Design of a Scalable Event Notification Service: Interface and Architecture

Antonio Carzaniga
Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430, USA
+1 303 492 4463
carzanig@cs.colorado.edu

David S. Rosenblum
Dept. of Inf. and Computer Science
University of California
Irvine, CA 92697-3425, USA
+1 949 824 6534
dsr@ics.uci.edu

Alexander L. Wolf
Dept. of Computer Science
University of Colorado
Boulder, CO 80309-0430, USA
+1 303 492 5263
alw@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-863-98
August 1998

Abstract

Event-based distributed systems are programmed to operate in response to events. An event notification service is an application-independent infrastructure that supports the construction of event-based systems. While numerous technologies have been developed for supporting event-based interactions over local-area networks, these technologies do not scale well to wide-area networks such as the Internet. Wide-area networks pose new challenges that have to be attacked with solutions that specifically address issues of scalability. This paper presents Siena, a scalable event notification service that is based on a distributed architecture of event servers. We first present a formally defined interface that is based on an extension to the publish/subscribe protocol. We then describe and compare several different server topologies and routing algorithms. We conclude by briefly discussing related work, our experience with an initial implementation of Siena, and a framework for evaluating the scalability of event notification services such as Siena.

1 Introduction

The *event-based* architectural style is well established and widely used. Several classes of applications adopt an event-based architecture, including integrated development environments, workflow and process support systems, software deployment systems, graphical user interfaces, network management tools, and security monitors. In this style, components are programmed as reactive objects that perform actions in response to certain events. Such style is particularly suitable for applications that are reactive by nature, such as network and security monitors, and also for systems that integrate heterogeneous components and thus require loosely coupled interaction.

The connectivity provided by wide-area networks such as the Internet offers even stronger motivation for using an event-based architecture. New applications can be designed that take advantage of the vast number of information sources available on-line. Examples are stock market analysis tools and data mining and indexing tools. Also, in the context of a wide-area network, existing applications can be integrated at a much higher scale; for example, workflow systems can be federated for companies that have multiple distributed development sites or even across corporate boundaries.

The common infrastructure underlying event-based systems is the *event service*. An event service is a general-purpose facility that provides for *observation* and *notification* of events among distributed objects. Numerous technologies that realize an event service have been developed and effectively used for quite a long time. However, most of them target local-area networks. Extending the support of an event service to a wide-area network creates new challenges and trade-offs. Not only does the number of objects and events grow tremendously, but also many of the assumptions made for local-area networks, such as, low latency, abundant bandwidth, homogeneous platforms, continuous reliable connectivity, and centralized control, are no longer valid.

Some technologies address issues related to wide-area services. Among them, are new technologies such as Tibco [8] that specifically provide an event service, but also, more mature technologies such as the USENET news infrastructure, IP multicasting, the Domain Name Service (DNS), that, although not explicitly targeted at this problem domain, represent potential or partial solutions. The main problem with all of these technologies is that they are either specific to some application domain or not flexible enough to be usable as a generic infrastructure for event-based applications.

This paper presents Siena, a project directed towards the design and implementation of a scalable general-purpose event service. The contributions of this work are a formal definition of an event service that combines expressiveness with scalability together with the design and implementation of the architectures and algorithms that realize this event service as a distributed infrastructure. One obvious issue that we must face in this research is the evaluation of the solutions that we propose. To this end, we have performed systematic simulations of our architectures and algorithms in several network scenarios.

The following section gives the basics of the Siena event service. The paper then continues in Section 3 with a formal definition of the interface and the semantics of the event service. The architectures and algorithms that realize the service are presented in Section 4. Section 5 provides an overview of some related systems and research topics. Our evaluation effort and our experience with a prototype is presented in Section 6. We then conclude in Section 7 with some directions for future work and additional analysis and evaluation.

2 Event Service

An *event service* is a dispatcher of event notifications. Applications that use the event service can be *interested parties*, i.e., event consumers, or *objects of interest*, i.e., event generators, or both. The dispatching is regulated by advertisements, subscriptions, and publications.

Figure 1 shows the high-level architecture of an

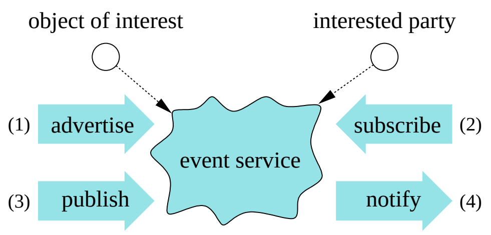

Figure 1: Event service

event service. Informally, objects of interest specify the events they intend to publish by means of *advertisements* (1), while interested parties specify the events they are interested in by means of *subscriptions* (2). Objects of interest can then publish *notifications* (3), and the event service will take care of delivering the notifications to the interested parties that subscribed for them (4). The terms used in this paper, in particular the terms *notification*, *object of interest*, and *interested party*, follow the framework proposed in [13].

Without loss of generality, we will always assume that objects of interest are "active", i.e., they autonomously publish event notifications. Passive objects, such as files, can participate in an event-based interaction by means of other active objects that act as *proxies* and that notify events on behalf of the passive objects. This distinction is similar to the one made in JEDI [2]. In any case, the passive object will not be considered in the models.

