 *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: $f \prec_S^S f' \wedge X \in \mathit{subscribers}(f')$. 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 successors and the immediate predecessors 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 Figure 13. Figure 13a 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).

In Figure 13b, 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 13c, 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

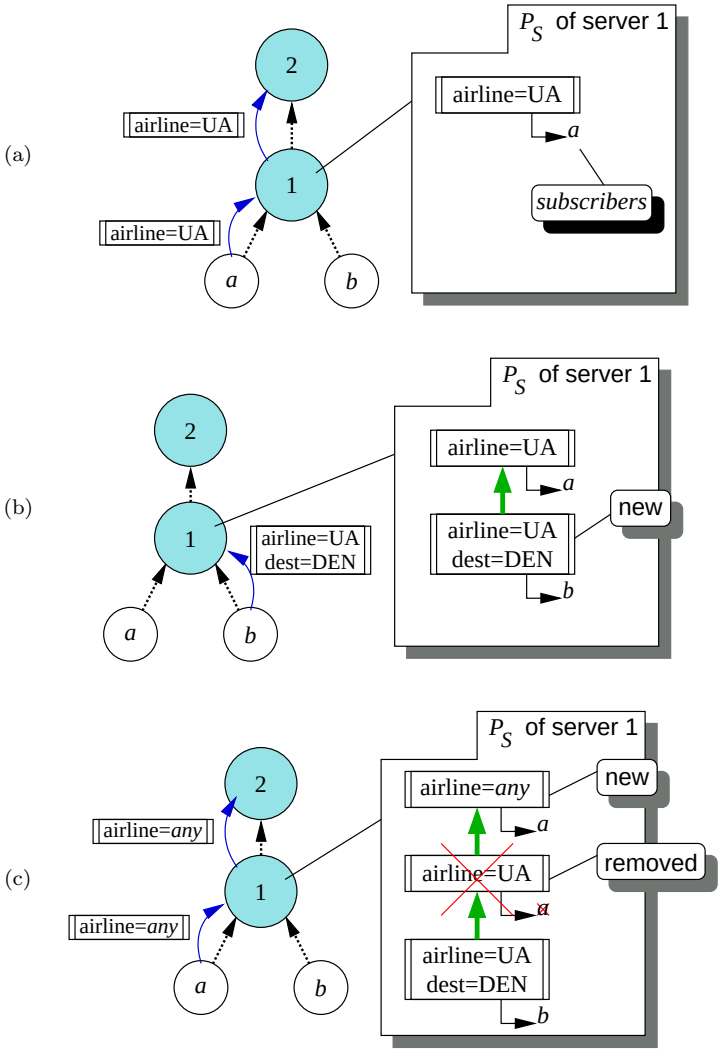

Fig. 13. Example Scenario Using a Hierarchical Client/Server Architecture (Subscription).

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.

5.2.2.2 *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 $n \prec_S^N s$, the server appends to Q all the immediate predecessors of s that have not yet been visited. Otherwise, if $n \not\prec_S^N s$, 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.

5.2.2.3 *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 $\prec_S^S$ is used to fill the queue instead of $\prec_S^N$.

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 predecessors 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 14 continues the scenario from Figure 13c. 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].

5.2.2.4 *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

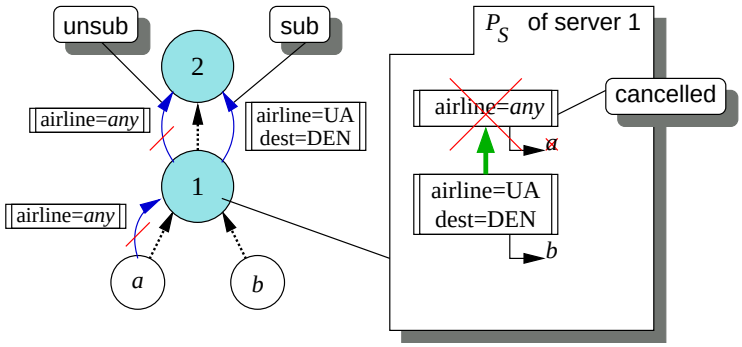

Fig. 14. Example Scenario Using a Hierarchical Client/Server Architecture (Unsubscription).

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.

5.2.3.1 *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. 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 [Carzaniga 1998].

5.2.3.2 *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.

5.2.3.3 *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: $f \prec_S^S f' \wedge X \in \mathit{subscribers}(f')$. 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: $f \prec_S^S f' \wedge f' \prec_S^S f$. Here the server adds X to $\mathit{subscribers}(f')$; or

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

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 $\mathit{forwards}(f)$ be defined as follows:

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

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. $\mathit{NST}(f)$ 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 [Dalal and Metcalfe 1978].

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.

We illustrate the processing of subscriptions in the acyclic peer-to-peer architecture with the scenario depicted in Figure 15. Figure 15a 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 15b 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 predecessor of the previous (root) subscription and is forwarded to server 3, which is the only neighbor that is not in the forwards set for the covering subscription [airline=*any*].

In Figure 15c, 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].

