POLITECNICO DI MILANO
DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA

Architectures for an Event Notification Service Scalable to Wide-area Networks

PhD Thesis of:
Antonio Carzaniga

Advisor:
Prof. Alfonso Fuggetta

Co-Advisors:
Prof. Alexander L. Wolf
Prof. David S. Rosenblum

Supervisor of the Ph.D. Program:
Prof. Carlo Ghezzi

X ciclo

to my wonderful family

Acknowledgments

I would like to thank my advisor Prof. Alfonso Fuggetta for trusting, encouraging, and supporting me all throughout my PhD studies. With advices and lively discussions, he has contributed a lot to the development of this work.
I am deeply grateful to Prof. Alexander Wolf for his support, for his guidance and for the confidence he gave to me during the past two years.
I am also thankful to Prof. David Rosenblum. Prof. Wolf and Prof. Rosenblum initially formulated the basis for this research and then worked together with me in pursuing it.
A remarkable influence on my work was brought also by the ideas and the insights of Gianpaolo Cugola, Elisabetta Di Nitto, Richard Hall, Dennis Heimbigner, and André van der Hoek.
I also want to thank prof. Carlo Ghezzi for being supportive in many occasions.
I shared my experience as a PhD student between the Software Engineering group of Politecnico di Milano and the Software Engineering Research Laboratory of the University of Colorado at Boulder. In both these environments I had the opportunity to meet great people that in so many ways contributed to this achievement. In addition to the ones I already mentioned, I would like to express my gratitude to Luciano Baresi, Gino Biondini, Fabiano Cattaneo, Thorna Humphries, Artur Klauser, Pier Luca Lanzi, Edoardo Marcora, Mark Maybee, Mattia Monga, Alessandro Orso, Gian Pietro Picco, Matteo Pradella, Giuseppe Ricci, Massimo Ricotti, Sergio Silva, Judith Stafford, Laura Vidal, Giovanni Vigna, . . . and many others.

Grazie di cuore a tutti.

Milano, December 1998.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work is about an infrastructure for supporting *event-based* applications on a wide-area network.

A wide range of software systems are designed to operate in a reactive manner. In such systems, the high-level control flow is not explicitly programmed, instead it is driven by the occurrence of *events*. These systems realize their functionality by performing some actions in response to events, possibly using the information associated with the stimulating events. Examples of reactive systems are integrated development environments [64, 38, 11, 33], work-flow and process analysis systems [16, 4], graphical user interfaces [59], network management systems [65], software deployment systems [32] and security monitors [36, 56, 75].

There are two major motivations for designing applications as reactive systems. First, some applications are reactive by their nature. Typically, the ones that involve the direct interaction of human agents are characterized by an asynchronous input of relatively small pieces of data. In these cases, and in the general case of "on-line" input, the concept of *event* is a good modeling and design abstraction. Similarly, the same abstraction is useful for those components that, although not necessarily functioning in an asynchronous way, are integrated by means of some communication mechanisms that introduce asynchronicity in their interactions. The other benefit of adopting an event-based style is that it requires only a loose coupling for the integration of heterogeneous components. Components do not need to export interfaces to be accessed by other components. Components can request some services without addressing a specific server component and, to a certain extent, components can interoperate even if they have been designed and developed separately without any mutual knowledge.

A common logical component of every event-based application is what we call an *event observation and notification service*, or more concisely an *event ser-*

vice. The event service observes the occurrence of events or the occurrence of combinations of events and consequently notifies all the applications or components that have declared their interests in reacting to such events. Because the semantics of event-based applications is substantially independent of the mechanisms that are used to capture and dispatch events of interest, it is convenient to separate the event service from all the applications that make use of it. In accordance to many standardization efforts that have been proposed recently (e.g., [58]), and according to strategic plans and research in network technology [15], we envision a unified event service implemented as a common "middle-ware" to support the event-based interaction among software components.

The idea of integrating software components by means of a common event service seems to be very promising ([12, 6]) especially for those distributed applications that are deployed on a wide-area network such as the Internet. For one thing, the vast number of available information sources offers a great deal of opportunities for the development of new applications. New classes of wide-scale event-driven applications can be devised including stock market analysis tools, efficient news and mailing systems, data mining tools, and indexing tools. Also, many existing applications that are already designed to exploit event-based infrastructures can be proficiently integrated at a much higher scale thanks to the "global" connectivity provided by the network. For example, work-flow systems can be federated for companies that have multiple distributed branches or even across corporate boundaries, or else software deployment systems can connect software producers and consumers through the Internet [32]. In general, the asynchronicity, the heterogeneity, and the high degree of loose coupling that characterize wide-area networks suggest that a wide-scale event service would be a good integration infrastructure for existing systems and for new applications.

1.1 Focus and contribution of the thesis

This work presents *S*IENA[1], a project directed towards the design and implementation of a scalable general-purpose event service.

Numerous technologies that realize an event service have been developed and effectively used for quite a long time, examples are Field [64], SUN ToolTalk [38], and Yeast [44]. However, most of them are targeted towards single computers or at most local-area networks, and it is very clear that they can not be simply "adapted" to scale to the Internet. In fact, 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

[1]*S*IENA is an acronym for *S*calable Internet Event Notification Architecture

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 we can find new technologies such as TIB/Rendezvous™ [73] 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 to the problem of scalability. The main shortcoming of all of these technologies is that they are specific to some application domain and not flexible enough to be usable as a generic and open infrastructure for integrating event-based applications (see [66, 12]).

In summary, we see two main challenges in the area of event-based infrastructures, that SIENA proposes to address:

- *scalability*: this is the ability to provide an event service across a whole wide-area network. The large *scale* of the network implies a large number of applications scattered over many distant sites exchanging many events;

- *expressiveness* or *flexibility*: this quality denotes the ability of the event service to provide a good level of support to a wide variety of applications. Expressiveness has to do with the type of information that can be attached and propagated with notifications as well as the ability of the event service to aggregate and filter events as precisely as possible on behalf of applications.

Intuitively, a simplistic service can be implemented in a very scalable way, whereas an event service with a rich semantics poses serious scalability limitations. Thus, this thesis focuses on the trade-offs that exist between scalability and expressiveness in a distributed 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 validation and verification of the solutions that we propose. To this end, we used a simulation environment by which we performed systematic simulations of our architectures and algorithms in several network scenarios. Here we present the framework that we built and the modeling abstraction that we adopted to perform this analysis. We also discuss some initial results that clearly differentiate the SIENA event service from traditional ones. Further simulations will help clarify the trade-offs and differentiators between the alternative solutions that we propose. In addition to the simulation environment, we implemented a real prototype of SIENA, consisting of a server that realizes one of our distributed architectures plus a client-side interface with a mapping to the Java™ language. This prototype has been used in distributed settings of limited scale to support an event-based software deployment system.

1.2 Structure of the thesis

Chapter 2 explores the problem space for an event service. Related technologies and systems are surveyed and classified. This classification helps us in isolating and defining the main challenges that we intend to face in this research. Chapter 3 defines the SIENA event service. This definition covers some basic terminology, the data model adopted to represent event notifications, the interface functions exported by the event service, and the semantics of the event service. Chapter 4 describes the dispatching algorithms defined for SIENA. The algorithms are presented and discussed referring to the semantics of the event service and emphasizing the optimization strategies that can be applied in forwarding notifications and in matching patterns of notifications. Chapter 5 illustrates the simulation framework. First we describe our network model and the simulation methodology, then the parameters and the metrics we focused on, and finally we show the results we obtained with our algorithms in some network scenarios. Chapter 6 concludes summarizing the results of this research indicating also some directions for future work.

Chapter 2

Background and New Challenges

The field of event-based systems and infrastructures has been very lively in the past few years. This research area is not entirely new since the concept of event-based integration has been studied and adopted in several different branches of computer science for a long time. However, similarly to many other research areas that are related to the exploitation of wide-area networks —mobile code, agents, and hyper-media, just to mention a few— the event-based style and its related technology has received a new impetus following the explosive development of the Internet. Undoubtedly, the global connectivity offered by the Internet has made a massive amount of information available for applications and people and it has dramatically increased the potential applications of event-based technologies. But at the same time, it has unveiled a whole variety of new challenges for the design of such technologies, thus drawing the attention of both academia and industry.

Indeed, the event-based style holds the promise of introducing revolutionary changes in the way distributed software systems are built and integrated. Under this pressure, the spectrum of technologies related to the concept of wide-area event service has become vast to a point that it is often difficult to identify the applicability of every single technology and its value added with respect to well known “old” technologies.

One problem is the multitude of available and publicized systems. A fair number of technologies implementing some variances of an event service have been available for a long time. But also, in addition to these “old” systems, more recent efforts have been made by industry, academia, and various organizations, in the direction of supporting an event service over wide-area networks.

The other main problem with evaluating technologies related to event-based systems is that this field lacks a widely accepted characterization of the problem as well as a common vocabulary [80]. As a result, an event service is often confused with specific *applications* (e.g., a bulletin board) or *application domains* (e.g., human-computer interaction), or *design frameworks* (e.g., standard interfaces or naming conventions like JavaBeans™ [72]), or else, because an event service provides and uses some sort of communication facility, it is compared to other general purpose —but radically different— communication mechanisms such as point-to-point messaging systems.

As a first step in this work, we intend to identify the basic *principles* underlying event-based technologies. With these principles, we discuss and assess existing technologies and we give a precise characterization of the most important research issues that remain open.

An initial working definition

For the time being, we will adopt the following informal definitions that intuitively describe an *event service*:

- an *event* is represented by a data structure called *event notification*, or simply a *notification*;
- we call *interested parties* those applications that consume event notifications;
- the event service accepts *subscriptions* from interested parties. Subscriptions express the class of events or combinations of events an interested party is interested in;
- whenever an event or a combination of events occurs, the job of the event service is to dispatch a notification to all the interested parties that subscribed for that event or combination of events.

The terms that we used here refer to the conceptual framework posed by Rosenblum and Wolf in [67]. Note that at this point we ignore most of the issues discussed in that framework, in fact we have not even mentioned what an event is or what it means to combine events. We will just assume that events occur somewhere and that the event service is capable of detecting their occurrence and extracting some data describing their context. Also, for combinations of events, we can think of sequences of events, e.g., in a security monitor, three consecutive login failures for a user are a significant combination of events. In Chapter 3 we will give a more comprehensive and formal definition of event service.

2.1 Related technology

In this section we compile 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.

		architecture	
		centralized	*distributed*
service	*not related*		Internet technology
	event service	"old" message-based environments	"new" distributed event services

Table 2.1: A first cut in classifying related technologies

As an initial cut, we examine two dimensions (see Table 2.1). One is the nature of the architecture of the system for which we see two values: *centralized* and *distributed*. The other one is the kind of service, that we classify as *not related* and *event service*. This simplistic schema allows us to distinguish three main categories of technologies that we want to examine further: *centralized event-based or message-based environments* that use events with a centralized infrastructure to integrate components, *Internet technologies and protocols* that provide distributed infrastructure not specific to event-based systems, and *distributed event-based services* that combine the event-based style with a distributed realization.

2.1.1 Centralized event/message based environments

In this group, we classify graphical user interfaces, message-based software development environments, and generic event-based infrastructures that have a centralized architecture. We consider the fact that these systems are implemented with a centralized architecture as their major limitation. Nonetheless, these systems provide valuable insights as far as the type of service that they offer.

Message-based integrated environments

The idea of integrating different components by means of *messages* is first realized by Field [64]. Field, together with other commercial products including

HP SoftBench [11], DEC FUSE [33], and Sun ToolTalk [38], implements an environment in which several software development tools can cooperate by exchanging messages. Messages are the means by which one tool can request services to be carried out by other tools, or they can be sent out by a tool to announce a change of state —for instance, the termination of an operation— so that other tools can proceed with their task. This use of messages enables a fine control of the software development process by coordinating the actions of different tools. As an example, consider the interaction between a compiler and an editor. While compiling a source file, the compiler might find parse errors. Whenever such an error occurs, the compiler posts a message that requests that an editor open the source file positioning the cursor at the specific line that caused the error. Similarly, the editor might signal the compiler when the developer has completed its update to a source file.

The heart of these software development environments is a message dispatcher or *message bus*. In some cases tools address their requests to specific instances of other tools, but other times a tool might simply request a service. In these latter cases, the message bus gets in charge of dispatching that message to a tool that is able to handle the requested service. Hence, the role of the message bus is to deliver a message to the right application, i.e., one that registered a corresponding service, possibly invoking the application if none is already running.

Integrated software development environments embody a primitive event service since registering tools to handle service requests is equivalent to subscribing for those requests. However, domain of event notifications and subscriptions is usually very limited. Tools can generate a fixed set of messages and in some cases (e.g., in DEC FUSE), this set of messages is statically mapped into a set of *call-back* procedures that are hardwired within the tool.

Event-action systems

Yeast [44] is an event-action system. Yeast allows the definitions of *rules* (or specifications) that have the general form (*event-pattern* **do** *action*). Unlike message-based integrated environments, Yeast is very similar to a general-purpose event service. For the left part of a specification, Yeast defines a rich language that allows detailed event patterns including also temporal expressions. The action part of a specification is a shell script.

Yeast provides two classes of mechanisms to detect the occurrence of events. Events can either be explicitly announced by applications by means of the `announce` command (provided by Yeast), or they can be observed directly by Yeast. The events that Yeast is able to poll are defined with a set of objects each one having a number of attributes. Yeast matches an event whenever one of these attributes changes. Classes of pre-defined objects are: file, directory, host, file system, host, tty, and user. Examples of attributes are modification time and permissions for files and directories, load and number of users for hosts, login time for users, etc.

A different class of systems that are conceptually equivalent to event-action systems are active databases [13]. In active databases, primitive events are operations on database objects. Event-action rules are also called *triggers*. Events can be combined and correlated in the left part of the trigger. A trigger can also impose additional conditions (or *guard*) to be evaluated once the requested sequence of events has occurred. If this condition is satisfied, the database executes the action part of the trigger, which is usually expressed as a query.

The main difference between an event service and an event-action system like Yeast is that an event service only dispatches event notifications, possibly to multiple recipients, so responses to events are executed by interested parties within their context. This way, the observation of an event or a pattern of events can be de-coupled from the corresponding response. Instead, an event-action system has a *logically* centralized architecture because actions are executed by the system itself within its environment. In other words, in event-action system, there is a strict binding between the observation of events and the reaction to those events.