2.1 Event Servers

The event service can be realized by connecting many *events servers*. An application contacts the event service via one event server also referred as its *access point*. (see Figure 2).

2.2 Identifiers and Handlers

In order for interested parties, objects of interest, and event servers to communicate, a *naming* scheme must be adopted whereby objects can be uniquely identified. A *handling* scheme must also be adopted so that objects can be contacted using

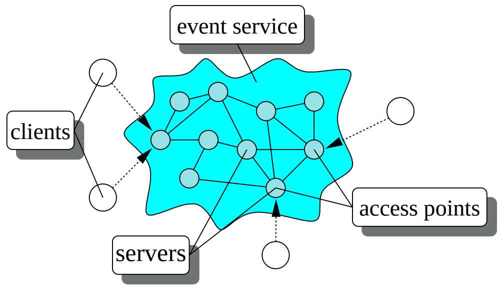

Figure 2: Internal architecture of the event service

appropriate communication protocols.

The Siena event service adopts the generic URI [1] form for both its naming and handling scheme. This means that every object has a URI associated with it that defines both the *identity* of that object and the *handler* used by the event service to deliver a notification to that object. For example, if the URI *mailto:carzanig@cs.colorado.edu* identifies an object, then *mailto:carzanig@cs.colorado.edu* is both the unique name of that object and the method that the event service uses to communicate with that object. In this case, in order to send a notification to that object, the event service will send an e-mail message to *carzanig@cs.colorado.edu*.

The event service recognizes the most common URI schemas, including *mailto* and *http*, and thus implements the communication protocols implied by each schema. The implementation of the event service defines and maintains the URIs corresponding to event servers, however it does not directly assign or maintain URIs for interested parties or objects of interest. Such URIs are provided and operated by clients themselves. This means that if a client identifies itself as *mailto:carzanig@cs.colorado.edu*, then the event service will simply assume that the mailbox *carzanig@cs.colorado.edu* exists and is directly accessible.

3 Interface and Semantics of the Event Service

The Siena event service exports the following main functions:

publish(notification n)
subscribe(URI subscriber, pattern p)
unsubscribe(URI subscriber, pattern p)
advertise(URI publisher, filter f)
unadvertise(URI publisher, filter f)

In the following subsections we present the syntax and the semantics of these functions by formally defining *notifications*, *filters*, and *patterns* and their role in every function. We then present a formal definition of the semantics of the event service showing how it can affect scalability.

3.1 Notifications, Filters, and Patterns

An event notification is a set of attributes in which each attribute is a triple: *attribute* = (*name*, *type*, *value*). For example, the notification displayed in Figure 3 represents a stock price variation event.

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

Figure 3: Example of a notification

In an event notification, attributes are uniquely identified by their name. Attribute types belong to a pre-defined set of types. A fixed set of operators is also defined. Types and operators are an integral part of the event service definition. We do not a give a precise definition for the types and operators here, but instead simply assume those defined in modern programming languages. If α is an attribute of a notification, $\alpha.name$, $\alpha.type$, and $\alpha.value$ denotes its name, type, and value respectively.

3.2 Filters

An *event filter*, or simply a *filter*, defines a class of event notifications by specifying a set of attribute names and types and some constraints on their values.

string	*event*	*=	*finance/exchanges/**
string	*exchange*	==	*NYSE*
string	*symbol*	==	*DIS*
float	*change*	$<$	*0*

Figure 4: Example of an event filter

Figure 4 shows a filter that selects negative stock price variations for a specific stock on a specific exchange. More formally, a filter is made of a set of *attribute filters*. Each attribute filter specifies a name, a type, a boolean binary operator, and a value for an attribute: *attr-filter* = (*name*, *type*, *operator*, *value*). In an event filter, there can be more than one attribute filter with the same name. For an attribute filter α, $\alpha.match\_op(operand_1, operand_2)$ denotes the application of the operator defined by α to $operand_1$ and $operand_2$.

3.3 Patterns

A *pattern* of events is defined by combining a set of event filters using filter *combinators*.

string	*event*	*=	*finance/exchanges/**
string	*symbol*	==	*MSFT*
float	*change*	$<$	*0*

and then

string	*event*	*=	*finance/exchanges/**
string	*symbol*	==	*NSCP*
float	*change*	$>$	*0*

Figure 5: Example of a pattern of events

An example of a pattern that combines two filters into a sequence is shown in Figure 5. More formally, an event filter is itself a pattern, and any two patterns can be combined to form another pattern by means of a combinator. Intuitively, while a filter selects one event notification at a time, a pattern can select several notifications that *together* match an *algebraic combination* of filters.

We say that a pattern is *simple* when it contains only one event filter. Also, since subscriptions submit patterns to the event service, we say that a subscription is *simple* when it requests a simple pattern or *compound* when it requests a pattern with two or more filters.

For the purpose of this paper, we will only discuss the *and then* or *sequence* combinator that construct patterns matching a temporal sequence of events.