5.2.3.4 *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 26). 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

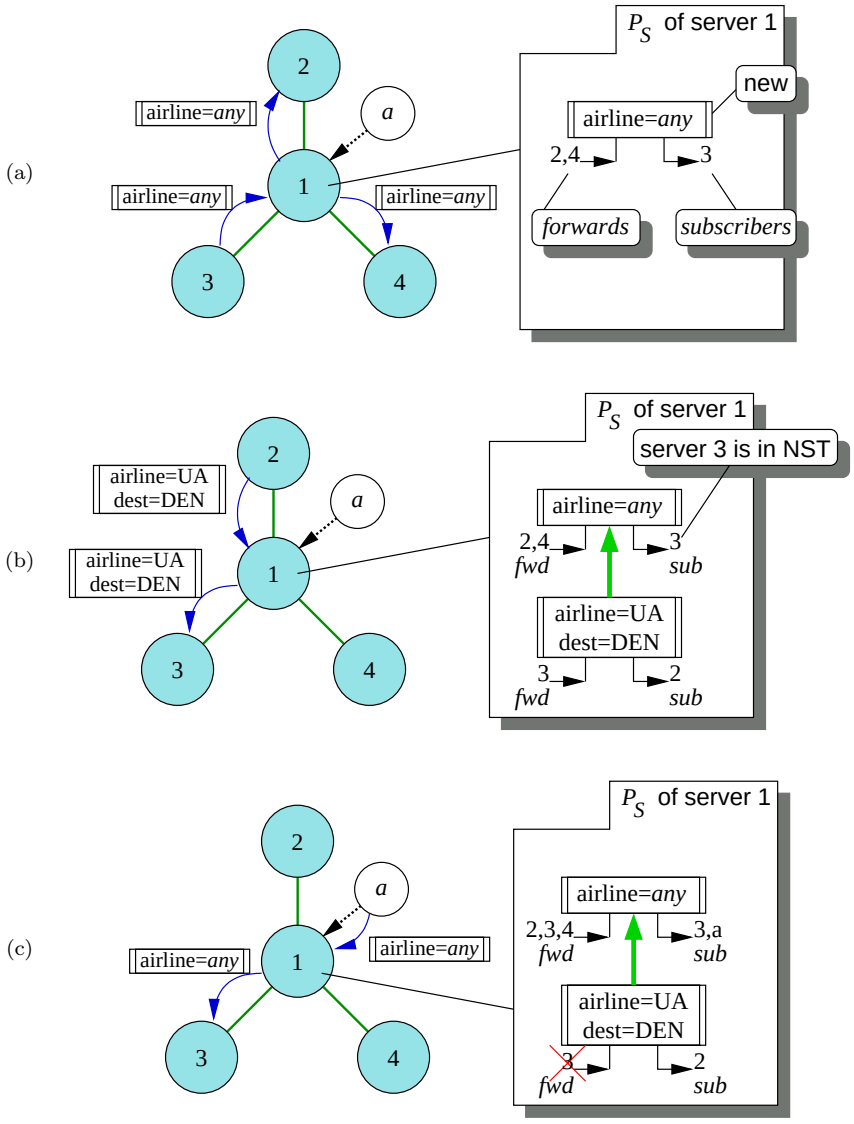

Fig. 15. Example Scenario Using an Acyclic Peer-to-Peer Architecture.

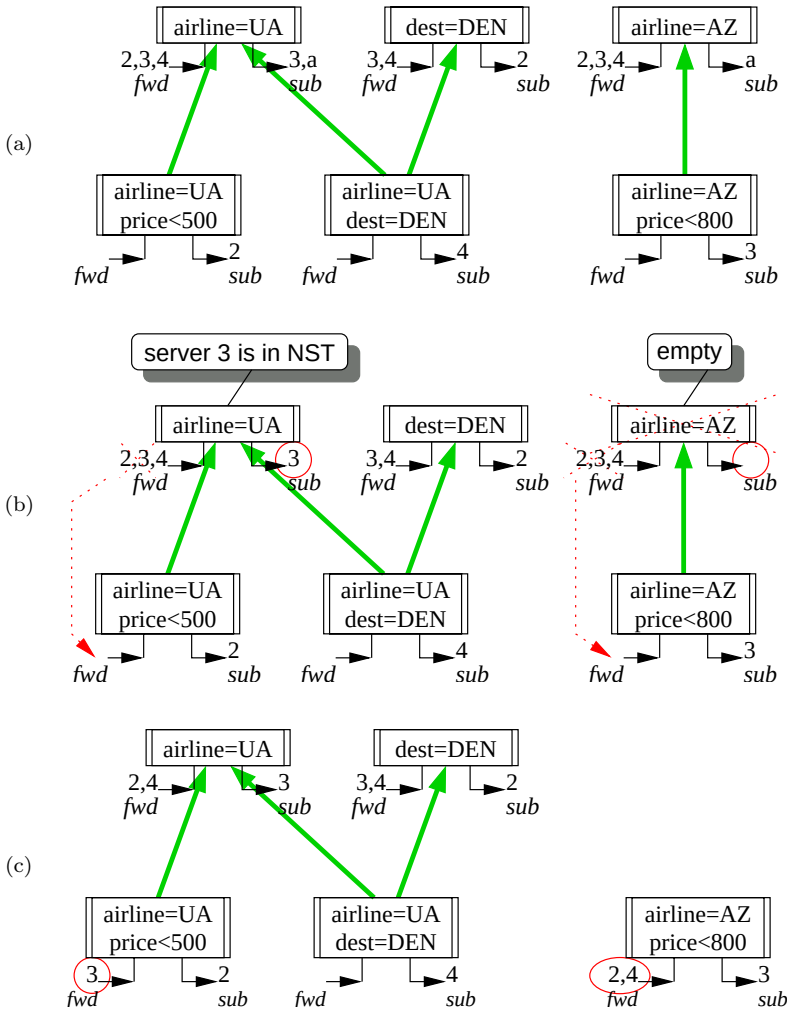

Fig. 16. Unsubscriptions in an Acyclic Peer-to-Peer Architecture.

generic subscriptions.