User interfaces

The X Window System is a client-server windows system. The server manages the physical display and its input devices —usually a keyboard and a mouse— while applications connect as clients. Clients react to input events dispatched by the server and request graphical display operations to the server. The X Window protocol allows a very fine granularity of control, so usually applications are built with higher level libraries that implement graphical objects (or *widgets*). Widgets are themselves reactive objects: they collect some classes of events from the server and execute some actions in response to those events.

A core component of user interfaces like the X Windows System is the dispatcher of input events that resides both on the server and on the client. Usually, the client sets some filters on the server so that the server will dispatch only those events that are of interest for that client. For example, the application might want to avoid receiving *mouse-motion* events, but it might be interested in receiving *mouse-click* events. A similar dispatcher can be set up on the client side where different widgets can register their *call-back* routines in response to some specific events or sequences of events, for example, a button widget might register a call-back function to be executed right after a *key-pressed* event is followed by a *key-released* event.

The X Window System is not an event service, however its event notifications have an articulated data structure and, in general, it is designed to cover several aspects of human–computer interactions that involve sophisticated event-based components. Hence its relevance to this research.

2.1.2 Internet technology

A number of Internet technologies are worth mentioning in this work because they indeed realize services on a wide-area scale. Thus, even if none of them is really designed to realize an event notification service, it might be worthwhile to borrow their ideas vis-a-vis scalability.

Domain Name Service

Domain Name Service (DNS [54, 53]) maps symbolic host or domain names into IP addresses. The current implementation of DNS has proved to be extremely scalable especially considering the recent explosion of domains caused by the commercial exploitation of the Internet. DNS is realized with a distributed architecture. In particular, DNS servers form a hierarchical structure. The reason why a hierarchical architecture works so well for DNS is that the structure of servers can be naturally laid out so to map very well onto the structure of the data that they manage. In fact, the space of host names and the space of IP addresses are hierarchical themselves and the mapping between them preserves a lot of the hierarchical properties. In other words, because host names are partitioned in domains (e.g., `edu`, `it`, `com`, etc.), that in turns are partitioned in sub-domains (e.g., `colorado`, `uci`. etc.) and so on, the physical architecture of DNS can be set up so that requests that pertain one domain are handed off to a server dedicated to that sub-domain and no other host outside that domain can affect the mapping realized within the domain.

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

Another differentiator of DNS with respect to an event service is the essential read-only nature of the DNS service. In fact, DNS is very efficient in resolving names also thanks to appropriate caching policies. This is made possible by the the fact that mappings maintained by DNS are relatively stable, i.e., they are read much more frequently then they are modified. In general, this is not true for the information exchanged in event notification services.

USENET News

The USENET News system with its main protocol NNTP [39] is perhaps the best example of a scalable user-level many-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. The infrastructure that supports the propagation of articles is made of a network of news servers. NNTP servers store news articles, pass articles to clients, and exchange articles with other servers. NNTP provides both client-to-server and server-to-server commands. The infrastructure of servers can be easily configured and extended with an incremental process that is managed locally by server administrators. New servers can join the infrastructure by connecting as *slave* to another (*master*) server that is already part of the infrastructure and that is willing to share articles. The structure of serves thus formed is a tree in which articles are flooded from master servers to slave servers. Besides receiving the feed of articles from their master server, slave servers can also send articles that have been posted locally to their master server.

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

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

IP Multicast

IP multicast [22] is a network-level infrastructure that extends the Internet protocol in order to realize an efficient one-to-many communication service. IP multicast is an extension to the usual unicast routing mechanism realized over the Internet. The network that realizes this extension is also referred to as MBone. In MBone, a multicast address (or host group address) is a virtual (IP) address that corresponds to a *group* of hosts possibly residing on different subnets. IP datagrams that are addressed to a host group are routed to every host that belongs to the group. Hosts can join or leave a group at any time using a special group membership protocol [27]. IP multicast per se is at the same level as IP, thus it is a connectionless best-effort (unreliable) service. A reliable transport layer can be implemented on top of IP multicast[47, 79].

We consider the IP multicast infrastructure and its routing algorithms to be the most important technology related to the SIENA event service. As a first observation, note that IP multicast can be used as an underlying transport mechanism for notifications, but the most important aspect that makes IP multicast so relevant to this research is that an event service can be thought of as a multicast communication infrastructure in which addresses are *expressions of interest*. With this model the ideas developed for routing multicast datagrams can be adapted to solve the problem of forwarding notifications in an event service.

Unfortunately, the IP multicast infrastructure alone does not qualify as an event service because of limitations in its addressing. The first issue is mapping expressions of interest into IP group addresses in a scalable way. The second issue is the limited expressiveness of IP addresses. In fact, even assuming that we can encode subscriptions so to map them into IP multicast addresses and that a separate service, perhaps similar to DNS, is available for managing and resolving the mapping, the addressing scheme itself still poses major limitations when observing combinations of events or when combining different subscriptions into more generic ones. Because IP multicast never relates two different IP groups, it would not be possible to exploit the similarities between subscriptions. Different notifications matching more than one subscription or participating in more than one combination of events would map into several separate multicast addresses, each one being routed in parallel and autonomously by the IP multicast network, thus defeating the whole purpose of the event service.

In Chapter 4 we will examine in greater details the similarities between SIENA and MBone highlighting the ideas that SIENA inherits from the work and experience done for MBone with special attention to multicast algorithms.

2.1.3 Distributed event-based infrastructures

Some technologies specifically realize an event notification service with a distributed infrastructure. Among them we find JEDI [18], Elvin [68], TIBCO's TIB/Rendezvous™ [73], Keryx [81, 40], and iBus [69].

To our knowledge, except perhaps for TIB/Rendezvous, no one of these systems has been actually deployed and used on a significant number of sites across the boundaries of a local-area network. Also, while all of these systems have implementations available, either for free or commercially, we are not aware of any systematic assessment of their stability or their scalability on a wide-area network[1]. Hence for this class of systems, we focus on the analysis of the expressiveness of the service that they provide.

To do this, we examine the data model that they adopt for event notifications, and in particular we distinguish them according to the part of that data model that is visible to the event service through subscriptions. In other words, we classify the expressiveness of the event service by the expressiveness of its subscriptions that in turn depends on the structure of notifications. This criterion follows the classification framework defined in [19].

Frameworks for distributed event-based interaction

There is a class of infrastructures that we discuss here that do not realize an event service, although they are publicized as such. These infrastructures are frameworks, typically frameworks of virtual classes (or interfaces) in an object oriented language, that support an event-based interaction among software components.

The most significant example is the Java™ Distributed Event Specification [71]. This framework defines the Java interfaces of roles such as *event generators*, *event listener*, and *event notifications*. Typically, an event generator exports a *register* method that allows event listeners to declare their interest in the events emitted by the event generator. The listener interface defines a method called *notify* that will be called by the event generator whenever an event occurs. This framework of classes allows a distributed interaction because both event generators and event listeners are *remote* objects and their interaction is handled by means of RMI calls [70, 26].

The notification model adopted by the Java Distributed Event Specification is very versatile. In fact notifications are Java objects implementing the *RemoteEvent*. The filtering capability is instead rather limited. Event notifications have an event *id* of type *long*. The registration of a listener can select events based only on their *id*.

It should be clear that, regardless of the expressive power of notifications and subscriptions, this framework does not specify an event service since event listeners directly contact event generators and, the other way, event generators directly notify event listeners. This introduces a strong dependency between producers and consumers of event, thereby defeating the purpose and the benefits of having an event service.

[1]TIB/Rendezvous is a commercial product. Thus its evaluation is somewhat difficult. The information we gathered on TIB/Rendezvous are derived mainly from its user manual. We were not able to get specific technical data neither about its architecture nor about its usage and its scalability.

Channel-based subscriptions

The simplest subscription mechanism is what is commonly referred to as *channel*. Interested parties can subscribe or *listen* to a channel. Applications explicitly notify the occurrence of events by posting notifications to one or more channels. The part of an event that is visible to the event service is the identifier of the channel to which the event has been sent. Every notification posted to a channel is delivered by the event service to all the interested parties that are listening to that channel.

The abstraction of the channel is equivalent to the one given by a mailing list or an IP multicast group address. Channel-based event services are functionally identical to a reliable multicast with a one-to-one mapping between channels and multicast addresses, in fact iBus, a channel-based event service, uses a transport mechanism based on IP multicast [48] (in which the mapping is the identity function). There is no interplay between two different channels and in most of the system, it is not clear how channels can be publicized. In addition to iBus, the CORBA Event Service [58] adopts a channel based architecture.

Subject-based subscription

Some systems extend the concept of channel with a more flexible addressing mechanism that is often referred to as *subject-based* addressing. In this case, event notifications contain a well-known attribute —the *subject*— that determines their address. The remaining information contained in the notification, although structured with an expressive type system, remains 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 be evaluated against the subject of a notification. This implies that a subscription might define a set of event notifications, and two subscriptions might specify two overlapping sets of notifications. This in turn means that one event may match any number of subscriptions.

JEDI [18] and TIB/Rendezvous [74], as well as the USENET News system, adopt a subject-based subscription mechanism. In both TIB/Rendezvous and JEDI, the subject is a list of strings[2] 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 first position followed by `exchange` in second position, any string in third position, and a fourth string that starts with `MSFT`. Note that both, JEDI and TIB/Rendezvous allow to attach structured and typed infor-

[2] In JEDI, events are given in the form of a function call where the first string is the function/event name and the following ones are the parameters.

mation to event notifications, JEDI with an object-oriented model implemented in Java, TIB/Rendezvous with a record structure with mappings on different programming languages. However, this information is not accessible from subscriptions.

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, since they can access the whole structured content of notifications visible, they give more freedom in encoding the data upon which filters can be applied and that the event service can use for setting up routing information. Moreover, if the notification model adopts a type system, exposing the structure of notifications makes their type system visible too, thus, allowing more expressive and readable filters as well as enabling some consistency checking. Examples of systems that provide this kind of subscription language are, Yeast [44], GEM [49] (these two are not distributed though), Elvin [68], Keryx [41] and SIENA itself.

2.2 New challenges for Event-based technologies

2.2.1 Discussion and principles

From the survey of systems that we have gone through, we can distill a set of fundamental principles and a set of features that we believe are essential in understanding problems and solutions in the context of event-based systems.

Conceptual framework

The first step in analyzing event services is to identify and model the entities that partake in the event-based interaction. A conceptual framework defines terms and roles for these entities. The models proposed in [67] offer the guidelines for this task. We will adopt some of these models and terms and we will develop a conceptual framework for SIENA in Section 3.1.

Architecture of the event service

The architecture of the event service describes the software components that realize the event service together with their connections. The description should emphasize the location of components, the topology of their connections, and how connections are controlled.

Clearly, the architecture has a significant impact on functionality and scalability of a system. For example, with a centralized event service, it is relatively easy to implement complex filtering of notifications, but it is evidently difficult to obtain a scalable service. On the other hand, a distributed event service seems to guarantee more scalability, but at the same time, it introduces serious difficulties in realizing complex filters and in guaranteeing time ordering of events.

Event notification space

The data associated with an event is captured and transmitted by means of event notifications. Thus, the data model chosen for event notifications defines what can be communicated with events or at least it dictates the way information have to be encoded. Just to give some examples, an event notification can be a single number, a string, a sequence of strings, a list of named attributes, a typed (flat) structure, a composite structure, an object in an object-oriented language, etc.

Given a certain structure for notifications, an event service might also introduce some data with pre-defined semantics, for example, assuming event notifications in the form of lists of named attributes, an event service might automatically add some attributes to every notification to pass additional "system" information such as a time-stamp or authentication credentials.

Subscription language

Subscriptions express the interests of applications. With a subscription, an application can instruct the event service to monitor a certain event or combination of events. We distinguish *simple* and *compound* subscriptions. We call *simple* those subscriptions that select one event notification at a time while we call *compound* the ones that can select a combination more than one notifications. For Example, a simple subscription could request all the "alarm" events, while a compound subscription could request all the sequences of three or more "alarm" events.

It is useful to distinguish two concepts within the analysis of the subscription language: the *filtering function* that selects single events based on their contents, and the *pattern monitoring function* that recognizes combinations of events.

Setting a *filter* with a subscription means defining a predicate that the event service evaluates against every notification. It is of fundamental importance to define the domain of these predicates. In other words, it is crucial to determine (1) which parts of a notification are visible to the event service for the evaluation of subscriptions and (2) what kind of primitive predicates and connectors are available.

A *pattern monitor* is a mechanism that groups events. Every single event in a pattern might be selected through a filter, while the "proper" pattern monitor defines the relations (e.g., temporal relations) among events that form the group. Typically, events are initially ordered in a temporal sequence, then the monitor defines some sort of expression in which every single term can be matched by an event in the sequence. Recalling the above example, a pattern that monitors three "alarm" events can be expressed by a catenation of three filters, each one selecting "alarm" events. In designing or assessing the monitoring capabilities of an event service, we should consider the relations that can be defined among events to order them into a *stream* as well as the language that defines the observable sequences of events.

2.2.2 Goals: scalability and expressiveness

We believe the new challenges for wide-area event services can be synthesized into two conflicting aspects: *scalability* and *expressiveness*. It is the tension between scalability and expressiveness that introduces the major difficulties in designing a wide-area event service.

2.2.3 Scalability

We say a system is scalable when it is able to grow gracefully. In particular, we refer to all the implications of moving an event service from a local-area network to a wide-area network. These are the most significant requirements subsumed by scalability:

- *vast numbers*: scaling up an event service involves serving more and more applications and components from many sites, dispatching a vast number of events;
- *distance and network robustness*: on a wide-area network, distance becomes a very relevant factor. The most immediate effect on the event service is that bandwidth becomes scarce, latency becomes noticeable, and the reliability of the network decreases. In general, we ought to think of the network as a precious resource;
- *heterogeneity*: when scaling up outside the boundaries of a group or an organization, it is no longer legitimate to assume a common platform. This requires the event service to make minimal assumptions on the capability of each site or on its conformance to one particular data representation standard or one particular communication protocol. In general, it is good to design an event service that is independent of any particular protocol and thus can "speak" many different ones;
- *openness*: wide area networks change continuously in an unpredictable way (see [61]), thus, the event service should be able to accommodate extensions and somehow evolve together with the network;

- *de-centralized architecture and control*: the Internet is characterized by many separate domains managed by different authorities. In this scenario it is not realistic to design a service that requires centralized coordination. The event service should be designed to allow autonomous control and de-centralized operations.

2.2.4 Expressiveness

Expressiveness refers to the main functional properties of the event service assessed from the viewpoint of applications. An event service is expressive and flexible when it has a rich notification model and when it has the ability to observe notifications with a high degree of accuracy. A flexible event service gives the application programmer a versatile schema for modeling data as well as fine selectivity in accessing data of interest.