3.4 Compatibility Relations

In order to give the precise semantics of the event service, we must introduce and define the concept of *compatibility* between notifications and subscriptions, and between subscriptions and advertisements. The compatibility between notifications and subscriptions defines the semantics of subscriptions and comes into play because the main job of the event service is to decide whether or not notifications that are published match any subscription submitted by an interested party. In case a notification matches some subscriptions, the event service routes the notification towards all the interested parties that posted such subscriptions. The compatibility between subscriptions and advertisements is also important because, in setting up the routing information, the event service takes advertisements into account to see if they are relevant to any subscription. The compatibility between subscriptions and advertisements subsumes a relation between notifications and advertisements that defines the semantics of advertisements.

The following sections define what it means for a notification to be compatible with a subscription and for a subscription to be compatible with an advertisement. Initially we consider only simple subscriptions (i.e., event filters) and then extend the compatibility relations to compound subscriptions.

3.4.1 Notifications vs. Subscriptions

Let $\mathcal{N}$ be the domain of notifications and $\mathcal{S}_0$ the set of all the simple subscriptions. We define the following binary relation:

$$IsCompatible_N^S \subseteq \mathcal{N} \times \mathcal{S}_0$$

For brevity, we represent the relation $IsCompatible_N^S$ with the symbol '$\sqsubset_N^S$'. When a notification n is compatible with a subscription s, we also say that s *covers* n, and we denote with $N(s) \subseteq \mathcal{N}$ the set of notifications n covered by s.

We define the semantics of $\sqsubset_N^S$ by defining $N(s)$ as follows:

$$\begin{aligned} N(s) = \{n \in \mathcal{N} : \forall \alpha_s \in s : \exists \alpha_n \in n : \\ \alpha_s.name = \alpha_n.name \wedge \alpha_s.type = \alpha_n.type \\ \wedge\ \alpha_s.match\_op(\alpha_n.value, \alpha_s.value)\} \end{aligned} \quad (1)$$

This mandates that all attributes in the subscription appear by name in the notification and that they match by type and value. The notification can also contain other attributes that are not specified in the subscription.

3.4.2 Subscriptions vs. Advertisements

We first define the semantics of advertisements similarly to what we have done in the previous section for subscriptions. Let $\mathcal{A}$ be the domain of advertisements and $a \in \mathcal{A}$ an advertisement. We define the set of notification defined (or *covered*) by a:

$$\begin{aligned} N(a) = \{n \in \mathcal{N} : \forall \alpha_n \in n : \exists \alpha_a \in a : \\ \alpha_n.name = \alpha_a.name \wedge \alpha_n.type = \alpha_a.type \\ \wedge\ \alpha_a.match\_op(\alpha_n.value, \alpha_a.value)\} \end{aligned} \quad (2)$$

This says that an advertisement covers all the notifications that have a set of attributes *included* (present by name and matching by value) in the set of attributes of the advertisement.

Given the definition of $N(a)$ we can easily define $IsCompatible^A_S$ ($\sqsubset^A_S$ for short), the compatibility relation between subscription and advertisements:

$$\sqsubset^A_S \subseteq \mathcal{S}_0 \times \mathcal{A}$$

Intuitively, the compatibility between a subscription s and an advertisement a corresponds to the relation between the two sets of notifications defined by s and a respectively, thus:

$$s \sqsubset^A_S a \Leftrightarrow N(a) \cap N(s) \neq \emptyset \qquad (3)$$

This says that a subscription s is compatible with an advertisement a whenever the set of notifications defined by a, $N(a)$, includes one or more notifications that are also covered by s. When a subscription s is compatible with an advertisement a, we also say that *a covers s*.

3.5 Semantics of the Service

In this section we discuss the behavior of the event service in response to advertisements, subscriptions, and notifications. We have studied and implemented two alternative semantics:

- *subscription-based*, and
- *advertisement-based*.

These two behaviors define two *different* event services. The reason to present both and not to make a definite choice here is that these two semantics impose different requirements upon the implementation of the event service, resulting in different architectures with different degrees of scalability. At this point, we do not have enough experience in using the event service to know which one is more suitable, flexible, and scalable. It might also make sense to provide both of them and let the user choose which one works best for each particular situation.

3.5.1 Subscription-based Event Service

In the *subscription-based* event service, only subscriptions determine the semantics of the service. Advertisements *may* be used by the event service (e.g., to optimize the routing of subscriptions), but they are *not required*. The event service will guarantee the delivery of a notification to all interested parties that have subscribed for it. Referring to the compatibility relation between notifications and subscriptions, the event service will deliver a notification n to an interested party X *if and only if*:

1. X subscribes for s; *and*
2. $n \sqsubset^S_N s$.

3.5.2 Advertisement-based Event Service

In the *advertisement-based* event service, *both* advertisements and subscriptions are used. In particular, advertisements are used to make notifications visible to all the participants of the event service. More specifically, the event service will guarantee the delivery of a notification n posted by object Y to interested party X *if and only if*

1. Y advertises a;
2. X subscribes for s;
3. $s \sqsubset^A_S a$; *and*
4. $n \sqsubset^S_N s$.

Note that if an interested party X sends a subscription s' that covers n, but Y has never posted any advertisement a that covers s', then the event service will not guarantee the delivery of n to X.

3.6 Patterns

So far we have discussed the semantics of the event service for *simple* subscriptions, i.e., for subscriptions that are composed of one event filter. However, both the subscription-based and the advertisement-based semantics can be easily extended to incorporate patterns.