We illustrate the processing of unsubscriptions in the acyclic peer-to-peer architecture with the scenario depicted in Figure 16. Figure 16a depicts the subscriptions poset of server 1 from Figure 15c after it has received some subscriptions from local clients and neighbor servers. This is the state of server 1 just before it receives an unsubscription filter [airline=*any*] from client a.

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 16b 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 predecessors of the canceled subscriptions since their forwards might have changed as a consequence of the previous unsubscriptions. Figure 16c 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) successor 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 successors. 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).

5.2.3.5 *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.

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 $\prec_A^A$, as well as a poset of subscriptions P_S. In P_A, each advertisement a has an associated set of identities $\mathit{advertisers}(a)$ and another set of identities $\mathit{forwards}(a)$.

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 $\mathit{neighbors}_s \subseteq \mathit{neighbors}$ that is specific to s. $\mathit{neighbors}_s$ is defined as the set of advertisers listed in P_A for all the advertisements covering s. Formally:

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

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 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*.

5.3.1 *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 4 shows an

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

Table 4. Example of a Table of Available Patterns.

example of a table of available patterns. The table 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.

5.3.2 *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 4, 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,

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 5. Example of a Factored Compound Subscription.

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 5 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.

5.3.3 *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 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 [Carzaniga 1998].

In the example of Table 5, 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

$s_1 =$ *string* alarm = "failed-login"
integer attempts = 1 • *string* alarm = "failed-login"
integer attempts = 2

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

$s_2 =$ *string* file = "/etc/passwd"
string operation = "file-change"

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 17 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

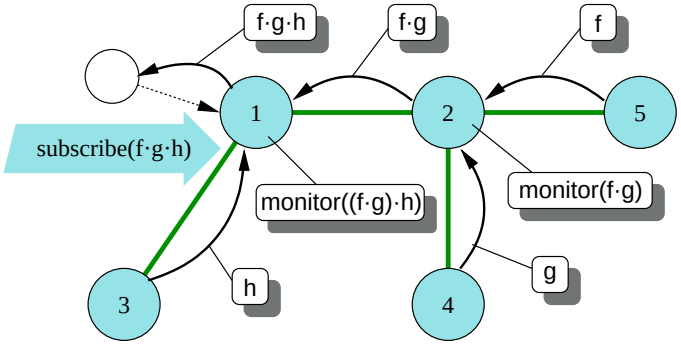

Fig. 17. Pattern Monitoring and Delegation.

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.

Our chosen level of expressiveness in *Siena* represents a compromise, at which notification structure, attribute types, and attribute operators approximate those of the well-understood and widely-used database query language SQL.

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 $\prec_S^N$ is $O(n+m)$, where n is the number of attribute constraints in the subscrip-

tion filter and m is the number of attributes in the notification. The complexity of computing $\prec_S^N$ 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 $\prec_S^S$, $\prec_A^A$, and $\prec_A^S$ 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 $[\mathrm{x} > k_1]$ covers $[\mathrm{x} > k_2]$ we can simply verify that $k_2 \geq k_1$.

We also restrict the expressiveness of patterns in SIENA in the interests of efficiency. Patterns, as we discuss in Section 3.3, 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 $\prec_S^N$ 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 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

There are many questions that one could ask about a wide-area event notification service. In our initial simulation 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. A *site* represents a subnet and its possibly many hosts, where the cost of communication between hosts within a site is assumed to be zero. 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.

6.2.1.1 *Network Configuration.* Figure 18 shows the layered structure of a network configuration in our simulation framework. At the bottom level is a model of a

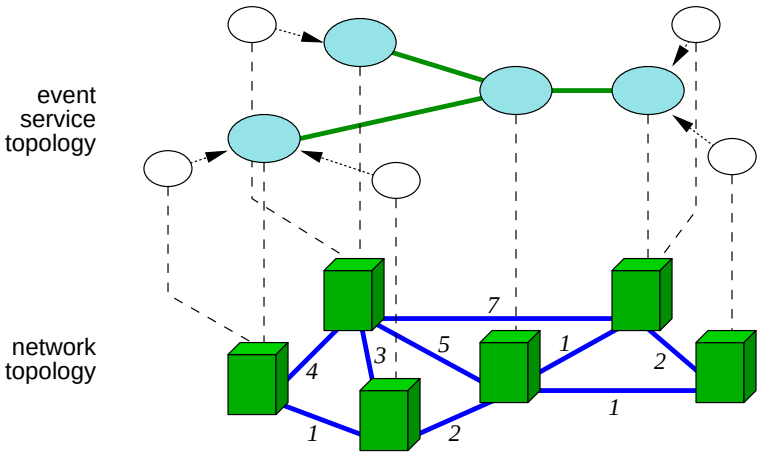

Fig. 18. Layers in a Network Configuration.

wide-area network topology. This model defines *sites* (depicted as cubes) and *links* (depicted as heavy lines between cubes). To develop realistic network topologies that account for important properties such as latency, and approximate the relative costs of real wide-area networks, we use a publicly available generator of random network topologies [2] that implements the Transit-Stub model [Zegura et al. 1996]. (A discussion of this and other models for generating network topologies can be found elsewhere [Zegura et al. 1997].) The results we present here were performed on networks of 100 sites.