Expressiveness translates into the following requirements:

- *structured notifications* we believe that notifications must exhibit some form of structure. With this structure, it should be possible to associate heterogeneous information to an event. Notifications may transport additional "black-box" information. Each piece of information should be accessible separately. It is not strictly necessary to adopt a type system for notifications, however it must be possible to encode the most common types such as numbers, strings, and dates. If a type system is not explicitly imposed, a reference encoding schema for the most common types must be defined as part of the event service;

- *expressiveness of filters* the scope of subscription filters should cover the whole structured part of a notification. Primitive predicates must include at least the most common equality and order relations for all the common types (or encodings) as well as some form of wild-card matching function for strings. The usual boolean operators should be provided.

- *patterns* we believe that an event service should provide some sort of pattern monitoring facility. An example of simple classes of patterns is the temporal sequence.

2.2.5 Trade-offs

From what we listed above, it is quite clear that neither expressiveness nor scalability pose significant difficulties if requested singly. Satisfactory results of programming languages and database research solve the problems related to the modeling, encoding, filtering, and monitoring of data elements. Similarly, as far as scalability is concerned, several techniques have been developed by the network community to cope with scalable multi-cast communications.

It should be also clear that expressiveness and scalability are conflicting features. In fact, techniques that optimize the dispatching of notifications to multiple recipients adopt very simple addressing schemas that are not suitable for expressing sophisticated filters. On the other hand, highly expressive data models with rich subscription languages are better realized with centralized engines that do not scale well. We believe the real challenge for the design of an event service is to offer an expressive interface while assuring good scalability to wide-area networks. We found that none of the system that we have examined satisfies these two requirements together.

Chapter 3

The SIENA Event Service

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

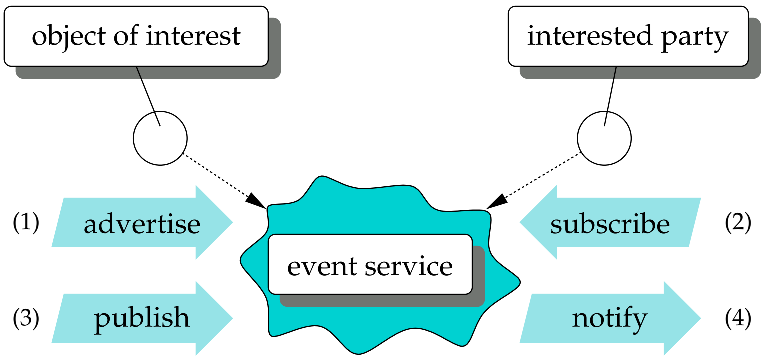

Figure 3.1: Event service

Figure 3.1 shows the high-level architecture of an 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 consequently the event service will take care of delivering the notifications to the interested parties that subscribed for them (4).

This chapter gives a formal definition of the interface and the semantics of the event service realized by SIENA. We will initially define some basic concepts and terms that we will use for our models, then we will define the notification model adopted in SIENA, the interface functions exported by SIENA, and the semantics of SIENA, i.e., its behavior in response to various combination of calls to the interface functions.

3.1 Basic principles

The terms used in this thesis, in particular the terms *notification*, *object of interest*, and *interested party*, follow the framework proposed in [67]. Since we are modeling software systems that are distributed over a computer network, we need some abstractions for software components, network nodes, communication, and in general we have to model the interaction between components. Here we define the elements of our model.

3.1.1 Objects

With the generic term *object*, we denote applications, components, and every active or passive entity inside or outside the event service. Examples of objects are processes, threads, files, servers, web pages, organizations, etc. Objects have an identity and a location on the net. Without loss of generality, we will always model *active* objects, i.e., objects that communicate autonomously. Passive objects, such as files, can participate in an event-based interaction by means of other active objects that act as *proxies* (or *probes*). A proxy continuously monitors the state of one or more passive objects and notifies events on their behalf. For passive objects that are "wrapped" by a proxy, we will consider the proxy to be the object we model and we will ignore the real passive object.

We call *interested parties* and *objects of interest* those objects that produce and consume events respectively. One application can react to events and notify events, thus playing both roles at the same time, however for simplicity, we will always model interested parties and objects of interest as separate objects. Interested parties and objects of interest are clients of the event service therefore they are external to the event service.

3.1.2 Event Servers

The objects that compose the event service are referred to as *events servers*. An implementation of SIENA is realized by one or more interconnected event servers. An application, either an interested party or an object of interest, contacts the event service via one event server. This event server is said to be the application's *access point*.

Figure 3.2 shows an implementation of SIENA and some applications.

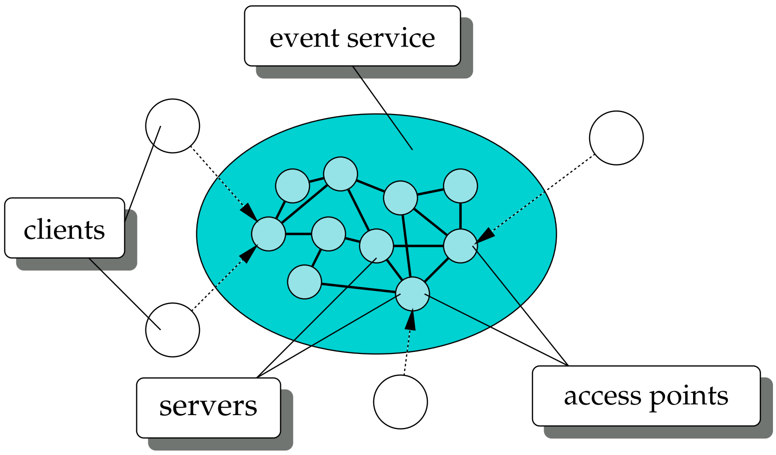

Figure 3.2: Internal architecture of the event service

3.1.3 Network

We assume that objects are connected by means of some communication facility that we refer to as *communication network* or simply *network*. The primitive unit of data that can be exchanged through the network is called a *message*. The primitive communication mechanisms are *send*, that takes a message as a parameter, and *receive* that returns a message. Sending a message to an object is also equivalent to invoking an asynchronous method called *process-message* on that object with a message as a parameter.

Clearly, this network model is very simplistic because here it serves only modeling purposes. This model subsumes a real communication infrastructure, possibly built over a datagram network (e.g., IP) with a high level reliable connection-oriented transport protocol (e.g., TCP), and with higher level protocols that realize the method invocation facility among objects with the proper encoding of parameters (e.g., HTTP, RPC, RMI).

3.1.4 Events and notifications

The concept of *event* seems to be quite straightforward, however rather different definitions have been proposed in literature. For example, in GEM [49] “An *event* is a happening of interest, which occurs instantaneously at a specific time”. Another definition given by [67] characterizes an event as “the instantaneous effect of the (normal or abnormal) termination of an invocation of an operation on an object”. Both definitions agree on giving no duration to events, but while according to the first one, events exist because some entity is interested in them, the second one defines events independently of any interested party. The second definition also subsumes an object model while the second one is neutral with respect to the model adopted for entities. Other systems less explicitly adopt other different definitions.

Being a modeling abstraction, an event might be given several different forms and semantics. Some of the critical aspects regarding the event model include:

- *duration*: it is not clear whether events have a duration or they are instantaneous happenings. We believe the two models are fundamentally different, but equally expressive. Events with duration can simply represent instantaneous events with a null duration, while events with duration can be modeled with two instantaneous events representing the beginning and the end respectively;

- *binding with object of interest*: as we have seen, events can be modeled in terms of the object of interest to which they relate or else they can be modeled irrespectively of their origin. There are events that are strongly related to an object of interest, for example, an event can be generated when a printer is out of paper. This event is clearly meaningful only if modeled together with the printer. On the other hand, there are events that can be reasonably modeled without a reference object, for example a clock tick event might be generated independently of any particular clock object;

- *event occurrence and observation*: some systems model the *occurrence* of events whereas other systems model the *observation* of events. In the first case, events exist and are modeled regardless of the existence of any object that detects their occurrence. In the second case, those events that are not observed simply do not exist. We believe that the first approach is logically more complete, however the difference is more profound from a philosophical viewpoint and in practice the two approaches are substantially equivalent.

- *information*: this determines how much contents can be associated with an event. Note that from some definitions, including the ones that we cited above, it is not clear whether or not events contain any information at all. A minimal piece of information is always embedded in the event notification, it is the name or type of the event itself, i.e., some sort of identifier by which applications can distinguish "stock exchange variation" events from "failed login" events from "printer out of paper" events and so on. In practice, events can be used to integrate software components only if they can encapsulate some information, thus many systems allow objects of interests (or their proxies) to add extra information concerning the event.

Ultimately, a precise characterization of an event involves philosophical issues and it is still subject of debate. In this work, we simply avoid discussing the event model, so we never model events themselves, but instead we refer to notifications that are their "physical" representation. In SIENA, *notifications* are the primitive elements. Notifications have a precisely defined structure and they are explicitly published by objects through the SIENA interface. In other words, an event is whatever an object publishes and the event model is what can be represented by a notification.

3.1.5 Identifiers and handlers

In order for interested parties, objects of interest, and other objects to communicate, a *naming* scheme must be adopted whereby objects can be uniquely identified, and a *handling* scheme must be adopted so that objects can be contacted using appropriate communication protocols. For every object,

- an *object identifier* X determines its identity; and
- a *handler* X_h represents the address used by other objects when sending messages to X.

In general, an object has exactly one identifier and one or more handlers. Also, while the identifier remains bound to an object, handlers may change during the existence of the object.

SIENA adopts the generic URI [5] form for *both* its naming and handling scheme. Also in SIENA, every object has exactly one URI that serves the functions of identifier *and* handler ($X \equiv X_h$). Note that handlers in the form of URIs determine a sort of "extended" address that includes the location of the object and the protocol used to communicate with it. 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*.

SIENA recognizes the most common URI schemas, including *mailto* and *http*, and thus implements the communication protocols implied by each schema. In any case, the handler schema is open to accommodate new communication protocols. SIENA defines and maintains the URIs corresponding to event servers, i.e., its internal components, but it does not directly assign or maintain URIs for interested parties or objects of interest. Such URIs are operated by clients themselves. This means that if an interested party 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.

3.2 Syntax of the SIENA event service

3.2.1 Interface functions

The SIENA event service exports the functions listed in Table 3.1.

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

Table 3.1: Interface functions of SIENA

Intuitively, **publish**(n) allows an object of interest to generate an event notification n and to make it visible to other applications, **subscribe**(X, p) states that object X is interested in being notified of events matching pattern *p*, and **unsubscribe**(X, p) declares that X is not (any more) interested in receiving notifications of pattern *p*. This far the interface of SIENA is pretty much equivalent to that of most event-based infrastructures. In addition to these functions, SIENA exports two more functions that are the dual counterpart of subscribe and unsubscribe. They are *advertise* and *unadvertise* respectively. With **advertise**(Y, q), an object of interest Y declares its intentions to generate notifications matching filter q, and with **unadvertise**(Y, q), the effect of **advertise**(Y, q) is negated.

In SIENA, subscribers and advertisers are explicitly reported as parameters in the interface functions. It is important to do so because we might want to allow objects to send subscriptions to the event service on the behalf of other objects. The publish function does not require an explicit *publisher* because the event service would not use that information anyway.

The rationale for introducing the advertise function is to give more information to the event service so that it can efficiently route subscriptions. In fact, since subscriptions define the potential targets of notifications, they can be used by the event service to direct the routing of notifications. For example, we could instruct the event service to route notifications only towards those objects that submitted matching subscriptions. This policy forwards a notification only if there is an interested party that requested it, thus reducing the traffic and the cost generated by notifications. However, such a policy requires every subscription to be propagated to every event server. Instead, by having objects of interest advertise their intentions, we let SIENA know about the potential sources of events, this way allowing it to prune the propagation tree of subscriptions only to those subnets that contain object of interest that generate relevant events. A complete discussion on the notification algorithms will be developed in Chapter 4.

In the following sections we present the syntax and the semantics of the interface functions of SIENA by formally defining *notifications*, *filter s*, and *patterns* and their role in every interface function.

3.2.2 Notifications

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 3.3 represents a stock price variation event.

string	*event = finance/exchanges/stock*
time	*date = Mar 4 11:43:37 MST 1998*
string	*exchange = NYSE*
string	*name = Walt Disney Co.*
string	*symbol = DIS*
float	*prior = 105.25*
float	*change = -4*
integer	*volume = 2260600*
float	*earn = 2.04*

Figure 3.3: Example of a notification

In SIENA, attributes are uniquely identified by their name. Attribute types belong to a limited set of types. For these types, a fixed set of operators is also defined. Types and operators are an integral part of the definition of the SIENA event service. The types defined by SIENA are: *char*, *integer*, *boolean*, *float*, *string*, *byte-array*, and *date*[1]. The operators are the (any) operator that matches any value, the equality (=) and order relation (>), that have their usual intuitive semantics[2], plus the string prefix operator ($> *$) and string postfix operator ($* <$). The prefix and postfix operators for strings have the meaning of "begins with" and "ends with" respectively, so for example "www.elet.polimi.it" $>*$"www" and "www.cs.colorado.edu" $*<$"edu".

We use a simple "dot" notation to refer to name, type, and value of an attribute. So, if α is an attribute of a notification, α*.name*, α*.type*, and α*.value* denotes its name, type, and value respectively. For simplicity and when there is no ambiguity, $\alpha.value$ can also be abbreviated with α.

[1] We will not go into details with a formal definition of these types. We can assume the standards defined for SQL

[2] The ordering relation with strings implements the lexicographical ordering.

3.2.3 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 3.4: Example of an event filter

Figure 3.4 shows a filter that selects negative stock price variations for a specific stock on a specific exchange. More formally, a filter is 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 two or more attribute filters with the same name. For an attribute filter ϕ, $\phi.match\_op(Op_1, Op_2)$ denotes the application of the operator defined by ϕ to operands Op_1 and Op_2.

Intuitively, a filter can be seen as a query-by-example mask to select notifications based on their contents, however it should be clear that the previous definitions are just *syntactic* data schemas that will be given a precise semantics depending on the interface function in which they are applied.

3.2.4 Patterns

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

An example of a pattern that combines two filters into a sequence is shown in Figure 3.5. 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. Syntactically, a pattern is equivalent to an *expression* in any Algol-like programming language. The elementary terms of a pattern are filters, so a single filter is itself a pattern and any one, two, or more patterns can be combined with unary, binary, n-ary operators to form another pattern.