As described above, patterns are defined by *pattern filters*, which are expressions whose elementary terms are simple filters. Thus, a subscription to a pattern filter can be logically viewed as a set of separate subscriptions to all the elementary components of that pattern filter plus a monitor that assembles sequences of notifications, each one matching one of the elementary components according to the semantics of the combinators. Thus, the event service will guarantee the delivery of a pattern of notifications matching an event filter only if it can guarantee the delivery of all the elementary components of the filter. Note that, from this definition of the semantics of patterns, the delivered pattern of notifications contains the *first* notification matching each elementary component.

3.7 Comments on the Semantics of the Event Service

The rationale behind the two semantics and their extensions to patterns is to define an event notification service that (1) behaves in an intuitive and useful way, and (2) allows for an efficient and scalable realization. In this paper, we do not explore the domain of applications that would make use of an event service, so we rely on our previous research and experience to justify the first item. Instead, we will elaborate more on the second item by showing how the information provided by advertisements and subscriptions with the given semantics can be effectively used to direct the communication between event servers in an efficient way.

Timing and *quality of service* are important, but they're not covered in details in this paper. Timing issues might arise when considering unsubscriptions and unadvertisements. For example, an interested party may send an unsubscription when some notifications have already been sent to it. In that case, the interested party will probably receive undesired notifications. Other timing issues regarding the ordering of notifications and thus pattern recognition can arise depending on the topology and latency of the network. For the time being we will assume that the event service is able within a finite time to shuffle notifications so that they are sent (and received) in the correct temporal sequence.[1]

By quality of service we refer to a number of non-functional properties that do not directly affect the semantics, but that are nonetheless of fundamental importance for the practical realization and usage of the event service. A number of other interface functions will be added to deal with quality of service settings such as authentication and security, and transactional communications.

3.7.1 Rationale: Expressiveness vs. Scalability

The rationale for our formal definition of notifications, filters, patterns, and compatibility relations goes beyond a clear specification of the semantics of the event service. The realization of the event service by means of distributed event servers, requires to disseminate some information concerning subscriptions and advertisements among event servers in order to control the flow of notifications towards interested parties. In the distribution of this information, the compatibility relations together with other similar relations between filters ($\sqsubset_S^S$ that defines the compatibility of two simple subscriptions and $\sqsubset_A^A$ that works for two advertisements), play a fundamental role. In fact, similarly to the optimization of queries in a database, using the compatibility relations, the event service can optimize the deployment of filter- and pattern-matchers to minimize the usage of communication and computation resources.

Thus, for the practical realization of the event service and for its scalability, it is essential that these relations can be efficiently implemented. The relations that pose significant problems are clearly the ones that involve two filters (e.g., $\sqsubset_S^A$); in fact, computing $n \sqsubset_N^S s$ is just a matter of applying the filter defined by s to n, which involves computing a conjunction of simple predicates evaluated for a

[1]This assumption would require the existence of a global clock, an upper bound for the network latency and the network diameter, and sufficiently big communication buffers. Note that while these latter requirements can pose serious engineering trade-offs, the availability of high-resolution GPS services makes the first assumption very reasonable for most practical applications.

particular instance of their independent variables. On the other hand, comparing two filters, to verify $s \sqsubset_S^A a$ is equivalent to verifying the implication between two expressions of predicates for every possible notification.

Even in our particular case in which filter expressions are conjunctions of simple predicates, this problem can be very hard to solve depending on the nature of types and operators that constitute the simple predicates. Given an attribute filter $f_1 = (N, T, Op, V)$ of name N, type T, operator Op and value V, and another attribute filter $f_2 = (N, T, Op', V')$ having the same name and type plus operator Op' and value V', we want to be able to decide whether or not the first filter implies the second:

$$(f_1 \Rightarrow f_2) \Leftrightarrow \forall x \in T : Op(x, V) \Rightarrow Op'(x, V')$$

Good operators are those that define equivalence relations and order relations on totally ordered sets. The usual set of basic types found in a modern programming language (numbers, strings, chars, booleans, etc.) and the usual operators (equality, inequality, regular expression match for strings), satisfy this constraint and also constitute a quite expressive vocabulary for filters.

Other systems adopt different notification models and different filtering capabilities. As a consequence, they realize different degrees of expressiveness and scalability. Section 5 comments on some of these choices with respect to the expressiveness/scalability spectrum.

4 Topologies and Algorithms

The Siena event service is architected as a distributed system. This section presents some alternative realizations in which many *event servers* cooperate to provide a network-wide event service.

4.1 Server Topologies

4.1.1 Hierarchical Server Topology

A natural way of connecting event servers is according to a *hierarchical* topology; for instance, this is the topology of the distributed implementation of the JEDI event dispatcher [2]. As shown in Figure 6, each server in a hierarchical topology has a number clients that can be either normal objects of interest or interested parties or other event servers. In addition to these connections, a server could also have a special connection to a parent server (the only outgoing arrow).