We make several simplifying assumptions about sites and links at this level. First, we assume that the costs of computation at sites and communication through links are linear functions of load. Second, links have constant latency. Third, sites and links have infinite capacity. Note that one consequence of these assumptions is that our model does not account for the effect of congestion.

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).

[2] Georgia Tech Internet Topology Models (GT-ITM).

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 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 [Clark 1989].

In addition to simulating the various multi-server architectures, we simulate a single-server, centralized architecture, which serves 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.

6.2.1.2 *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

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

and within each cycle publishes n notifications. In addition, an average delay between publish requests, $\overline{t}$, can be specified (with the delays generated according to a Poisson distribution). Similarly, an interested party executes p sequences

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

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

6.2.1.3 *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.

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.[3] Furthermore, we are only simulating the objects of interest and interested parties associated with one particular kind of event. Because the main purpose of the simulations presented here is to highlight the relative behaviors of the architectures and

[3] A more extensive set of data plots is presented elsewhere [Carzaniga 1998; Carzaniga et al. 1999].

algorithms, we choose not to complicate the experiments by simulating additional kinds of events (with their associated objects of interest and interested parties). While it is conceivable that this choice could affect the results in some way, our intuition tells us that our conclusions about the relative behaviors would remain the same.

To reveal the scaling properties of the architecture/algorithm combinations, 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. Referring to the characterization given in Section 6.2.1, we used parameter values of $m = 10$ and $n = 10$ for objects of interest, indicating 10 sequences of 10 iterations each, and an average inter-publication delay $\overline{t}$ in the interval 2000–2500. The behavior parameters for interested parties was $p = 10$ and $q = 10$, indicating 10 iterations of 10 received notifications each.

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. Except for the plots of the acyclic peer-to-peer architecture with advertisement forwarding (whose behavior is unstable and as yet inexplicable), the variance within each set of 10 simulation runs was negligible and, therefore, we choose not to include error bars in the plots.

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 plots 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.

6.2.2.1 *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. The total cost captures an important aspect of scalability by revealing how communication cost is impacted by increases to the load presented to the service. Figure 19 compares the total costs incurred by the three distributed architectures with 1, 10, 100 and 1000 objects interest; we omit the curves for the centralized architecture because its total cost far outweighs that of the distributed architectures, exhibiting exponential blowup starting at around 1000 interested parties in each plot. There are several interesting observations we can make about these plots.

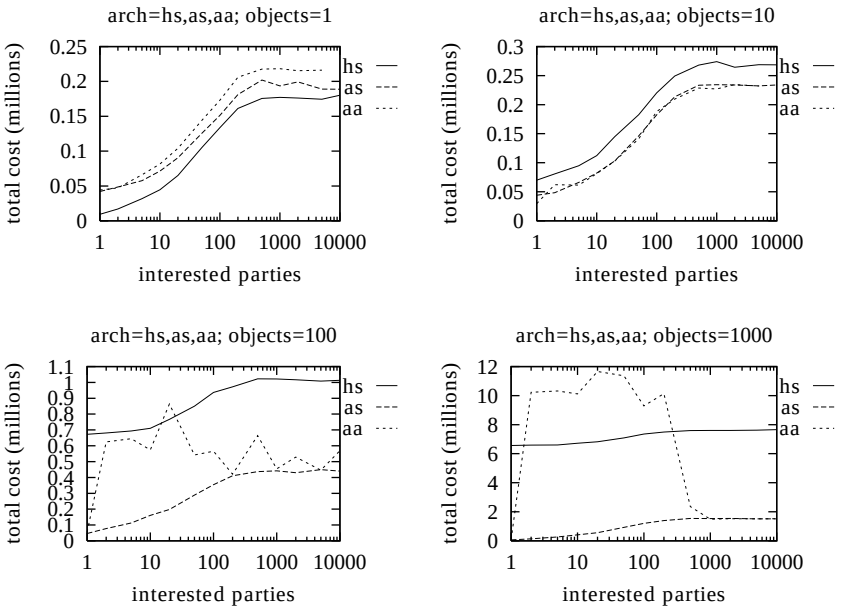

Fig. 19. Comparison of Total Costs Incurred by Distributed Architectures.

First, when there are more than 100 interested parties, the total cost is essentially constant, meaning that there is a point beyond which there is no additional cost incurred in delivering notifications to additional interested parties. 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.)

Second, all the architectures scale sublinearly when the number of interested parties is below the saturation point. When below this point, it is likely that an object of interest and an interested party are not at the same site. Therefore, it is also likely that the cost to deliver a notification is nonzero. However, as the number of interested parties grows, the likelihood increases that each new interested party will be located at a site at which there is already an interested party. The marginal cost of each additional interested party decreases up to the saturation point, where the cost becomes zero. This sublinear scaling character of the architectures can be discerned more easily in the plots of Figure 20, which present the portion of the data of Figure 19 below the saturation point using a linear scale in the horizontal axis.