We say that a pattern is *simple* when it contains only one event filter and no operators. Patterns that are not *simple* are also called *compound*. Also, since subscriptions submit patterns to the event service, we extend the same terminology to subscriptions, thus we say that a subscription is *simple* or *compound* when it requests a simple pattern or a compound pattern respectively.

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

and then

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

Figure 3.5: Example of a pattern of events

Again, note that these are syntactic definitions. Later we will present some operators and we will discuss the semantics of the patterns that they generate.

3.3 Semantics of the SIENA event service

3.3.1 $\sqsubset$: *covering* relations

In order to give the precise semantics of the event service, we must introduce and define the concept of *covering* (or *compatibility*) relation between notifications and subscriptions, and between subscriptions and advertisements. The compatibility between notifications and subscriptions defines the semantics of subscriptions. Since the main job of the event service is to decide whether or not notifications match subscriptions, this compatibility plays a fundamental role. 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 we extend the compatibility relations to compound subscriptions.

3.3.2 $\sqsubset_f^n$: semantics of attribute filters

Firstly, let us define the semantics of an attribute filter. We indicate that an attribute α matches an attribute filter ϕ with the relation:

$$\phi \sqsubset_f^n \alpha$$

defined as follows:

$$\phi.name = \alpha.name \wedge \phi.type = \alpha.type \wedge \phi.match\_op(\alpha.value, \phi.value)$$

This expression says that an attribute filter covers an attribute when both have the same name and the same type, and when the value of the attribute matches the value of the filter according to the operator defined by the filter.

Note that here and in the following sections, we will use different expressions to denote some type of *covering* relations. For any two entities x and y, these expressions and symbols are interchangeable:

- $x \sqsubset y$
- x **covers** y
- y **matches** x
- y **is compatible with** x

3.3.3 $\sqsubset_S^N$: semantics of subscriptions

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

$$\textbf{Covers}_S^N \subseteq \mathcal{S}_0 \times \mathcal{N}$$

For brevity, we represent the relation $\textbf{Covers}_S^N$ with the symbol '$\sqsubset_S^N$'. Consistently we say that whenever $s \sqsubset_S^N n$, the notification n is compatible with or matches the subscription s. We denote with $N_S(s) \subseteq \mathcal{N}$ the set of notifications n covered by s.

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

$$N_S(s) = \{n \in \mathcal{N} : \forall \phi \in s : \exists \alpha \in n : \phi \sqsubset_f^n \alpha\}$$

This formula mandates that every attribute in the subscription matches the corresponding attribute in the notification. The correspondence is based on attribute names. If the subscription contains more than one attribute with the same name, the matching rule applies to all of them. The notification may also contain other attributes that have not correspondents in the subscription. Table 3.2 contains some examples of pairs of notifications and simple subscriptions that show the meaning of $\textbf{Covers}_S^N$.

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

Table 3.2: Examples of $\mathbf{Covers}_S^N$

3.3.4 $\sqsubset_A^S$: semantics of advertisements

We 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 let $a \in \mathcal{A}$ be an advertisement. We define the set of notifications $N_A(a)$ *covered* by a:

$$N_A(a) = \{n \in \mathcal{N} : (\forall \alpha \in n : \exists \phi \in a : \phi.name = \alpha.name) \\ \wedge (\forall \alpha \in n : \forall \phi \in a : \phi.name = \alpha.name \Rightarrow \phi \sqsubset_f^n \alpha)\} \quad (3.1)$$

This says that an advertisement a covers all the notifications n that have a set of attributes that is *included* in the set of attributes of the advertisement. Also attributes in n and a that have the same name must also have matching values. A notification that has an attribute that is not present in the advertisement is not covered by that advertisement.

Given the definition of $N_A(a)$ we can easily define $\mathbf{Covers}_A^S$ ($\sqsubset_A^S$ for short), the covering relation between advertisements and subscriptions:

$$\sqsubset_A^S \subseteq \mathcal{A} \times \mathcal{S}_0$$

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:

$$s \sqsubset_A^S a \Leftrightarrow N_A(a) \cap N_S(s) \neq \emptyset \tag{3.2}$$

This says that an advertisement a covers a subscription s if the set of notifications defined by a, $N_A(a)$, includes at least one notification that is also covered by s. Consistently we say that s is compatible with a, that in this case can be interpreted also as "a is relevant for s", or else "a advertises something that might be of interest for s". See Table 3.3 for some examples.

advertisement		*subscription*
string event = alarm *time* date *any* *integer* level > 0	$\sqsubset_A^S$	*string* event = alarm *integer* level > 3
string event = alarm *time* date *any* *integer* level > 0	$\not\sqsubset_A^S$	*string* event = alarm *integer* level > 3 *string* user *any*
string event = alarm *time* date *any* *integer* level > 0	$\sqsubset_A^S$	*integer* level > 5
string event = alarm *time* date *any* *integer* level > 0	$\not\sqsubset_A^S$	*integer* level *any*

Table 3.3: Examples of $\mathbf{Covers}_A^S$

3.3.5 Behavior of the service

We discuss the semantics of SIENA by defining the expected behavior of SIENA 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.

3.3.6 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, this means that the event service will send a notification n to an object X if and only if X has sent at least one subscription s that covers n and that has not been canceled by a subsequent unsubscription.

Summing up, the event service will deliver a notification n to an interested party X *if and only if*:

1. X subscribes for s; *and*

2. $s \sqsubset_S^N n$.

Note that if these conditions are not satisfied *at the time* the notification n is published, the event service will not complete the delivery. In particular, if X issues the subscription s after n has been published, then X will not be notified on the occurrence of n.

3.3.7 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. $a \sqsubset_A^S s$; *and*

4. $s \sqsubset_S^N n$.

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. Again, here the conditions must be matched at the time Y publishes n. However we do not set any temporal precedence between subscriptions (X subscribes for s) and advertisements (Y advertises a). This implies that the conditions that determine the delivery of a message can be met either way.

3.3.8 Un-subscriptions and un-advertisements

The above definitions would not be complete without a more formal specification of the effects of unsubscriptions and unadvertisements. As we said, an unsubscription (unadvertisement) cancels one or more *corresponding* subscriptions (advertisements).

Given a simple unsubscription **unsubscribe**(X, f_{us}) where X is the identifier of an interested party and f_{us} is an event filter, the event service cancels all the simple subscriptions **subscribe**(X, f_s) submitted for the same interested party X with a subscription filter f_s *covered* by f_{us}. An unsubscription covers a subscription when the set of notifications covered by its filter f_{us}, interpreted as a subscription, includes the set of notifications covered by the subscription filter f_s, i.e., when $N_S(f_{us}) \supseteq N_S(f_s)$.

A similar rule applies to advertisements and unadvertisements, so the simple unadvertisement **unadvertise**(X, f_{ua}) cancels all the advertisements **advertise**(X, f_a) issued for the same object of interest X with a filter f_a that defines a set of notifications included in the set of notifications defined by f_{ua} interpreted as an advertisements, i.e., when $N_A(f_{ua}) \supseteq N_A(f_a)$.

3.3.9 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. The composition of notifications to form patterns is defined upon the temporal sequence of notifications. Each notification is implicitly time-stamped at the time it is published, so a pattern is a *string* of notifications with increasing time stamps from left to right. When two or more notifications are combined in a pattern or when two or more patterns are combined into a bigger pattern, the new pattern inherits the time stamp of the rightmost notification. In any case, the combination of two patterns must produce a time-ordered string of notifications.

More formally, given the subscription **subscribe**(X, P), with pattern filter $P = f_1 \cdot f_2 \cdots f_k$, being "$\cdot$" the *sequence* or *concatenation* operator, SIENA tries to assemble a sequence of notifications $n_1, n_2, \ldots n_k$, published at time $t_1, t_2, \ldots t_k$ with $t_1 \leq t_2 \leq \ldots \leq t_k$, each one matching the corresponding filter in P, thus:

$$\bigwedge_{i=1}^{k} f_i \sqsubset_S^N n_i$$

A pattern filter P determines the *alphabet* $\Sigma(P)$ against which the string of notifications is matched.

$$\Sigma(P) = \bigcup_{i=1}^{k} N_S(f_i)$$

All the notifications in $\Sigma(P)$ are evaluated in assembling a pattern that matches a pattern filter P. No other notifications are taken into account. More formally, a sequence $n_1, n_2, \ldots n_k$, published at time $t_1, t_2, \ldots t_k$ matches a pattern filter $P = f_1 \cdot f_2 \cdots f_k$ only if for every pair of notifications n_i, n_{i+1} matching f_i and f_{i+1} respectively, there is no notification $n'_i \in \Sigma(P)$ published within the interval (t_i, t_{i+1}).

The syntax of patterns filters in SIENA allows any kind of operators for combining sub-patterns, however, at this point, SIENA supports only the sequence operator $(\cdot)$. The implementation of other operators including the *or* (|) and the Kleene's closure (*) might be added in the future.

As far as the semantics of compound subscriptions and advertisements, 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 (or parser) that assembles sequences of notifications, each one matching one of the elementary components according to the semantics of the combinators[3]. 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. Similarly, compound unsubscriptions and unadvertisements will cancel those compound subscriptions and advertisements for which the internal simple filters would be canceled, one by one, by the corresponding filters.

3.4 Other semantic aspects of the event service

There are some factors that influence the behavior of the event service apart from the semantics defined by the covering relations. The main aspects are

[3]Note that it is always possible to implement patterns *outside* the event service by breaking them into a sequence of simple subscriptions and by setting up a single monitor on the access point of the subscriber

related to *time* and *quality of service*. Another very important issue that we will comment on in this section is the *mobility* of objects.

3.4.1 Time and ordering of events

Timing issues might arise when considering unsubscriptions and unadvertisements. For example, an interested party may send an unsubscription when

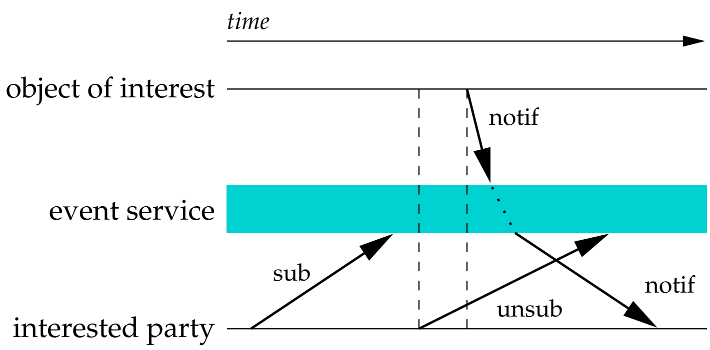

Figure 3.6: Race conditions in event notification

some notifications have already been sent to it (see Figure 3.6). In this case, the interested party will receive undesired notifications. This sort of race conditions is pretty much inevitable in our network model due to the finite transmission speed of communication links.

Other more critical timing issues regarding the ordering of notifications can arise depending on the topology of the network. Typically we might have the scenario depicted in Figure 3.7. Here two notifications $notif_1$ and $notif_2$ published by objects O_1 and O_2 respectively at time t_1 and t_2 with $t_1 < t_2$, might be delivered to the same destination P through two different routing paths with different latencies. In some cases, the event service would deliver $notif_1$ and $notif_2$ at time $\bar{t}_1$ and $\bar{t}_2$ with $\bar{t}_1 > \bar{t}_2$, thus creating potential problems for those applications that rely on the temporal ordering of events. Note that this constitutes a major problem also for the event service itself because the semantics of patterns is sensitive to the temporal ordering of notifications.

In SIENA we assume that the event service is able to examine events in the right time order. Ordering events with respect to time is a classical problem in distributed systems [45] for which there exist well known algorithms (see [28, 63]). In practice, this assumption requires that the event service buffer notifications and shuffle them in the correct temporal sequence within a finite time. Thus, the underlying assumptions are:

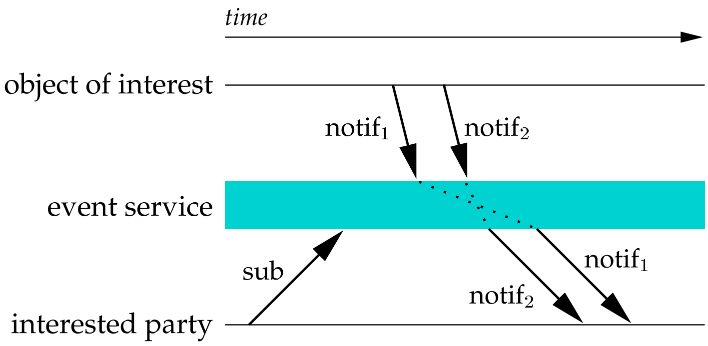

Figure 3.7: Temporal ordering of notifications

1. the existence of a global clock used to time-stamp notifications,
2. an upper bound for the network latency and the network diameter, and
3. sufficiently big communication buffers

The availability of high-resolution GPS services and extremely accurate synchronization protocols [46] together with the Internet Network Time Protocol [51, 50] make the first assumption very reasonable for most practical applications. Instead, the latter requirements can pose serious engineering trade-offs. Clearly these trade-offs might be different for different realizations of SIENA, typically a centralized service would be less sensitive to network latency than a distributed service where events are routed through long chains of servers and where the problems of out-of-order notifications and latency increases at every hop. Note once again that the existence of these trade-offs is inherent in the communication mechanisms underlying the event service. In other words, the event service can not do better than the communication mechanisms that it uses.

3.4.2 Quality of service

By quality of service (QOS) we refer to a number of non-functional properties that do not directly affect the semantics of the event service, but that are nonetheless of fundamental importance for its practical realization and usage. The interface of SIENA that we presented so far is just the core set of functionalities. A real implementation must deal with QOS settings such as authentication and security, and transactional communications. Much of the discussion pertaining quality of service is out of the scope of this thesis, so we will only comment on some possible extensions to the existing model by showing the features that can be used as placeholders for QOS parameters. In any case, we will not examine solutions that require structural changes to the architecture of SIENA.

3.4.3 Mobility of applications

Wide-area networks are not just communication facilities. Recently, research in code mobility has produced a number of languages and systems that make the network a complete computation infrastructure in which components can move from one site to the other even during their execution [30]. It is desirable that an event service support the integration of such mobile objects as well as the conventional "steady" objects.

The fact that an object can move affects the event service because that object might change its location while maintaining its identity. Also, when moving to its new execution site, the object might want to connect to a different access point, possibly local that new site. Supporting mobility means allowing this sort of operations without imposing extra requirements on objects. There are at least three different approaches to support mobility:

transparent this is the case of objects addressed by URIs that hide their location and their mobility. Such URIs use network-level mechanisms to transparently manage mobility or intermittent connectivity of objects (see [62, 37]). In this case, the objects is treated exactly like every other "steady" object.

native here the event service uses only "steady" URIs that also contain information about their location (examples are the *mailto* and *http* schemas), but it implements internal mobility mechanisms possibly exported through new interface functions. This approach has been adopted by JEDI [19] that exports the functions shown in Table 3.4[4]:

move_out(URI *orig*)
move_in(URI *orig*, URI *final*)

Table 3.4: Mobility support functions.