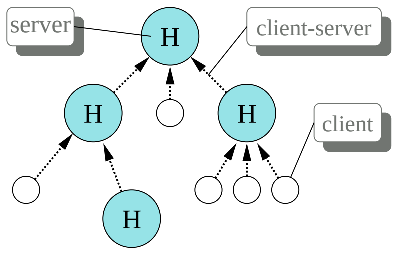

Figure 6: Hierarchical server topology

It is important to note that in this topology, a server does not distinguish between other servers and its clients, and thus it treats those servers as clients. Practically, this means that a ‘parent’ server will be able to receive notifications, subscriptions, and advertisements from all its clients, but it will send only notifications to its clients.

4.1.2 Acyclic Peer-to-Peer Server Topology

In the acyclic peer-to-peer topology, servers communicate with each other as peers, thus allowing a bi-directional flow of subscriptions and advertisements as well as notifications. Figure 7 shows an acyclic peer-to-peer topology of servers. Once again, notice the different kinds of communication occurring between clients and servers and among servers.

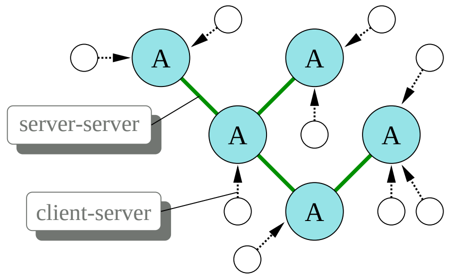

Figure 7: Acyclic peer-to-peer server topology

4.1.3 Generic Peer-to-Peer Server Topology

The generic peer-to-peer topology allows the same type of server-to-server communication introduced by the acyclic peer-to-peer, but in addition to that, it allows any pattern of connections between servers (see Figure 8).

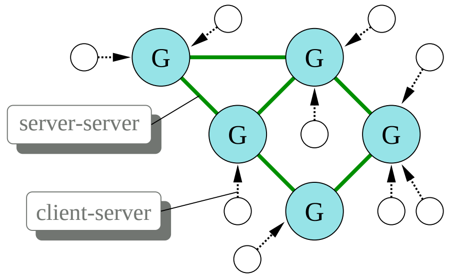

Figure 8: Generic peer-to-peer server topology

4.2 Routing Techniques

Once connected, server must exchange notifications, subscriptions, and advertisements to realize the service. This section presents the algorithms that disseminate the proper information throughout the network of servers making sure that notifications are correctly delivered. Note that a network of servers implementing the event service is logically equivalent to a network of routers connecting sub-nets and realizing multicast routing. In fact, the algorithms presented here are very similar in principle to a combination of the Internet Group Management Protocol (IGMP [6]) and a reverse-path multicast routing algorithm [5, 4]. More details on the similarities and differences with network-level multicasting can be found in Section 5.

We have defined two classes of algorithms:

subscription forwarding: This technique uses subscriptions to set the paths for notifications. Every subscription is stored and forwarded from the originating server to all the servers in the network so to form a tree that connects the subscriber 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 in reverse the path put in place by the subscription;

advertisement forwarding: This technique uses advertisements to set the paths for subscriptions, which in turn set the paths for notifications. Every advertisement is forwarded 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 path to the advertiser, thereby *activating* the path. Notifications are then forwarded only through the *activated* paths.

There exists also the degenerate case of broadcasting notifications, which we do not take into consideration. Unsubscriptions and unadvertisements are handled in a similar way to undo the effect of the corresponding subscription or advertisement. As suggested by their names, subscription forwarding and advertisement forwarding implement the subscription-based and advertisement-based semantics respectively. There are two main optimization strategies for saving communication and computation resources that can be pursued by applying these two algorithms, they are:

1. *applying filters and matching patterns upstream*: this means filtering notifications and assembling patterns of notifications as close as possible to publishers (see Figure 9);

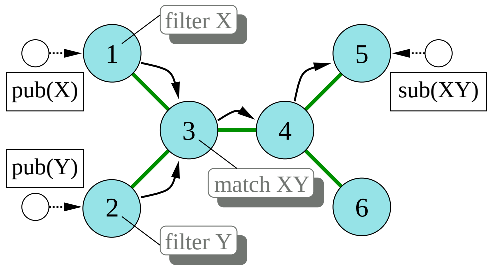

Figure 9: Applying filters and patterns upstream

2. *replicating notifications downstream*: this means multicasting notifications as close as possible to subscribers (see Figure 10).

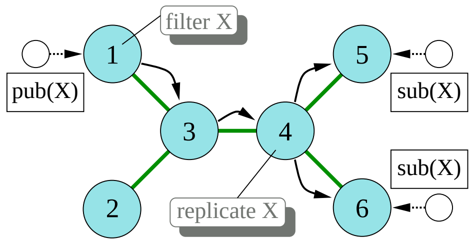

Figure 10: Multicasting of notifications downstream

The broadcasting or *flooding* process that characterizes both subscription forwarding and advertisement forwarding creates minimal spanning trees for each source. The realization of this process depends upon the underlying topology of servers. The solution is trivial in the case of acyclic topologies (i.e., hierarchical and acyclic peer-to-peer), but it requires additional data structures and protocols for the generic graph topology [3].