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 notifications only towards the root of the hierarchy, is forced to do so whether or not interested parties exist on the other

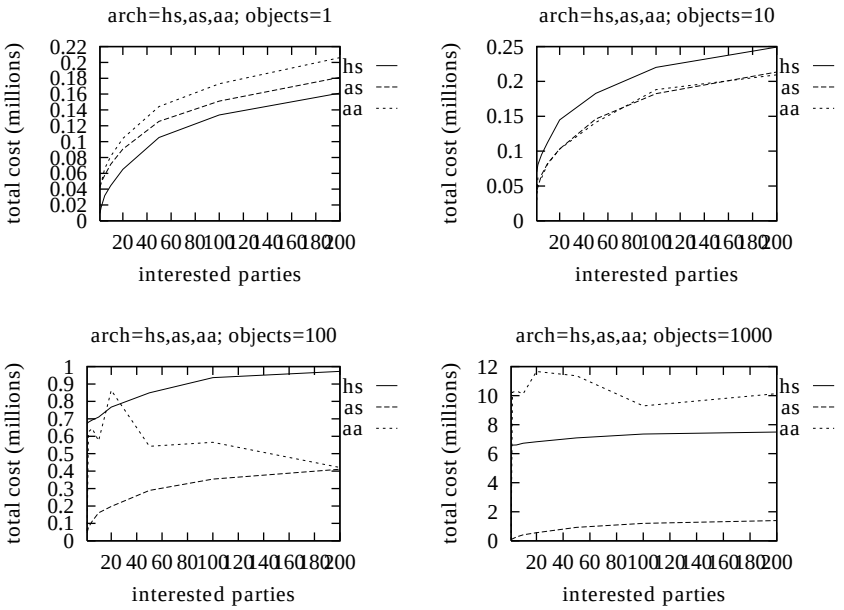

Fig. 20. Comparison of Total Costs Below the Saturation Point.

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, and as yet inexplicable, 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.

Overall, the acyclic peer-to-peer architecture with subscription forwarding 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.

6.2.2.2 *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 21. First, the centralized architecture is unreasonable in essentially all

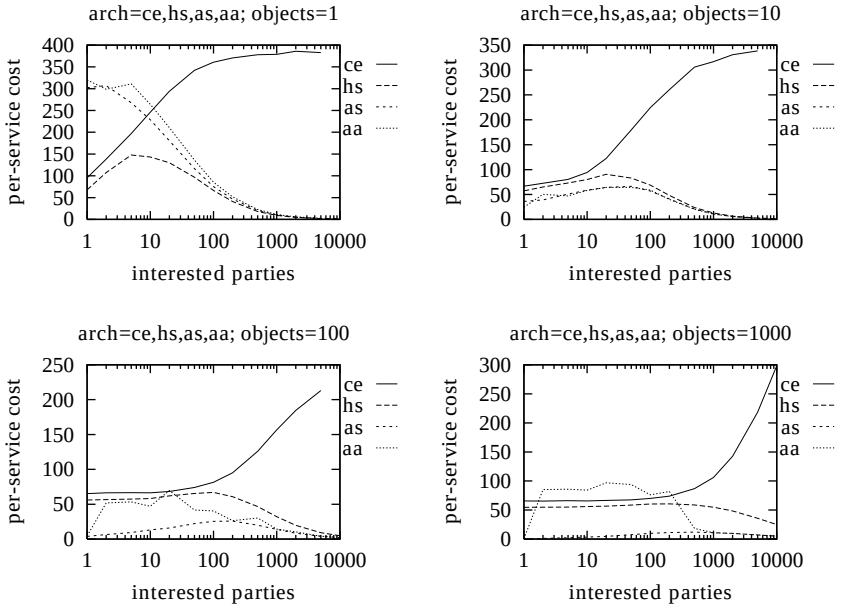

Fig. 21. Comparison of Per-Service Costs.

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.

6.2.2.3 *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 22 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 22 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 $O(\log N)$.

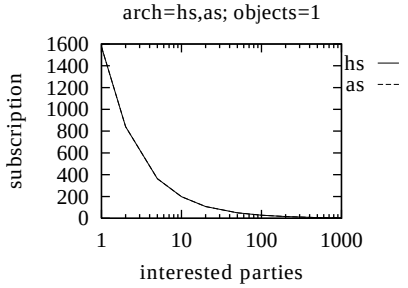

Fig. 22. Comparison of Per-Subscription Costs For Hierarchical Client/Server and Acyclic Peer-to-Peer Architectures under Varying Numbers of Interested Parties.

The acyclic peer-to-peer architecture recoups its greater setup costs for subscriptions by reducing the average cost of notifications. Figure 23 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 23, 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 21.

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

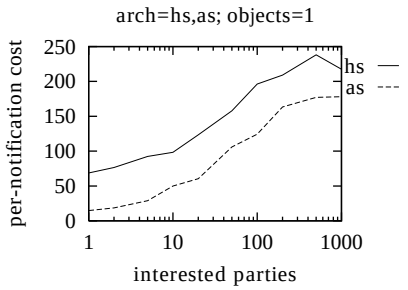

Fig. 23. Comparison of Per-Notification Costs For Hierarchical Client/Server and Acyclic Peer-to-Peer Architectures under Varying Numbers of Interested Parties.

interested parties. For example, in the scenario represented by the results presented here, 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 24, 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.