With **move_out**, an object declares that it intends to move from its original identity/location *orig*. With **move_in**, an object declares its new identity/location. Note that it is likely that a call to **move_out** and its corresponding **move_in** are issued to two different access points. In practice, **move_out** requests that the event service *suspends* all the notifications to

[4]These interface functions have been slightly changed from their original definition in JEDI

an object while **move_in** requests that these notification be resumed possibly somewhere else on the net.

What makes this approach difficult to implement are the timing issues discussed in Section 3.4.1. In fact, not only the event service must make sure that object "profiles" (their subscriptions and advertisements) are moved to the new access point, but it also has to deliver all the notifications in the right order to the object through its new access point. These are:

- notifications that have been published before the object moves, but that have not yet been received by the object at the time it leaves its original access point because of network latency;
- notifications that are published after the object leaves its original access point and before it connects to the new access point. In JEDI, a component can request that these notifications be stored by the event service during the time the object is "traveling";
- notifications that are published after the object connects to the new access point.

external in this case, mobility is dealt with outside the event service by adding an extension layer between the event service and mobile objects. This layer records all the subscriptions and advertisements and implements the move_out function by buffering all the notifications, and the move_in function by unsubscribing the old URI and re-subscribing the new URI. This layer also sets up a forward mechanisms that re-sends all notifications to the new URI. The forwarding mechanisms must also include a filter that drops possible duplicates. Duplicate notifications can be received because, in order not to loose any event, the re-subscription from the new URI must be issued before the unsubscription from the original URI. This interval and the natural latency of the network may cause some notifications to be routed towards both destinations. The forward can be eventually dismissed after an amount of time that depends on the diameter of the network.

This approach is quite simple, but clearly it does not preserve temporal order of notifications, also it duplicates information that are already available to the event service.

We believe that, in the framework of SIENA, mobility poses a whole set of issues that remain open. Especially considering its impact on scalability. With its interface, SIENA is obviously open to use transparent mobility and it also permits the implementation of external mobility, however SIENA does not provide native support for mobile objects.

3.5 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 thesis, 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 and qualitative conjectures to justify the fact that the semantics of SIENA satisfies the first requirement. In the following sections we will comment on these aspects. Instead, we will elaborate more on the second requirement 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, also showing why other semantics, albeit slightly different, would not allow such exploitation.

3.5.1 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. We will see in Chapter 4 that event service attempts to minimize the usage of communication and computation resources by deploying filters and patterns monitors in strategic places on the network of event servers. In this optimization process, the compatibility relations play a fundamental role. Hence it is essential that these relations be efficiently implemented.

The elementary relation between an attribute filter and an attribute is a straightforward verification of one of the predefined predicates (*match_op*) that we assume has constant complexity:

$$\phi \sqsubset_f^n \alpha = (\phi.name = \alpha.name \wedge \phi.type = \alpha.type \wedge \phi.match\_op(\alpha.value, \phi.value))$$

Relations that involve a filter and a notification are also easy to implement because they are the conjunction of attribute filters relations:

$$s \sqsubset_S^N n = \bigwedge_{\phi \in s} \exists \alpha \in n : \phi \sqsubset_f^n \alpha$$

Being s and n finite sets of cardinality $|s|$ and $|n|$, the complexity of evaluating this expression is $O(|s| \log |n|)$.

The relations that pose significant problems are clearly the ones that involve two filters (e.g., $\sqsubset_A^S$); in fact, given the semantics defined in Section 3.3, comparing two filters is equivalent to computing the intersection of two possibly infinite sets.

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 form the simple predicates. It is easy to reduce the problem of evaluating a covering relations between two filters f_1 and f_2 to the evaluation of the *implication* between their corresponding attribute filters. Given an attribute filter $\phi_1 = (N, T, Op, V)$ of name N, type T, operator Op and value V, and another attribute filter $\phi_2 = (N, T, Op', V')$ 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:

$$(\phi_1 \Rightarrow \phi_2) \Leftrightarrow \forall x \in T : Op(x, V) \Rightarrow Op'(x, V')$$

To allow an efficient evaluation of this formula, we must have "well behaved" operators and types.

The types and operators that we chose for SIENA are quite predictable and their semantics makes it easy to verify implications between filters. In fact, being '$\triangleright$' any one of the ordering relations we defined ($\geq$, $\leq$, $>*$, or $*<$) we can reduce every relation between two attribute filters to the following cases:

- $\forall x : x = V \Rightarrow x = V' \quad \text{iff} \quad V = V'$
- $\forall x : x \triangleright V \Rightarrow x \triangleright V' \quad \text{iff} \quad V \triangleright V'$
- $\forall x : x = V \Rightarrow x \triangleright V' \quad \text{iff} \quad V \triangleright V'$

3.5.2 Possible counter-intuitive behavior

In both advertisement-based and subscription-based semantics, objects can publish all sorts of notifications regardless of what they advertise and, similarly, there are no constraints on the subscriptions that an interested party can post to the event service. This means that those operations will always succeed even though they will never result in any notification because of incompatibilities.

In several discussions with our colleagues, this has been regarded as a counter intuitive, if not incorrect behavior. For example, it has been argued that it does not make sense to subscribe for something that has not been advertised. Or else, that a forgiving event service that does not block the publication of notification that have not been advertised would be misleading for application designers. The approach we have taken in SIENA is to constraint the behavior of clients only if this is strictly necessary for the task of the event service, especially keeping in mind that our design priorities are scalability and expressiveness. In other words, those consistency checks that can be confined to a single client or to a single access point, and that do not add information useful for the routing algorithms, should be performed outside the event service.

With respect to the specific case mentioned above, note that it is always possible to restrict the advertisement-based semantics by imposing a publication guard that blocks notifications that have not been properly advertised. Such a modification impacts only data structures and algorithms at the event service access point, thus not constituting a fundamental factor of scalability. These modifications could be just as well provided as an external layer.

3.5.3 Typed vs. untyped event service

The *type system* used for the event model, and consequently for filters, is subject of debate too. There are two conflicting aspects that influence the decision about the type system to use. They are once again a variation of the scalability-functionality trade-off. Let us consider these three options:

- *no types*: in this case, every piece of information has the same generic structure. For example, everything could be represented as a string (examples are scripting languages like Tcl and other command interpreters). In order to realize some filtering with the semantic that we defined, the event service would have to define some operators that match two strings interpreting them as integers, strings, dates, etc. So, a filter might look like this: $\mathit{birthday} >_{\mathit{date}}$ *"Dec 16"*;

- *simple predefined types*: this is the case that we described in SIENA. The event service understands some types and therefore it is able to overload comparison operators;

- *abstract user-defined types*: in this case, the event service would provide the features of a typed programming language that allows the definition of abstract data types (e.g., an object-oriented language). Extending filters to user-defined types can be done with easy extensions to the filtering language similarly to query languages for object-oriented databases like O_2SQL or OQL. A filter might look like this: *person.birthday.greaterThan*(*Date*(*"Dec 16"*));

From the viewpoint of the application programmer, the latter solution would be best for its enhanced modeling features. However, because filters could contain an arbitrary user-defined code, it would be very difficult to perform optimizations based on the static analysis of filters. This would severely impact on scalability. Also, on a large scale, it might not be legitimate to assume that every component of the system is able to understand an object oriented language.

From the viewpoint of the designer of the event service, the first solution is definitely the easiest one to implement. It poses less problems of encoding data and it is also the most open one because it requires a simpler common representation (e.g., the standard for Internet electronic mail messages [17] could be used), that is therefore more portable on heterogeneous platforms. Optimizations would well be possible given the semantics of the operators that the event service defines.

We decided to adopt the solution that is in the middle because it preserves the predictability of operators and thus the possibility to optimize the dispatching based on relations between filters, together with the benefits of having more readable filters and some type checking.

Chapter 4

Servers Topologies and Algorithms

As we have suggested in the previous chapters, SIENA is architected as a distributed system in which the distributed components, the event servers, cooperate to realize a unified network-wide event service. This chapter covers the architecture of SIENA. In particular, we will describe:

server topologies the kind of interconnections existing among event servers and the pattern they form. The kind of connection between two servers determines the interaction protocol and thus the type of information that can flow in each direction;

event dispatching algorithms these implement the routing of notifications. Since the routing of notifications is controlled by subscriptions, unsubscriptions, advertisements, and unadvertisements, the dispatching algorithms must take care of disseminating these messages too. Different algorithms adopt different strategies and require more or less complex data structures and computations on every event server. The dispatching algorithms also implement the monitoring of event patterns.

We will see that different servers topologies pose different constraints on the dispatching algorithms and vice versa. Also, we will show how a distributed realization of SIENA is similar to an infrastructure for multicast routing on a datagram internetwork such as Internet. This analogy, and in particular the study of the IP multicast structure and algorithms, will allow us to achieve a better understanding of the challenges for our dispatching algorithms.

4.1 Server Topologies

A *centralized* topology is composed of a single event server with its clients. The protocol used between a client and the event server is derived from the interface of the event service (see Figure 3.1 on page 21). When designing an architecture with multiple servers, we must also introduce:

1. some connections among servers, that determine which pairs of servers communicate directly, and

2. the protocol used in communications between servers.

These two elements define a topology of servers.

Note that the connections among servers are not necessarily persistent channels, but rather logical connections. In general, this is also the case with clients. When we say that an object X is connected to another object Y, we mean that X has Y's address and can send messages directly to Y. The "physical" implementation may or may not keep a live connection between X and Y.

4.1.1 Hierarchical

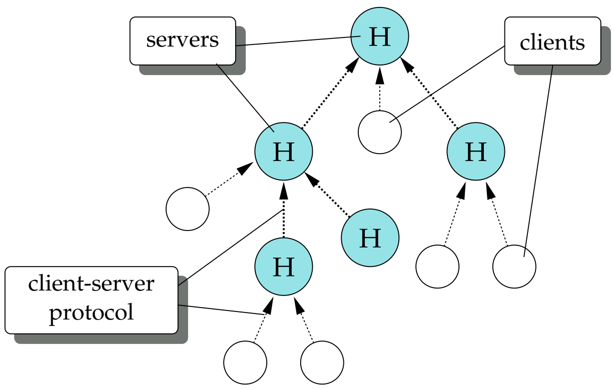

Figure 4.1: Hierarchical server topology

A natural way of connecting event servers is according to a *hierarchical* topology. As shown in Figure 4.1, each server in a hierarchical topology has a number of clients that can be either 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. Note that for every server, the connection to the parent server is the only outgoing arrow.

In this topology, the same client-server protocol is used for server-server communication, thus, a server does not distinguish other servers from objects of interest or interested parties among its 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 back to them.

The hierarchical topology is the natural extension of a centralized topology. It only requires an extension to the algorithm of the server that takes care of propagating information through the parent server. As far as configuration of servers, this topology allows entire subnets to join a community of servers by simply connecting their root servers to any server in the community. This topology is used in the USENET News network, in JEDI, in Keryx, and in TIB/Rendezvous.

The main problems of the hierarchical topology are the overloading of higher-level servers (we will see this effect in more details in Section 5.3.4) and the fact that every server is a critical point of failure for the whole architecture. In fact, a failure in one server disconnects all the subnets reachable from its parent server and all the client subnets from each other.

4.1.2 Acyclic Peer-to-Peer

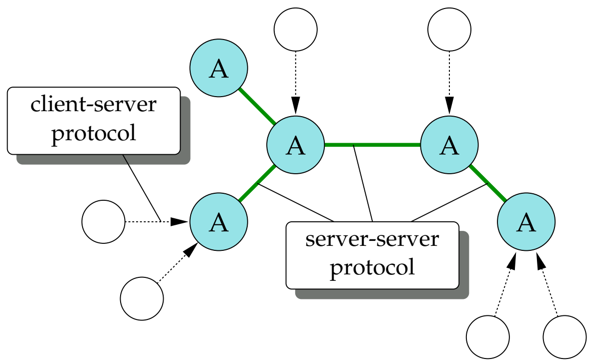

Figure 4.2: Acyclic peer-to-peer server topology

In the *acyclic peer-to-peer* topology, servers communicate with each other as peers, thus adopting a special protocol that allows a bi-directional flow of subscriptions and advertisements as well as notifications. Figure 4.2 shows an acyclic peer-to-peer topology of servers. The different kinds of communication occurring between clients and servers and among servers are denoted with different line styles.

Considering server-to-server links as non-directed arcs, the configuration of the connections among servers produces an acyclic graph. It is important that the procedures adopted to configure servers and to connect new ones maintain this property since specific algorithms might rely on it, e.g., to be sure that any two servers can be connected with only one path. Note that the incremental procedure suggested for the hierarchical topology can be used in this case as well. For example, every server X is initialized with at most one peer server to which X connects. When X is initialized with Y, X adds Y to its direct peers and sends a configuration request to Y. In response to this request, Y adds X to its list of peers. The fact that a server actively sends a configuration request only once, e.g., when it is initialized, assures that the resulting network does not contain redundant paths.

In a network-wide service where each server is administered by a local authority, it might not be a good idea to trust this procedure. Perhaps other control protocols must be implemented to validate the configuration of servers. Also, similarly to the hierarchical topology, this topology suffers from the lack of redundancy in the connection graph. Because the connection graph is a tree, a failure in one server X isolates all the groups of subnets reachable from those servers directly connected to X.

4.1.3 Generic Peer-to-Peer

Removing the constraint of the acyclic graph from the acyclic peer-to-peer topology, we obtain the *generic peer-to-peer* topology. Like the acyclic peer-to-peer,

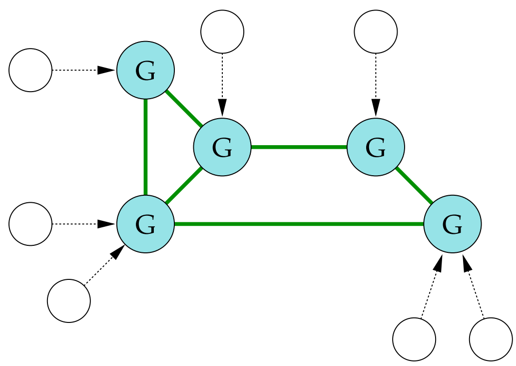

Figure 4.3: Generic peer-to-peer server topology

this topology allows bi-directional communications between two servers, but this time the network of connections among servers is a generic graph, possibly with multiple paths between servers. See Figure 4.3 for an example.