In propagating requests, servers maintain tables of subscriptions and/or advertisements. When an event server receives a new request, say a subscription, that is already *covered* by a previously served one, the server simply adds the subscriber to the local list and no other action is taken. If no such subscripition is present in the tables, the new request is added to the table and propagated. This allows to prune of entire subtrees in the propagation. For example, in the scenario of Figure 10, when server 4 receives the forward of a subscription for X from server 5 for the first time, it propagates it to server 3, and then all the way towards server 1. However, when server 6 sends the same subscription to server 4, server 4 stops the flooding.

Servers perform other types of optimizations too. For example, in the advertisement forwarding algorithm, when a server receives an advertisement from one of its neighbor servers for which there exist matching subscriptions, the server forwards all these subscriptions to the advertiser. In doing that, the server tries to merge the batch of subscriptions into a smaller number of more generic subscriptions. Again, this can be done thanks to the simple structure of filters and the predictable nature of predicates and types that can be used in filters.

For the recognition of patterns, event servers try to assemble patterns from smaller sub-patterns or single notifications that are already “available”. To do this, servers rely on their tables of advertised patterns. In short, when a server receives a subscription that requests a pattern, it looks up the table of advertised filters trying to break up that sequence into smaller available filters or patterns. If sub-patterns that together form the target sequence have been advertised by local clients or neighbor servers, the server dispatches subscriptions for every one of these parts and starts up a monitor that recognizes the requested sequence from the sub-sequences.

Whenever possible, a server will push the recognition of entire sub-sequences towards the sources of their components. For example, in the scenario of Figure 9, server 5 notices that the sequence XY can be broken into X and Y and that both these parts are available from the same server (4), thus server 5 simply forwards the subscription for the entire sequence. Server 4 in turns does exactly the same thing forwarding the subscription to server 3. Server 3 notices that X and Y are advertised by two different sources, thus it sends the two separate simple subscriptions and start up the monitor.

Once again notice how the compatibility relations are crucial in every step of the forwarding techniques. Also notice that the way that the compatibility relations define the semantics of the event service is motivated by the forwarding techniques. In particular, the constraints posed by the advertisement-based semantics make it possible for event servers to maintain advertisements tables that are necessary for decomposing and optimizing pattern recognition.

5 Related Work

In this section we provide a brief survey of technologies that we believe are tightly related to the problem of wide-area event notification, either because they attack the same problem or because they provide important pieces of solutions.

5.1 Internet Basic Technology

A number of Internet technologies are worth mentioning because, if nothing else, they indeed realize services on a wide-area network. Thus, even if none of them is geared towards an event notification service, it might be worthwhile to borrow their ideas vis-a-vis scalability.

5.1.1 Domain Name Service

DNS is a scalable network service that is realized in a distributed manner. The valuable idea that we can borrow from DNS is its hierarchical architecture. The reason why the hierarchical architecture works so well for DNS is that it maps very well onto the data that it manages. In fact, the space of domain names and the space of IP addresses are hierarchical themselves and the mapping between them preserves a lot of the hierarchical properties. Unfortunately, the space of notifications doesn't exhibit any hierarchical structure and, even if we decided to force this type of structure (e.g., by defining a mandatory well known attribute and a hierarchical set of values for it), this would not naturally map onto a hierarchical location of objects. Another differentiator is the essential read-only nature of the DNS service, which does not apply much to event notification services.

5.1.2 USENET News

The USENET News system with its main protocol NNTP [9] is perhaps the best example of a scalable user-level one-to-many communication facility. USENET News messages are modeled after e-mail messages, yet they provide additional information (headers) that can be used by NNTP commands to direct their distribution. NNTP provides both client-server and server-server commands. The topology of news servers is very similar to (and in fact inspired) the acyclic peer-to-peer topology.

The main problem with USENET News and NNTP that limits usability as an event service is that the selection mechanisms are not very sophisticated. At the protocol level, messages are filtered based only on their newsgroups and on their date. The newsgroup name space is organized in a hierarchy, and the protocol allows wild-card expressions over group names. Although group names and sub-names reflect the general subject and content of messages, the filter that they realize is too coarse-grained for most users and definitely inadequate for a general-purpose event service. This results in unnecessary transfers of entire groups of messages. The service is thus scalable but still quite heavyweight, and the time frame of news propagation ranges from hours to days.

5.1.3 IP Multicast

MBone is a network-level infrastructure that realizes an efficient one-to-many communication service. An MBone or multicast IP address is a virtual address that corresponds to a *group* of hosts. Hosts can join or leave a group at any time. An IP packet addressed to a host group will be delivered to all the hosts in that group. IP multicast per se is a connectionless best-effort (unreliable) service. A reliable transport layer can be implemented on top of IP multicast.

We consider the IP multicast infrastructure and its routing algorithms to be the most important technology related to the Siena event service. IP multicast may be used as an underlying transport mechanism for notifications and the ideas developed for routing multicast packets can be adapted to solve the problem of forwarding notifications in an event service. But the IP multicast infrastructure alone does not qualify as an event service because of limitations in its addressing. The main problem is mapping expressions of interest into IP group addresses in a scalable way. Even assuming that a separate service, perhaps similar to DNS, is available for managing and resolving the mapping, the addressing scheme itself still poses major limitations in combining filters and patterns. Because MBone never relates two different IP groups, it would not be possible to exploit the compatibility relations between filters and thus it would not be possible to assemble filters into patterns. Notifications matching more than one filter or participating in more than one pattern would map into several multicast addresses, each one being routed in parallel and autonomously by MBone, thus defeating the whole purpose of the event service.