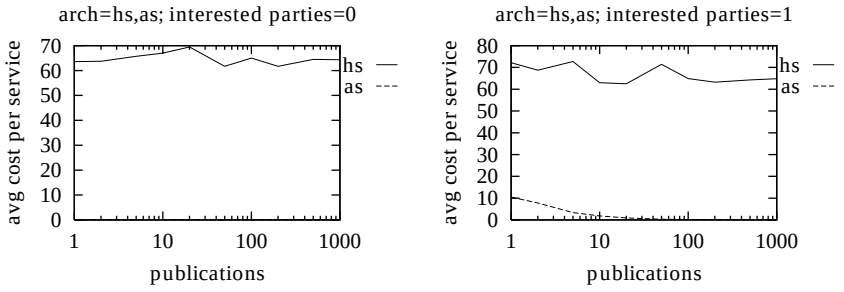

Fig. 24. Comparison of Per-Notification Costs For Hierarchical Client/Server and Acyclic Peer-to-Peer Architectures under Varying Numbers of Publications in Two Degenerate Cases.

6.2.2.4 *Worst-Case Per-Site Cost.* A critical issue in a communication network is how it behaves in the face of congestion. In the simulation model presented here, however, we cannot directly simulate congestion (see Section 6.2.1). Nevertheless, we can ask a related question that gives us an indication of which architecture would approach a given level of congestion soonest. The metric we use is the worst-case per-site cost. It is calculated, for each scenario, by averaging the cost of communication incurred by each site over the ten simulations of that scenario, and then by computing the maximum over those average per-site costs.

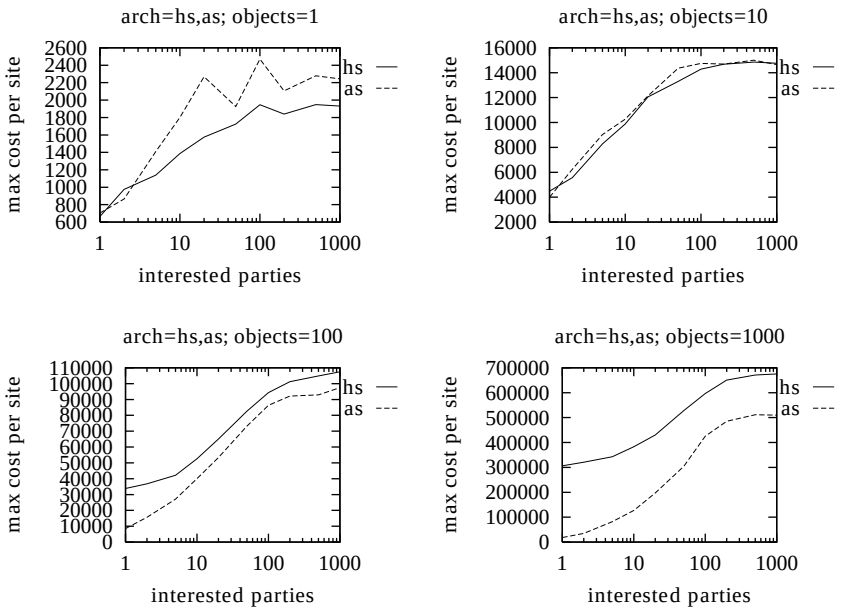

Fig. 25. Comparison of Worst-Case Per-Site Costs.

The plots of Figure 25 show the maximum cost incurred by a site in the hierarchical client/server and acyclic peer-to-peer architectures under the same scenarios of Figure 19 and Figure 20. The hierarchical client/server architecture exhibits slightly lower worst-case per-site cost under low densities of objects of interest. For high densities of objects of interests, and therefore under high volumes of notifications, the hierarchical client/server architecture incurs a much greater maximum per-site cost than the acyclic peer-to-peer architecture. In other words, a high volume of notifications is likely to cause congestion to be reached sooner in the hierarchical client/server archtitecture than in the acyclic peer-to-peer architecture.

How do we explain this difference in behavior? We can see that it is not attributable to a simple overloading of the root server in the hierarchical client/server architecture. This, perhaps counterintuitive, observation is based on the fact that the acyclic peer-to-peer architecture gains advantage only when, for a given object of interest, there is no interested party on the "opposite side" of the network, and hence the traffic remains within a localized neighborhood of the object of interest.

This situation rarely arises under high densities of interested parties. Thus, the difference in behavior is explained solely by the effect of relatively high numbers of ignored notifications unavoidably forwarded to the root server, as discussed above.

6.2.3 *Summary.* 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 per-subscription cost than the acyclic peer-to-peer ($O(\log N)$ and $O(N)$ respectively). 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.

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 [Carzaniga 1998; Carzaniga et al. 1999].

[4]Source and binary packages are available at `http://www.cs.colorado.edu/serl/siena/`.

7.1 Classification Framework

In order to understand and classify technologies that are related to SIENA, we can compare them from the perspective of their server architectures, which affects scalability, and from the perspective of their subscription language, which affects expressiveness. Table 6 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 7.

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. We note that user-defined operators suffer from the disadvantage of having arbitrary, unknown, and potentially unbounded complexity. Moreover, the computation of the covering relations that allow the pruning of propagation trees, such as $\prec_S^S$, might be undecidable.

From Table 7 we derive the four classes of subscription languages used in Table 6. 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 6 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 6 in greater detail.