The advantage of this topology over the previous ones is that it requires less coordination and more flexibility in the configuration of connections among servers. Moreover, being redundant in connecting different points in the network, this topology is more robust with respect to failures of single servers. The drawback of having redundant connections is that special algorithms must be implemented to chose the best paths and to avoid cycles. Typically, messages should carry a *time to live* counter and routes should be set up according to minimal spanning trees. Consequently, the server-to-server protocol adopted in the generic peer-to-peer topology must accommodate these extra information.

4.1.4 Hybrid topologies

A network-wide service like SIENA poses different requirements at different levels of aggregation, or in other words, we envision intermediate levels between what is referred as local-area network and a wide-area network. These different levels with their specific requirements lead to hybrid topologies. A hybrid network exhibits different topologies at different levels of granularity.

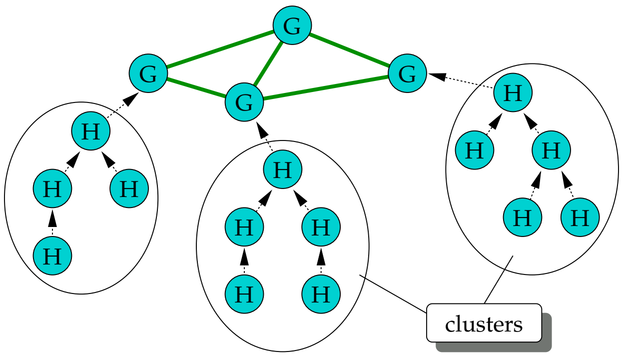

Figure 4.4: Hybrid topology: *hierarchical*/*generic*

For example, in the case of big corporations, it might be reasonable to assume a high degree of control and coordination in the administration of a cluster of subnets. In this case, system administrators might very well be able to design and manage the whole network of event servers that covers their subnets, thus it might be a good idea to adopt a hierarchical topology inside the cluster even when the global network is laid out as a generic topology. Figure 4.4 shows this scenario.

In other cases (see Figure 4.5) we might want to set up SIENA the opposite way. For example, suppose that some clusters of subnets have a very intense traffic of local events, and for some specific applications or perhaps for security

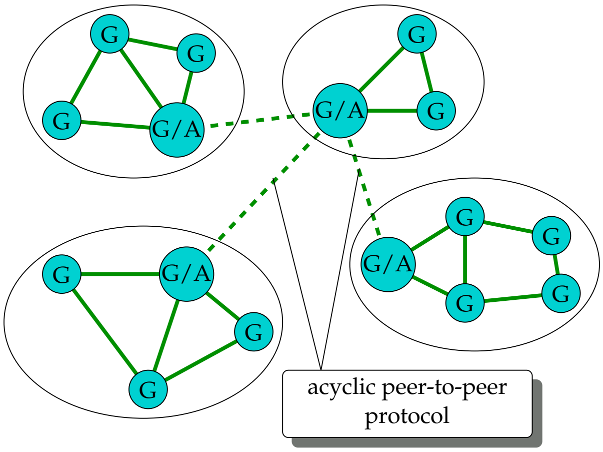

Figure 4.5: Hybrid topology: *generic/acyclic*

reasons, only a small fraction of the events flowing inside the cluster are visible outside the cluster. Here, for efficiency reasons, a generic graph topology might be preferable inside the cluster while the high-level topology could be set up as an acyclic graph. For every cluster, the "gateway" server should be able to filter the messages used for the protocol inside the cluster or adapt them to the protocol used among clusters. For example, if a specific protocol is used to discover minimal spanning trees, its messages will not be propagated outside the cluster by the "gateway" server since every path will inevitably pass through that gateway.

4.2 Dispatching Algorithms

4.2.1 Analogy with multicast routing

The architecture of SIENA is equivalent to the physical architecture of a datagram internetwork. Each event server has a set of local clients that form its *subnet*. Every server is also connected to other servers, i.e., to other subnets. Connections between servers, although they can be dynamically reconfigured, are the equivalent of the physical links between subnets. An event server acts as a gateway for its local clients by connecting them to the rest of the net, but it also functions as a bridge that connects two other servers or subnets. Delivering a notification to all the interested parties means routing this notifications through the network of servers using the routing information set up by subscriptions and advertisements. Since for each notification, there may be numerous interested parties, routing notifications through the event service is conceptually similar to routing datagrams to multiple destinations in a network like the Internet.

The IP multicast infrastructure realizes routing to multiple destinations on the Internet. The model established by Deering [22] extends IP addresses with *group addresses*. These are particular IP numbers that correspond to *host groups* [21]. A host group addresses a set of hosts, possibly sparse over different subnets. A multicast-enabled IP service provides two extra functions: **JoinHostGroup**(group_address g, interface i) that instructs the IP service to deliver to i IP datagrams addressed to g, and **LeaveHostGroup**(group_address g, interface i) that removes the binding created by **JoinHostGroup**. A special protocol called Internet Group Management Protocol (IGMP) is used in IP multicast networks to propagate group membership information. Since any number of hosts can join a host group, datagrams addressed to that group must be dispatched to different points on the network. This is realized with special multicast routing protocol. Several of these protocols have been proposed with different degrees of scalability [23, 24, 55, 77]. These protocols use the membership information set up by IGMP.

IP multicast routing	*event service*
host/interface	interested party or object of interest
router/gateway	event server
physical link	connections between servers
physical network topology	servers topology
IP_send(*datagram*)	**publish**($notification$)
JoinHostGroup(*group*,*interface*)	**subscribe**($handler, pattern$)
LeaveHostGroup(*group*,*interface*)	**unsubscribe**($handler, pattern$)
IGMP multicast routing	dispatching algorithms

Table 4.1: Analogy between event service and multicast routing

Table 4.1 shows the mapping of concepts between multicast network service and event service. This mapping suggests that some models and solutions that already proved to be effective for multicast routing can be proficiently applied to SIENA. On the other hand, it also allows us to identify the differences between SIENA and the multicast routing infrastructure. The main ones that pose new challenges are:

1. handling IP addresses is substantially different from handling event filters with the semantic defined in Section 3.3;

2. the monitoring of patterns of events provided by SIENA has no counterpart in IP multicast.

These differentiators become more evident when examining the main functionality of the two infrastructures. What follows is a very simplistic description of the basic functioning of IP multicast:

routing tables every router has a table whose elements corresponds to group addresses. For every group address g, the table reports interfaces that have attached hosts that are members of g;

routing of datagrams when a router receives a datagram addressed to a group g, it looks up the table and, in case g is in the table, it forwards the datagram to all the interfaces associated with g.[1];

routing of *JoinGroup* information the group membership information is disseminated throughout the network with the IGMP protocol[2]. In practice, when a router receives a **JoinGroup** request for group g, it looks up its table. If g is not in the table, then the router sends a **JoinGroup** request for g to every neighbor router;

routing of *LeaveGroup* information similarly, when the last host or network leaves a group g, the router removes g from its table and sends a *LeaveGroup* message for g to its neighbor routers.

In IP multicast, datagram addresses and group addresses are homogeneous —they are both IP addresses— so all the table look-up operations can be implemented with one function. Moreover, since addresses are totally ordered and both the ordering and the equality relations are trivial to compute, the look-up function can be very efficient. Instead, in SIENA the routing tables are indexed with subscription patterns or sub-patterns while notifications are addressed (implicitly) by their contents. This introduces two difficulties: first, there are two different look-up operations, one is called in routing notifications and finds the subscriptions matched by a notification, the other one is called when routing subscriptions and matches subscriptions against other subscriptions. Second, the covering relations implemented by the two look-up operations are not functions, but they are generic relations, thus, a notification being routed may match zero, one, or more subscriptions, and similarly a subscription being propagated may match any number of existing subscriptions. Note that these relations are the core elements that determine the semantics of the service as well as the routing procedures.

[1]The actual protocols are much more complex. Routers usually avoid routing a datagram d that they receive from an interface that is not the one they would use to route normal (unicast) datagrams towards the source address of d. Anyway, we do not need to go into details here.

[2]This protocol defines a periodic mechanism by which routers exchange group membership information. This information is proactively pulled by routers with generic or group-specific queries. Again we are interested in a more conceptual —and very simplistic— description of IGMP here.

4.2.2 Routing strategies in SIENA

In IP multicast, the main idea behind the routing strategy is to forward a datagram only towards networks that contain members of the target group, possibly using the shortest path tree. In addition to this, in order to save communication and computation resources, SIENA exploits the fact that subscriptions can partially overlap and that one notification can match more than one subscription. These ideas can be summarized in the following two principles:

downstream duplication this says that notifications should be routed in one copy as far as possible and that they should be replicated only downstream, i.e., as close as possible to the interested parties that requested them. Figure 4.6 shows this principle. Two interested parties subscribe

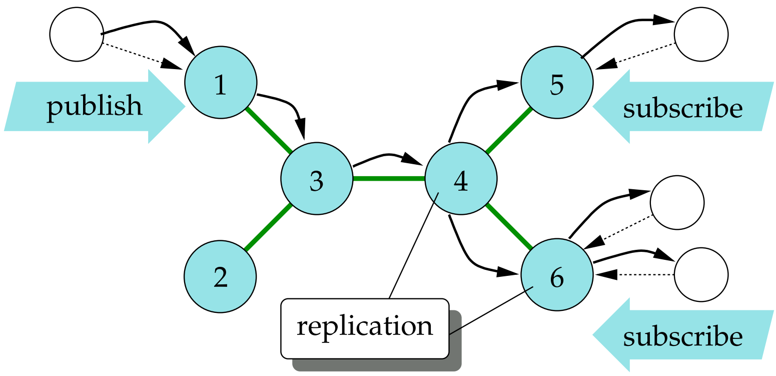

Figure 4.6: Multicasting of notifications downstream

for some patterns on server 6 and another one subscribes for some other pattern on server 5. A notification matching all the subscriptions is published on server 1. The event service routes one copy of that notification through server 1 and server 3 to server 4. Server 4 duplicates the notification sending one copy to server 5 and one copy to server 6. Server 5 forwards the notification to its interested party while server 6 duplicates the notification again to send a copy to each one of its clients.

upstream monitoring this tells us to apply filters and to assemble patterns upstream, i.e., as close as possible to the sources of events or sub-patterns. This principle is shown in Figure 4.7. The example shows one interested party subscribing for pattern XY and two objects of interest on the other side of the network publishing notifications matching X and Y respectively. The event service allocates a filter for X and one for Y on server 1 and server 2 respectively. Then a monitor that recognizes patterns XY is

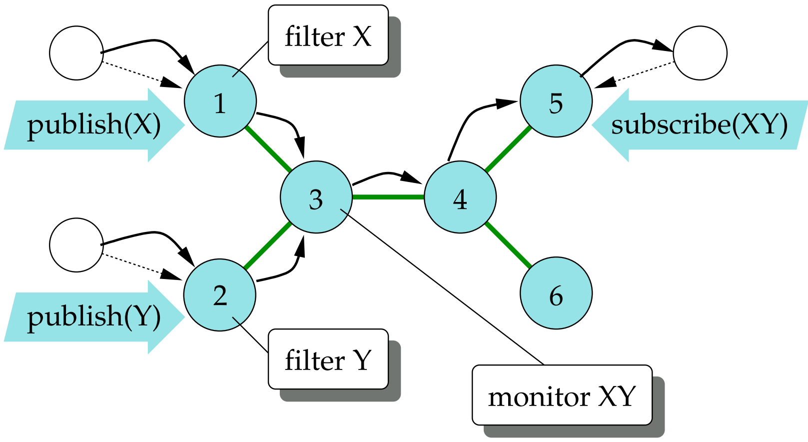

Figure 4.7: Applying filters and patterns upstream

activated on server 3. The pattern assembled on server 3 is then routed towards the interested party through servers 4 and 5.

To implement these principles, we formulate two classes of algorithms: *subscriptions forwarding* algorithms, and *advertisements forwarding* algorithms. Subscription forwarding algorithms realize a *subscription-based* semantics and they apply the *downstream replication* optimization principle[3]. Advertisement forwarding algorithms on the other hand realize an *advertisement-based* event service and apply both the *downstream replication* and the *upstream monitoring* principles.

Subscriptions forwarding

With *subscriptions forwarding*, the routing paths for notifications are set by subscriptions that in turn are broadcasted throughout the network. 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 the reverse path put in place by the subscription.

[3]Section 4.3 explains why *upstream monitoring* can not be implemented fully without an advertisements-based semantics

Advertisements forwarding

Broadcasting subscriptions is necessary only if routing information is available through subscriptions and from nothing else. This leads to the idea of advertisements and to the advertisement forwarding algorithm. If advertisements are enforced, e.g., by adopting an advertisement-based semantics, it is safe to send a subscription only towards those objects of interest that intend to generate notifications that are potentially relevant to that subscription. 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* that path. Notifications are then forwarded only through the *activated* paths.

4.2.3 Putting together algorithms and topologies

In propagating requests, either subscriptions or advertisements, servers maintain tables of subscriptions 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 an no other action is taken. If no such subscription is present in the tables, the new request is added to the table and propagated. This allows to prune entire subtrees in the propagation. For example, in the scenario of Figure 4.6, 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 a compatible subscription to server 4, server 4 stops the flooding.

The broadcasting or *flooding* process that characterizes both subscription forwarding and advertisements forwarding creates per-source minimal spanning trees. The realization of this process depends upon the underlying topology of servers. Clearly 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 [20].

Other important covering relations

Before we examine every algorithm in details, we must define a couple of other covering relations, used by these algorithms, that are natural extension of the covering relations introduced in Section 3.3.1.

$\sqsubseteq_S^S$ is a subset of $\mathcal{S}_0 \times \mathcal{S}_0$ and is defined by $\sqsubset_S^N$ (and thus $N_S()$) as follows:

$$s_1 \sqsubset_S^S s_2 \Leftrightarrow N_S(s_1) \supseteq N_S(s_2)$$

that says that a subscription s_1 covers another subscription s_2 when the set of notifications covered by s_1 includes the set of notifications covered by s_2. This relation includes the equality relation between subscriptions, more specifically,

$$s_1 = s_2 \Leftrightarrow N_S(s_1) = N_S(s_2)$$

$\sqsubseteq_A^A$ is a subset of $\mathcal{A} \times \mathcal{A}$ and is defined by $\sqsubset_A^S$ (and thus $N_A()$) as follows:

$$a_1 \sqsubset_A^A a_2 \Leftrightarrow N_A(a_1) \supseteq N_A(a_2)$$

that says that an advertisement a_1 covers another advertisement a_2 when the set of notifications covered by a_1 includes the set of notifications covered by a_2. This also includes the equality relation:

$$s_1 = s_2 \Leftrightarrow N_S(s_1) = N_S(s_2)$$

Hierarchical topology with subscription forwarding

We will describe each algorithm by showing the responses of event servers to various service requests. Such requests include at least the SIENA interface functions (subscribe, publish, etc.), but also control messages for server-to-server communications such as configuration messages and connection setup messages by which servers configure their topology.