5.2 Event Notification Technologies

Some technologies specifically realize an event notification service, and some of them also attempt to extend their support to wide-area networks. To relate these systems to Siena, we adopt the classification framework defined in [2] and concentrate in particular on subscription languages.

5.2.1 Channel-based Subscriptions

The simplest subscription mechanism is the *channel*. Interested parties can listen to a channel by subscribing to it. An object of interest publishes a notification by addressing it to a specific channel; as a consequence, the notification is delivered to all the parties that are listening to that channel. Channel-based event services offer coarse-grained filtering and no patterns. Since channels define a partitioned address space for notifications, their service is equivalent to a reliable multicast communication. The CORBA Event Service [12] adopts a channel based architecture.

5.2.2 Subject-based Subscription

Some systems extend the concept of channel to *subject-based* addressing. In this case, event notifications contain a well-known attribute (the *subject*) that determines their address, while the remaining part of the notification is opaque for the event service. The main difference with respect to channels is that here subscriptions can express interest in many (potentially infinitely many) subjects/channels by specifying some form of expressions to match a subject. Also, in this model, two different subscriptions can define overlapping sets of notifications. TIBCO Rendezvous [8] adopts a subject-based subscription mechanism. In TIB Rendezvous, the subject is a list of strings over which it is possible to specify filters based on a limited form of regular expressions; for example, the filter `economy.exchange.*.MSFT*` will select all the notifications whose subject contains `economy` in its first position followed by `exchange` in second position, any string in third position, and a fourth string that starts with `MSFT`.

5.2.3 Content-based Subscription

By extending the domain of filters to the whole content of notifications we obtain another class of subscriptions called *content-based*. Content-based subscriptions are conceptually very similar to subject-based ones. However, by making the whole structured content of notifications visible to subscriptions, they give more freedom in the encoding the data upon which filters can be applied and which the event service can use for setting up routing information. Moreover, exposing the structure of notifications makes their type system (if any is adopted) visible too, thus, allowing more expressive and clear filters. Examples of systems that provide this kind of subscription language are JEDI [2], Yeast [10], GEM [11], Elvin [14], and Siena itself.

6 Experience

The evaluation of distributed software systems is a difficult task, and the Siena event service is no exception. Because of its highly distributed nature, it is impossible to implement an event service and deploy it on a significant number of nodes just to see how it works. Not only it would be difficult to measure its performance, but also the cost of refining its topologies and algorithms would be too high at least in the early phases of its design.

So, in order to obtain feedback early in the design and development of Siena, and to have a quantitative evaluation of its topologies and algorithm, we adopted an approach that is common practice in the computer networks community for the validation of communication protocols and distributed systems in general. We chose to perform systematic simulations of different combinations of server topologies and dispatching algorithms in several network scenarios.

6.1 Simulation Framework

In simulating a network scenario, several models must be incorporated into the scenario. These models define the network topology, the layout of event servers, the population of applications (i.e., the distribution of interested parties and objects of interest), their behavior, etc. Clearly, every model makes certain simplifying assumptions. Also, every model is characterized by a significant number of parameters, which must be adjusted to reflect reality as faithfully as possible. These are the most important models we adopted in our simulation framework together with their primary parameters:

- *network model*: The network model describes the physical (wide-area) communication network underlying the event service. The network is characterized by a graph. Each node in the graph represents a host or a cluster of nodes connected by a fast local-area subnet, and every edge represents a link with its latency and bandwidth. One way of modeling networks is to use a real network as a *benchmark* for which these parameters can be measured. The other way, adopted in our framework, is to use randomly generated graphs that are good approximations of the real network [15].
- *server model*: We use a layout of servers in which every network node hosts one server. We also assume that the connections between servers match the physical topology of the network. This second principle assumes that system administrators have a view of the topology of the immediate neighborhood of their subnet and that they can configure the event servers accordingly.
- *object distribution*: This model defines the distribution of objects of interest and interested parties among subnets. For now, we use an homogeneous distribution of objects. Thus, two parameters are given for the number per node of objects of interest and interested parties respectively.
- *objects behavior*: Although we could simulate real applications, we model applications as Poisson processes with respect to generation and consumption of events. Thus, the parameters that govern their behavior are the average time between two requests and the ratio between the number of requests issued per request type. This defines, for example, how many notifications are published for each advertisement.
- *computation and communication model*: Objects communicate by exchanging messages. These messages can carry event service requests (notifications, subscriptions, etc.) as well as control messages or forwarded messages that follow from some service request. Objects execute their own algorithm in response to messages or to the expiration of a time-out. Objects can send messages, set time-outs, create other objects or destroy other objects.

6.2 Running Simulations

Once a network scenario is defined, we run several simulations and collect traces of all the low-level messages exchanged between processes, hosts, and subnets. Then we analyze these traces by grouping messages according to some specific criteria (e.g., per host, per event service request, per type of request) and by computing the message count, minimum and maximum values, average value, and standard deviation of metrics such as network cost and delay. The network *cost* is a per-link parameter provided by the network model that accounts for the usage of communication resources in sending a message through a link. In our framework, we assume that the cost is proportional to the inverse of the bandwidth. By varying some parameters of the network scenario, such as the number of interested parties per node, we can plot these metrics and obtain indications of the behavior of that algorithm.