7.2 Related Technologies

The idea of integrating different components by means of *messages* was pioneered in a research system called Field [Reiss 1990]. As in several commercial products that followed (e.g., HP SoftBench [Cagan 1990], DEC FUSE [Hart and Lupton 1995], and Sun ToolTalk [Julienne and Holtz 1994]), Field implements a *message-based integrated environment* in which several software development tools can cooperate by

		Architecture		
		centralized	*hierarchical client/server*	*peer-to-peer*
Subscription Language	*channel-based*	Field [Reiss 1990] CORBA Event Service [Object Management Group 1998a] Java Distributed Event Specification [Sun Microsystems, Inc. 1998] Java Message Service [Sun Microsystems, Inc. 1999]	CORBA Event Service [Object Management Group 1998a]	IP multicast [Deering and Cheriton 1990] SoftWired's iBus
	subject-based	ToolTalk [Julienne and Holtz 1994]	NNTP [Kantor and Lapsley 1986] JEDI [Cugola et al. pear] TIBCO's TIB/Rendezvous	Talarian's SmartSockets Vitria's BusinessWare
	content-based	Elvin [Segall and Arnold 1997]	Keryx [Wray and Hawkes 1998] Yu et al. [Yu et al. 1999]	Gryphon [Banavar et al. 1999]
	content-based with patterns	GEM [Mansouri-Samani and Sloman 1997] Yeast [Krishnamurthy and Rosenblum 1995] CORBA Notification Service [Object Management Group 1998b] object-oriented active databases [Ceri and Widom 1996]	*Siena*	*Siena*

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

Table 6. A Classification of Related Technologies.

		Scope		
		single notification one 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 7. Typical Features of Subscription Languages.

exchanging messages. Messages carry service requests to other tools or announcements of changes of state. 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.

Yeast [Krishnamurthy and Rosenblum 1995] is an *event-action 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 with a rich event pattern language. The action part of a rule is a UNIX shell script. GEM [Mansouri-Samani and Sloman 1997] 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 [Ceri and Widom 1996]. 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, so that responses to events (i.e., the actions) are executed by interested parties externally to the service. An event-action system or an active database, on the other hand, is responsible for also executing the actions taken in response to event notifications.

The USENET News system, with its main protocol NNTP [Kantor and Lapsley 1986], is perhaps the best example of a scalable, user-level, many-to-many communication facility. News articles (modeled after e-mail messages) are propagated through a network of servers. A new server can join the network by connecting as a *slave* to another (*master*) server that is already part of the infrastructure. Articles are posted to *newsgroups*, which are organized in a hierarchical name/subject space. NNTP provides a primitive filtering capability, such that articles can be selected by means of simple expressions denoting sets of group names and 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. Although group names and subnames 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. As a result, it is common for news readers (the client programs of USENET News) to allow users to perform additional, more sophisticated filtering over the messages that have been transferred from the server, but

this is outside the purview of the service itself and thus cannot be used to reduce network traffic. The service is scalable but still quite heavyweight, and in fact the time frame of article propagation ranges from hours to days, which is inadequate for an event notification service.

IP multicast [Deering 1991] is a network-level infrastructure that extends the Internet protocol to realize a one-to-many communication service. The network that realizes this extension is also referred to as the MBone. A multicast address is a virtual IP address that corresponds to a group of hosts. 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 reporting their group membership with a specific protocol [Fenner 1997]. 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, and in which subscribing is equivalent to joining a group. However, the IP multicast infrastructure has major limitations when used as a generic event notification service. The first issue is how to map expressions of interest into IP group addresses in a scalable way. A separate service, perhaps similar to DNS, could be sufficient to resolve the mapping. However, we would have to assume that there exist enough multicast addresses to map every possible expression of interest. The second and most crucial issue is the limited expressiveness of the addressing scheme itself. In fact, since IP multicast never relates two different IP groups, it would not be possible to exploit similarities between subscriptions mapped to different IP group addresses. Different notifications matching more than one subscription would have to be sent to several separate multicast addresses, each one being routed in parallel and independently by the IP multicast network.

Another network-level technology that is at least somewhat related to *Siena* is *active networks*. An active network is a network with programmable switches [Tennenhouse et al. 1997]. 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.

Two prominent efforts are intended to lead specifically to event notification services. These are the CORBA Notification Service [Object Management Group 1999] and the Java™ Message Service [Sun Microsystems, Inc. 1999]. It is important to note, however, that both these efforts do not themselves realize an event notification service. Rather, they simply specify interfaces to be implemented by a service, with the critical issue of how an implementation is to provide a scalable service left unaddressed.

Commercial products such as SoftWired's iBus, TIBCO's TIB/Rendezvous™, Talarian's SmartSockets™, Hewlett-Packard's E-speak™, and the messaging system in Vitria's BusinessWare™ provide implementations of an event notification service. While these products support distribution of the service, they are not specifically designed to support wide-area scale to the degree discussed in this paper. In particular, they typically achieve simple distribution through a federation of centralized servers with statically configured inter-server routing.

There are several research efforts concerned with the development of an event notification service, including IBM's Gryphon [Aguilera et al. 1999; Banavar et al. 1999], Elvin [Segall and Arnold 1997], JEDI [Cugola et al. 1998], Keryx [Wray and Hawkes 1998], and the recent work of Yu et al. [Yu et al. 1999].