In the hierarchical topology, we can apply a "reduced" version of the subscription forwarding algorithm. In fact, hierarchical servers export only the basic interface functions, thus the propagation of subscriptions if allowed only upward in the hierarchy.

Data structures each hierarchical server keeps the URI of another server. This URI is referred to as its *parent* server and may be null. The server also maintains a table of subscriptions that associates a subscription filter to a set of URIs.

Subscription after receiving a simple subscription **subscribe**(U, f), a server X looks up its table searching for one subscription (filter) f' that covers f ($f' \sqsubseteq_S^S f$). If such f' is not present in the table, then the server sends the same subscription, **subscribe**(X, f), to its parent server. In any case the server adds f to its table and appends U to the list of subscribers if U is not listed already.

Unsubscription when receiving a simple unsubscription **unsubscribe**(U, f), a server X extracts from its table all the subscriptions s covered by f ($f \sqsubseteq_S^S s$). For every one of these subscriptions, the server removes U from the list of subscribers. For every subscription s that has an empty subscribers list, the server sends an **unsubscribe**(X, s) request to its parent server and then removes s from the subscriptions table.

Notification when a server receives a notification n, it looks up the table of filters trying to find all the subscriptions s such that $s \sqsubseteq_S^N n$. Then the server computes the union of all the URIs listed for the matching subscriptions and sends to each URI a copy of n. In any case, the server forwards a copy of n to its parent server.

Advertisements and unadvertisements it does not make sense to apply an advertisement forwarding algorithm with a hierarchical topology because connections among servers are asymmetrical. In fact, it would be possible to propagate advertisements from a server to its parent server, but that would be useless since the parent server would consider those advertisements as if they were sent by objects of interest (i.e., from outside the event service) and thus it would not respond by sending back subscriptions. In practice, advertisements and unadvertisements are silently dropped.

Acyclic peer-to-peer topology with subscription forwarding

Data structures each acyclic peer-to-peer server maintains a set N of URIs called *neighbors* that are the identities/addresses of the peer servers to which the server communicates directly. In addition to this, the server holds a table T_S representing a *lattice* of subscriptions in which each subscription s has

- a set of "pointers" to other subscriptions $\overline{S} = parents(s)$ also listed in the table, each one covering s. This is the relation that defines the lattice representing the partial order induced by $\sqsubseteq_S^S$. $parents(s)$ can be empty, in which case, s is said to be a *root* subscription;
- a set of URIs *subscribers*(s), including clients as well as neighbor servers that subscribed for s; and
- a set of URIs *forwards*(s) that lists the neighbor servers to which that subscription has been forwarded.

It is important that the structure of subscriptions in T_S represents *all* the covering relations but only the *direct* ones. In other words, a relation $s_1 \sqsubseteq_S^S s_2$ is reported in T_S (with $s_1 \in$ *parents*(s_2)) only if there is no other subscription s' *in between* s_1 and s_2, i.e.:

$$s_1 \in \textit{parents}(s_2) \Leftrightarrow s_1 \sqsubseteq_S^S s_2 \wedge \nexists s' \in T_S : s_1 \sqsubseteq_S^S s' \wedge s' \sqsubseteq_S^S s_2$$

In processing subscriptions and unsubscriptions, the server makes sure that this property is preserved. Figure 4.8 shows how the server inserts a new subscription in the lattice structure. In the following we will also use the notation *parents*$^*(s)$ indicating the set containing s and all of its ancestors, i.e., the transitive closure of the *parents* relation which includes every subscription that

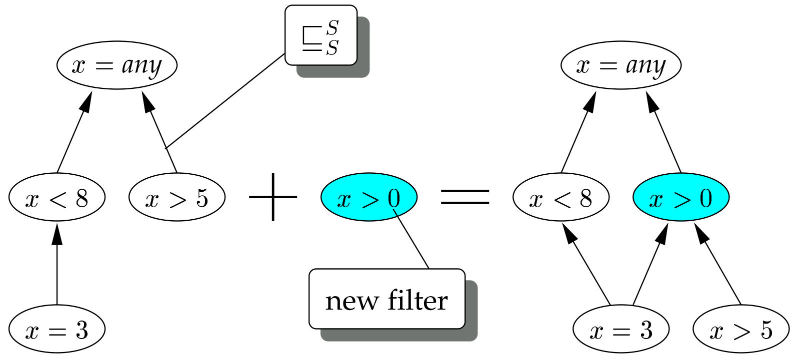

Figure 4.8: Example of a subscription lattice. Types in all filters are *integer*.

covers s. Given *parents*$^*(s)$, we also define the set of forwards that *provide* a given subscription:

$$F(s) = \bigcup_{\overline{s} \in parents^*(s)} forwards(\overline{s})$$

Control and setup the acyclic peer-to-peer server is initialized with zero or one URI representing its initial neighbor. At startup a server X initialized with server Y sends a **peer_connect**(X) request to Y. When a server receives a **peer_connect** request, it can either accept or refuse the connection. In case a server accepts a connection, it sends a confirmation message back to the requesting server and both servers add each other's address to their *neighbors* list. Then the accepting server forwards every *root* subscription to the requesting server adding the requesting server to the corresponding *forwards*. Servers can also be dynamically disconnected with a **peer_disconnect**(X) request. When a server receives a **peer_disconnect**(X), it removes X from its *neighbors*, then it implicitly unsubscribes X for all the *root* subscriptions and finally removes X from all its forwards lists.

Advertisements and unadvertisements with this algorithm, advertisements and unadvertisements are silently dropped.

Subscription When a server X receives a **subscribe**(U, s), it looks up its subscriptions table T_S in order to insert s in the proper position in the lattice defined by T_S. To this end, the server finds the *lowest* subscriptions that cover s:

$$\{s' \in T_S \wedge s' \sqsubseteq^S_S s \wedge \nexists s'' \in T_S : s' = parents(s'') \wedge s'' \sqsubseteq^S_S s\}$$

The following cases apply:

1. s is already in the table, i.e., one s' exists and is equal to s. Here the server adds the current subscriber U to the set of subscribers *subscribers*(s) that have already been registered;

2. no subscription s' exists in T_S, then the server adds s to T_S as a *root* subscription;

3. there exist a set of subscriptions $\overline{S}$, then the server adds s to T_S and sets *parents*(s) **:=** $\overline{S}$.

After positioning the new subscription, the server computes the set of URIs Z to which it will forward the subscriptions **subscribe**(X, s), these are:

$$Z = neighbors - F(s) - \{U\}$$

Intuitively this means that the server forwards the subscription to all its neighbors servers except those to which it already forwarded more generic subscriptions and except the one that sent the new subscription. Then the server subtracts Z from the forwards of the subscriptions covered by s, i.e., the ones that immediately follow s in the hierarchy.

$$\textbf{for each}(\overline{s} \in T_S \wedge s \in parents(\overline{s})) \quad forwards(\overline{s}) \mathrel{\mathbf{:=}} forwards(\overline{s}) - Z$$

Figure 4.9 shows the forward message generated by this algorithm. Labels on

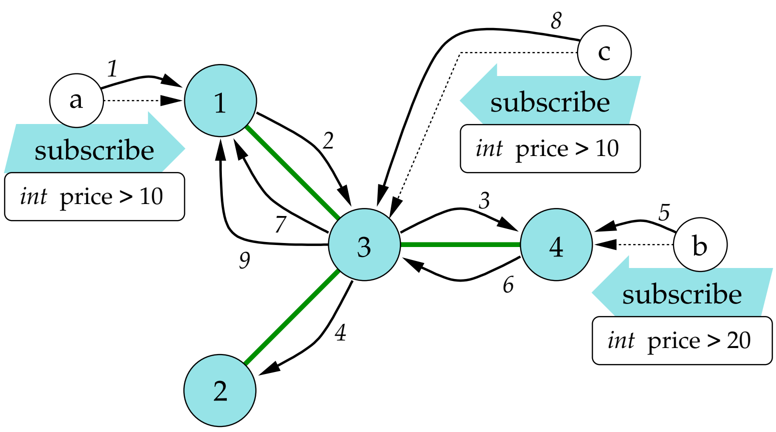

Figure 4.9: Example of subscription forwarding

arcs indicate the temporal sequence of each message. Figure 4.10 shows the

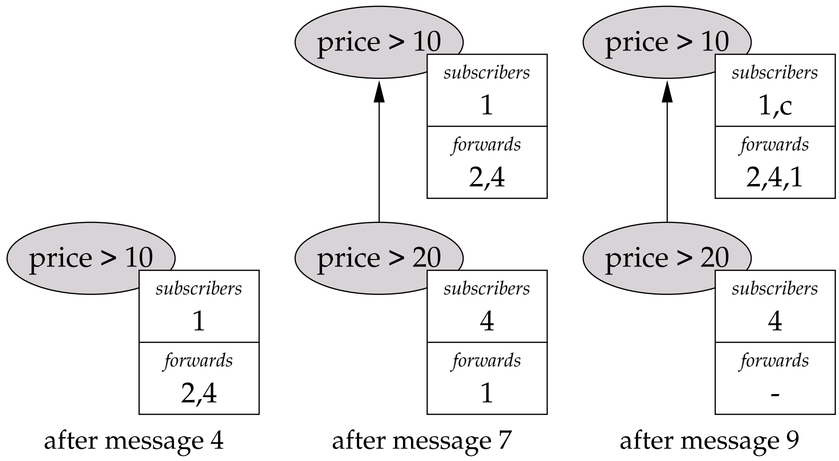

Figure 4.10: Subscription table of server 3 of Figure 4.9

effect of subscriptions and forwards on the subscriptions table of server 3 of Figure 4.9. Note how the effect of adding a more generic subscription s is to move the forwards from the subscriptions covered by s towards s, thus upward in the hierarchy.

Unsubscription The forwarding process for subscriptions makes sure that the following property holds:

$$\forall s \in T_S : F(s) \subseteq \textit{neighbors} \wedge (\forall u \in \textit{subscribers}(s) : F(s) \supseteq \textit{neighbors}/u)$$

Which means that for every subscription s, the set of forwards that *cover* s can be at most equal to the set of neighbor servers, and that for every subscriber u of s, the same set must contain at least the whole set of neighbors, possibly excluding u if it is one of them.

In practice, for each *root* subscription s_r, this can be reduced to two cases[4]:

- s_r has only one subscriber u that is also a neighbor server (we say that u is a *lonely server subscriber*): in this case, the set of forwards of s_r — which is equivalent to $F(s_r)$ since s_r is a root subscription— includes all the neighbors except for the lonely server subscriber u: $\textit{forwards}(s_r) = F(s_r) = \textit{neighbors} - \{u\}$;
- s_r has at least two neighbors subscribers or at least one local subscriber:

[4]In fact, the current implementation of this server stores only this piece of information for each subscription, as opposed to the set of forwards which is redundant.

in this case the set $F(s_r)$ includes all the neighbors servers: $F(s_r) =$ *neighbors*.

Also, since in any case for any given subscription s in T_S, $F(s)$ is a subset of *neighbors*, in non-*root* subscriptions, the set of forwards is either empty or it contains one element that corresponds to the *lonely server subscriber* of its parents. These cases are depicted in Figure 4.10.

These conditions guarantee what we call *optimal notification routing*[5], i.e., they allow servers to exchange only those notifications that are strictly necessary to realize the semantics of the event service. So, when the server receives an **unsubscribe**(U, f), it must cancel U from all the subscriptions covered by f, thereby implementing the semantics of unsubscriptions, possibly rearranging subscriptions in T_S in order to satisfy the *optimal notification routing* conditions. Rearranging subscriptions might result in forwarding unsubscriptions as well as old subscriptions that were covered by the canceled (more generic) subscription.

In details, when a server X receives a **unsubscribe**(U, f), it looks up the subscriptions table and removes U from the sets of subscribers of all the subscriptions covered by f:

$$\textbf{for each}(s \in T_S : f \sqsubset_S^S s) \quad \mathit{subscribers}(s) \mathbin{\textbf{:=}} \mathit{subscribers}(s) - \{U\}$$

Then the server identifies two classes of subscriptions:

1. subscriptions that remain with a lonely server subscriber that is also in their forwards:

$$S_l = \{s \in T_S : |\mathit{subscribers}(s)| = 1 \wedge \mathit{subscribers}(s) \cap \mathit{forwards}(s) \neq \emptyset\}$$

2. subscriptions that remain with no subscribers:

$$S_e = \{s \in T_S : \mathit{subscribers}(s) = \emptyset\}$$

For every subscription s_l in S_l, the server unsubscribes for s_l with the lonely forward u, i.e., it sends an **unsubscribe**(X, s_l) request to u, and then, if s_l has some descendents in the subscription lattice $(\{\overline{s}_l \in T_S : s_l \in \mathit{parents}(\overline{s}_l)\})$, the server moves the forward u to every descendents subscription by issuing a new subscription for each one of them.

For every subscription s_e in S_e, the server removes all its forwards by unsubscribing for s_e with all of its forwards, i.e., the server sends an **unsubscribe**(X, s_e) request to every peer server $u \in \mathit{forwards}(s_e)$. Then, similarly to what it does for subscriptions in S_l, the server finds all the descendents of

[5]Note that *optimal notification routing* refers only to notifications, thus, it does not imply minimal overall network traffic.

s_e and adds to each one of them the whole set of forwards removed from s_e. To do that, it adds the sets in its data structures, and also for every descendent subscription $\overline{s}_l$, it sends a **subscribe**$(X, \overline{s}_e)$ request to every peer server $u \in$ *forwards*(s_e).

Notifications this algorithm processes notifications exactly like the one that operates on the hierarchical topology. So, a subscription n is forwarded to every URI in:

$$R(n) = \bigcup_{s \in T_S : s \sqsubset^N_S n} \textit{subscribers}(s)$$

Acyclic peer-to-peer topology with advertisement forwarding

With the subscription forwarding algorithm described in the previous section, we have seen almost everything we need to know to implement an advertisements forwarding algorithm on the same topology of servers. In fact, we can exploit the duality between these two classes of algorithms to transpose the subscription forwarding algorithm to the advertisements forwarding. To some extents, 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 levels: one realizes the forwarding of advertisements while the other one realizes the forwarding of subscriptions. Both levels have similar data structures and similar algorithms, equivalent to the ones that implement the subscription forwarding algorithm. In particular, the server has a table of advertisements T_A that binds an advertisement a to a set of other advertisements $\overline{A} =$ *parents*(a), a set of URIs *advertisers*(a), and another set of URIs *forwards*(a).