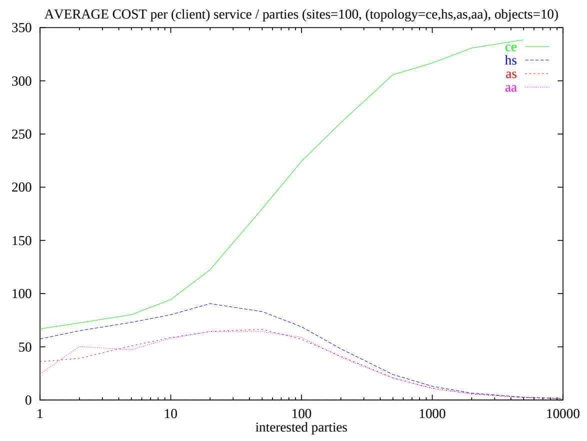

Figure 11: Scalability of some topologies and algorithms

Figure 11 shows the scalability of four combination of server topology and routing algorithm: ce = centralized, hs = hierarchical + subscription forwarding, as = acyclic + subscription forwarding, and aa = acyclic + advertisement forwarding. All the scenarios were executed on a network of a hundred nodes. The metric plotted is the average cost of messages grouped on a per-request basis; the unit of measure is not really relevant since we just want to compare different curves. The independent variable is the total number of interested parties.

The simulations that we performed so far clearly show that our topologies for distributed event services provide the scalability that can not be achieved with a centralized solution. In distinguishing the various distributed topologies, simulations show that the peer-to-peer topologies distribute the traffic evenly among servers, as opposed to the hierarchical topology that tends to over-load only a few nodes (at the highest level of the hierarchy). The simulator has been also a very effective testing tool for the development of the routing algorithms.

7 CONCLUSION

This paper has described our work on Siena, a distributed, Internet-scale event notification service. We have described the design of the interface to the service, the algorithms and topologies we have designed to support event notification, and our ongoing simulation and evaluation work.

The simulation framework that we constructed has helped us significantly in refining topologies and algorithms. Also, the simulations confirm our intuitions about the scalability of the topologies and algorithms that we propose. However, we do not consider our evaluation effort to be complete. In fact, we plan on continuing our evaluation effort by exploring the parameter space 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 have implemented a prototype of Siena that realizes the acyclic topology with the subscription forwarding algorithm. We used this prototype as the event service of an agent-based deployment system called the SoftwareDock [7]. The current version of the prototype provides a reduced version of the notification model with only strings and integers and a few operators. Siena uses standard Internet technology, so its data model is transmitted in XML format, and servers are able to use straight TCP/IP as well as SMTP as a transport layer for messages. We plan on extending the prototype to implement the advertisement forwarding algorithm with a larger variety of types and operators and other transport layers including HTTP. This new version of the prototype will also allow us to apply the pattern matching optimizations that we discussed.

Acknowledgments

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

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021; by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061, F30602-94-C-0253 and F30602-98-2-0163; 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, Air Force Office of Scientific Research or the U.S. Government.

References

[1] T. Barners-Lee. Universal Resource Identifiers in WWW, A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web. Internet Request For Comments (RFC) 1630, Internet Engineering Task Force, June 1994.

[2] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. Technical report, CEFRIEL – Politecnico di Milano, Milano, Italy, August 1998. submitted for publication.

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

[4] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei. The PIM Architecture for Wide-Area Multicast Routing. *IEEE/ACM Transactions on Networking*, 4(2):153–162, April 1996.

[5] S. E. Deering and D. R. Cheriton. Multicast Routing in Datagram Networks and Extended LANS. *ACM Transactions on Computer Systems*, 8(2):85–111, May 1990.

[6] W. Fenner. Internet Group Management Protocol, Version 2. Internet Request For Comments (RFC) 2236, Internet Engineering Task Force, November 1997.

[7] R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An Architecture for Post-Development Configuration Management in a Wide-Area Network. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, Baltimore, USA, May 1997.

[8] T. Inc. Rendezvous information bus. Technical report, TIBCO Inc., 1996. http://www.rv.tibco.com/rvwhitepaper.html.

[9] B. Kantor and P. Lapsley. Network News Transfer Protocol – A Proposed Standard for the Stream-Based Transmission of News. Internet Request For Comments (RFC) 977, Internet Engineering Task Force, February 1986.

[10] B. Krishnamurthy and D. S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[11] M. Mansouri-Samani and M. Sloman. GEM A Generalized Event Monitoring Language for Distributed Systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[12] Object Management Group. CORBAservices: Common Object Service Specification. Technical report, Object Management Group, July 1998.

[13] D. S. Rosenblum and A. L. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proceedings of the Sixth European Software Engineering Conference*, Zurich, Switzerland, September 1997. Springer-Verlag.

[14] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quencing. In *Proceedings of AUUG97*, July 1998.

[15] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom*, April 1996.