Gryphon is a distributed content-based message brokering system similar to *S*IENA. The techniques developed in Gryphon are complementary to the ones of *S*IENA. In particular, Gryphon uses a fast algorithm to match a notification to a large set of subscriptions [Aguilera et al. 1999]. This algorithm, similar to the one described by Gough and Smith [Gough and Smith 1995], 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 *S*IENA is that Gryphon propagates every subscription everywhere in the network, whereas *S*IENA 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 *S*IENA'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 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 that adopts channel-based addressing.

The same channel-based addressing is specified by the CORBA Event Service [Object Management Group 1998a] and by the Java™ Distributed Event Specification [Sun Microsystems, Inc. 1998]. The shortcomings of the channel-based addressing scheme have been recognized both in the CORBA and Java communities, therefore these specifications have recently been superseded by more advanced service specifications that are in line with the interface of *S*IENA. In particular, the OMG has added additional filtering based on notification contents with the specification of the CORBA Notification Service [Object Management Group 1999], while Sun has specified a radically new service, the Java™ Message Service [Sun Microsystems, Inc. 1999], that features SQL-like message selectors.

8. CONCLUSIONS

In this paper, we have described our work on *S*IENA, 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. To summarize, we found that the hierarchical architecture is suitable with low densities of clients that subscribe (and unsubscribe) very frequently, whereas the peer-to-peer architecture performs better when the total cost of communication is dominated by notifications. In situations where there are high numbers of ignored notifications (i.e., notifications for which there are no subscribers), the peer-to-peer architecture is also superior to the hierarchical architecture. We plan to continue exploring 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. Additionally, 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. Finally, we plan to explore other important aspects of a wide-area event notification service. Specifically, we are currently developing a model of secure publish/subscribe communication, as well as mechanisms for reliability and fault tolerance of event notification services.

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 [Carzaniga et al. 2000b]. 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 18 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 event notification services, supported by a new class of networks built from high-speed content-based routers.

Acknowledgments

We 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

Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M., and Chandra, T. D. 1999. Matching events in a content-based subscription system. In *Eighteenth ACM Symposium on Principles of Distributed Computing (PODC '99)* (Atlanta, GA, May 4–6 1999), pp. 53–61.

Banavar, G., Chandra, T. D., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. 1999. An efficient multicast protocol for content-based publish-subscribe

systems. In *The 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)* (Austin, TX, May 1999), pp. 262–272.

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

Cagan, M. R. 1990. 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 (June), 36–47.

Carzaniga, A. 1998. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. Ph. D. thesis, Politecnico di Milano, Milano, Italy.

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 1999. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99 (Oct.), Department of Computer Science, University of Colorado. Revised May 2000.

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. 2000a. 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), pp. 219–227.

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

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

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

Cugola, G., Di Nitto, E., and Fuggetta, A. 1998. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)* (Kyoto, Japan, April 1998), pp. 261–270.

Cugola, G., Di Nitto, E., and Fuggetta, A. To appear. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*.

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

Deering, S. E. 1991. *Multicast Routing in a Datagram Internetwork*. Ph. D. thesis, Stanford University.

Deering, S. E. and Cheriton, D. R. 1990. Multicast routing in datagram networks and extended LANs. *ACM Transactions on Computer Systems 8*, 2 (May), 85–111.

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

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

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

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

Kantor, B. and Lapsley, P. 1986. Network news transfer protocol—a proposed standard for the stream-based transmission of news. Internet Requests For Comments (RFC) 977.

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

Mansouri-Samani, M. and Sloman, M. 1997. GEM: A generalized event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal 4*, 2 (June), 96–108.

Object Management Group. 1998a. CORBAservices: Common object service specification. Technical report (July), Object Management Group.

Object Management Group. 1998b. Notification service. Technical report (Nov.), Object Management Group.

Object Management Group. 1999. Notification service. Technical report (Aug.), Object Management Group.

Reiss, S. P. 1990. Connecting tools using message passing in the Field environment. *IEEE Software 7*, 4 (July), 57–66.

Rosenblum, D. S. and Wolf, A. L. 1997. 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 (1997), pp. 344–360. Springer–Verlag.

Segall, B. and Arnold, D. 1997. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97* (Brisbane, Australia, Sept. 3–5 1997), pp. 243–255.

Sun Microsystems, Inc. 1998. *Java Distributed Event Specification*. Mountain View, CA: Sun Microsystems, Inc.

Sun Microsystems, Inc. 1999. *Java Message Service*. Mountain View, CA: Sun Microsystems, Inc.

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

Wray, M. and Hawkes, R. 1998. Distributed virtual environments and VRML: an event-based architecture. *Computer Networks and ISDN Systems 1-7*, 30 (April), 43–51.

Yu, H., Estrin, D., and Govindan, R. 1999. A hierarchical proxy architecture for Internet-scale event services. In *Proceedings of WETICE '99* (Stanford, CA, June 1999).

Zegura, E. W., Calvert, K., and Donahoo, M. J. 1997. A quantitative comparison of graph-based models for internet topology. *IEEE/ACM Transactions on Networking 5*, 6 (Dec.), 770–783.

Zegura, E. W., Calvert, K. L., and Bhattacharjee, S. 1996. How to model an internetwork. In *Proceedings of IEEE INFOCOM '96* (San Francisco, CA, April 1996), pp. 594–602.