These two levels interact in the sense that the *upper* level, i.e., the advertisement forwarding, directs the *lower* level, i.e., the subscription forwarding, by configuring some *parameters* of its table T_S. More specifically, in maintaining T_S, e.g., in subscribing or unsubscribing a pattern s, the server does not use the global set of *neighbors*, but instead it uses a subset *neighbors*$(s) \subseteq$ *neighbors* that is specific to s. *neighbors*(s) is bound to the set of advertisers listed in T_A for all the advertisements covering s. Formally:

$$\textit{neighbors}(s) = \bigcup_{a \in T_A : a \sqsubset^S_A s} \textit{advertisers}(a) \cap \textit{neighbors}$$

Note how this formula is similar to the one that is used in the subscription forwarding algorithm to route notifications. Also note that, one effect of this binding is that new advertisements and unadvertisements are viewed by the lower level like new peer connections or dropped peer connections, 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 advertisers.

Algorithms for the generic peer-to-peer topology

For the generic topology, we can use the algorithms that we developed for the acyclic topology provided we complete them with mechanisms that avoid routing messages along cycles that might be present in the topology. What we need to solve here coincides with the well known problem of routing in a datagram network (see [8]).

In SIENA, we use the classic reverse path forwarding technique [20]. According to this technique, a server X forwards a request —a subscription in case we use the subscription forwarding algorithm, or an advertisement if we use the advertisement forwarding algorithm— only if it is coming from the peer server that is on the shortest path that connects the *source* of that request to the server. The request is then forwarded to every peer server except the one that sent it. By *source* we mean the event server, usually an access point, that generated that request for the first time.

The reverse path forwarding technique requires that every server knows which peer server is the *next hop* towards every other server in the network. Thus, every server X has a table that maps URIs corresponding to other servers in the network to one of X's peer servers. To compute this map, the current implementation of generic peer-to-peer SIENA servers uses a simplified version of the distance-vector algorithm (also known as Ford-Fulkerson or Bellmann-Ford algorithm [29, 7]). The server maintains a table T_R that associates a source server s to a peer server *nexthop*(s) and a distance *distance*(s). Also, the server-to-server protocol is augmented with the information needed by this algorithm, in particular, every request r carries an attribute (*source*(r)) and an attribute (*distance*(r)) that indicates the length of the path that the request r has gone through[6].

When a server X receives from its peer server Y a new request r originated by $S =$ *source*(r), it acts according to the following situations:

- if S does not exist in the table T_R: the server adds an entry for S with *nexthop*$(S) := Y$ and *distance*$(S) :=$ *distance*(r). Then the server processes r normally;

[6] Several metrics can be chosen to express this distance. The current implementation of SIENA uses a metric based on the latency of each hop.

- if S is already in T_R: here we have other three cases:
  - if *nexthop*$(S) = Y$: the server processes r normally;
  - if *nexthop*$(S) \neq Y$ and *distance*$(S) >$ *distance*(r): the server updates the T_R for S, i.e., *nexthop*$(S) := Y$ and *distance*$(S) :=$ *distance*(r). Then the server processes r normally;
  - if *nexthop*$(S) \neq Y$ and *distance*$(S) \leq$ *distance*(r): the server drops the request.

When processing a request from a local client that needs to be forwarded to some other servers, the server assigns its address to the *source* attribute and 0 to the *distance* attribute. When processing non-local requests, the server adds the "cost" of the link through which it got the request. If the metric is the number of hops, then the cost is simply 1. In SIENA, we use a metric based on the latency of network links. In any case, the latency is not dynamically tested, but it is statically configured.

Note that in SIENA the routing information is piggy-backed on normal service requests. A logically alternative approach is taken on the Internet where routing information is exchanged among routers by means of a special routing information protocol (RIP [34]). RIP uses the same class of distance-vector algorithms.

4.3 Pattern observation

So far we have seen how simple subscriptions and simple notifications are handled by event servers. A major functionality introduced with SIENA is the distributed monitoring of patterns of notifications according to compound subscriptions. This functionality is implemented following the *upstream monitoring* principle set forth in Section 4.2.2.

For monitoring of patterns, event servers assemble sequences of events from smaller sub-sequences or from single notifications that are already "available". The availability of notifications is determined by advertisements, this is why this technique requires an advertisement-based semantics. The availability of sub-sequences is given by other monitors that the server has already set up for previous compound subscriptions. We call *pattern factoring* 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*.

4.3.1 Available patterns table

Every server maintains a table T_P of available patterns. This table is simply the advertisements table that, in addition to usual advertisements, contains also those patterns that the server has processed already. Each pattern p in T_P is associated with the set of URIs *providers*(p) that lists all the peer servers from which p is available, possibly including the server itself if the pattern is a simple pattern advertised by a local client or if it is a compound pattern that is monitored locally.

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

Table 4.2: Example of a table of available patterns

Table 4.2 shows an example of table of available patterns. The table says that events matching filter a_1, intuitively events that signal a failed login with an integer attribute named "attempts", are available from server 2, and that events matching filter a_2, file modification events, are available from servers 2 and 3.

4.3.2 Pattern factoring

Let us suppose a server X receives a compound subscription **subscribe**(U, p) where $p = f_1 \cdot f_2 \cdot \ldots \cdot f_k$. Now the server scans pattern p trying to match each f_i with a set of patterns $a'_i, a''_i, \ldots$, or trying to match a sequence $f_i \cdot f_{i+1} \cdot \ldots \cdot f_{i+k_i}$ with a compound pattern $a_{i \ldots i+k_i}$ where every pattern a is available in T_P.

For example, assuming the table of available patterns shown in Table 4.2, suppose the server receives a subscription s for a sequence of two "failed login" alarms with one and two attempts respectively, followed by a file modification events on file "/etc/passwd". In response to s, the server executes the factoring of s, matching the three events of s with the sequence of available patterns $a_1 \cdot a_1 \cdot a_2$. Table 4.3 shows the subscription and the factoring computed by the server. Since in SIENA the only operator for combining sub-patterns is the sequence operator, the output of the factoring process is always a sequence.

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

Table 4.3: Example of a factored compound subscription

4.3.3 Pattern delegation

Once a compound subscription is divided into available parts, the server must:

- send out the necessary subscriptions in order to collect the required sub-patterns, and
- set up a *monitor* that will receive all the sub-patterns and will notify the occurrence of the whole pattern

In deciding which subscriptions to send out, the server tries to re-assemble the elementary factors in groups that can be delegated to other servers, thereby implementing the *upstream monitoring* principle. A group of sub-patterns $A = a \cdot b \cdot c \cdots$ can be delegated to a peer server Y if the following conditions apply:

- $a \cdot b \cdot c \cdots$ are contiguous in the factorization. This is obvious since sequences can be composed only of sub-sequences, i.e., groups of filters, that are contiguous. For example, if $S = a \cdot b \cdot c \cdots$, the group $A = a \cdot c$ can not be delegated;
- $a \cdot b \cdot c \cdots$ are all available from the same peer server Y. This is quite clear too, a group $A = a \cdot b$ can not be delegated to Y if either a or b is not available from Y;
- $a \cdot b \cdot c \cdots$ are not available from any other peer server. This says for example that, if a and b are available from Y, but b is also available from Z, then the group $A = a \cdot b$ can not be delegated to Y. In fact, had the sequence been delegated, events matching b coming from Z could not contribute to the assembly of the sequence, thus causing the event service not to detect potentially valid sequences;

- none of the patterns in $a \cdot b \cdot c \cdots$ is listed anywhere else in the global pattern other than in other sequences that are identical to $a \cdot b \cdot c \cdots$. For example, in a sequence $S = a \cdot b \cdot a$, the group $A = a \cdot b$ can not be delegated because once the first $a \cdot b$ is matched, the monitor should be able to be notified of single a events, but these would be masked by the group A that has been delegated.

In the example of Table 4.3, the server would group the first two filters $a_1 \cdot a_1$ and delegate the sub-pattern defined by the corresponding two subscriptions to server 2, thus it would send a subscription **subscribe**(X, A_1) with pattern:

$$A_1 = \boxed{\begin{array}{ll} \textit{string} & \textit{alarm} = \textit{"bad-login"} \\ \textit{integer} & \textit{attempts} = 1 \end{array}} \cdot \boxed{\begin{array}{ll} \textit{string} & \textit{alarm} = \textit{"bad-login"} \\ \textit{integer} & \textit{attempts} = 2 \end{array}}$$

to server 2, and then it would subscribe for the simple subscription **subscribe**(X, A_2) with

$$A_2 = \boxed{\begin{array}{ll} \textit{string} & \textit{file} = \textit{"/etc/passwd"} \\ \textit{string} & \textit{operation} = \textit{"file-change"} \end{array}}$$

to server 3. Eventually the server will start up a monitor that recognizes the sequence $(A_1 = a \cdot b) \cdot (A_2 = c)$.

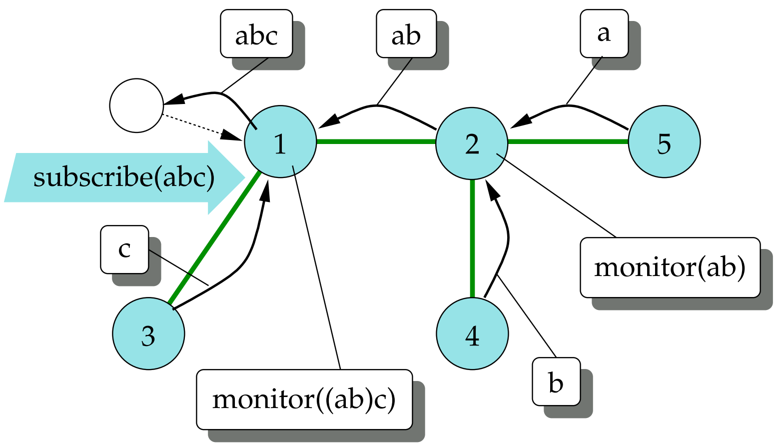

Figure 4.11: Pattern monitoring and delegation

Figure 4.11 shows a scenario that corresponds to the tables and subscriptions discussed above. In particular, we focused on server 1 that delegates $a \cdot b$ to server 2, subscribes for c, and monitors $(a \cdot b) \cdot c$. The diagram also shows how server 2 might handle the delegated subscription. Assuming that a is available from server 5 and b is available from server 4, then server 2 sends the two corresponding subscriptions to 4 and 5 and then starts up a monitor for $a \cdot b$.

4.4 Other optimization strategies

In addition to the main principles discussed in the previous section, SIENA event servers may perform other types of optimizations. These techniques are variants of the algorithms discussed so far.

4.4.1 Batching and merging subscriptions and advertisements

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

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

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

4.4.2 Space vs. processing vs. communication: trade-offs

All the algorithms we presented are geared towards the optimization of network resources. However there are trade offs between the need of reducing communications and the processing and memory usage on each event server.

In some cases, the data structures on servers can be compressed, thus saving space as well as processing power for maintaining and searching those data structures. Compression can be achieved by merging subscriptions in subscription tables much the same way they can be batched and merged in the transmissions to other servers.

In particular, it might be preferable to compress the data structures residing on servers a little more than what is implied exactly by the covering relations, thus saving space and processing power, at the expense of forwarding or receiving unnecessary messages. For example, a server X can merge two subscriptions s_1 and s_2 sent by the same non-local clients (i.e., from a peer server) into another subscription $\overline{s}$ that defines a set of subscriptions that is *larger* than the union of the notifications defined by s_1 and s_2 alone. This operation would cause the server to forward unnecessary notifications, i.e., those that match $\overline{s}$, but that do not match either s_1 or s_2. The advantage is that X now stores only one simpler —since it is more generic— subscription $\overline{s}$ as opposed to the two subscriptions s_1 and s_2. This compression can be done only with subscriptions forwarded by peer servers since *spurious* notifications must go through additional filtering and eventually they must be dropped before they reach an interested party.

4.4.3 Evaluation of the covering relations

The evaluation of the covering relations is clearly a crucial step in every dispatching algorithm. For example, when a server receives a new notification n it looks for all the subscriptions s that cover n. In general, this implies computing the relation $s \sqsubset_S^N n$ for each subscription s received by the server. However, the server can arrange its subscriptions in a lattice as explained in Section 4.2.3. The lattice is such that for every element s_1, all the subscriptions $s_{1.1}, s_{1.2}, \ldots, s_{1.k}$ covered directly by s_1 define sets of subscriptions that are included in the set defined by s_1, thus, if a notification n is not covered by s_1 it can not be covered by subscriptions $s_{1.1}, s_{1.2}, \ldots, s_{1.k}$ either, thus the server can safely skip the evaluation for the whole subtree rooted in s_1.

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

Chapter 5

Simulation Framework

The design of SIENA as a distributed architecture is difficult because of the decentralized nature of its topologies and algorithms. Some issues that we had to face are common in the design and development of a highly distributed systems. In fact, our problem is to program single "autonomous cells" —the event servers— that, when connected together, exhibit a high-level behavior that corresponds to the semantics of the event service. Here the critical activities are verifying, validating, fine-tuning, and debugging the algorithms and the topologies with respect to the specifications of the event service. In addition to what is related to the development of the dispatching algorithms, we are faced with the challenge of evaluating the performances of SIENA with respect to overall network traffic generated by the event service, the traffic around specific nodes, possible overloading of links, and ultimately our initial goal: scalability.

An empirical approach to these two problems is clearly not feasible. We can not simply implement our algorithms and deploy them on a significant number of nodes just to find out that they are done wrong and then retire them and re-iterate the deployment as a step of our development process. Even assuming our algorithms are correct when we deploy them, we are left with the problem of monitoring the whole system in order to collect the information that are necessary for the evaluation of performance. This is a difficult task in itself, plus unless we use some other communication facility that is external to the target network, it is likely that the performance data will be influenced by the observation process.

An opposite approach, at least for the verification of the semantics of the event service, is to give a formal description of the behavior of each server together with a formal model of the topology, and then to use a model checker to prove that the servers, connected according to that topology, implement the desired semantics. This approach has a fundamental limitation in the case of SIENA because the state space of each server and the state space of the network are infinite. Server can not be modeled with a finite state because of their dynamic data structures, similarly, all the topologies we developed are potentially infinite. Unfortunately, while model checking techniques for finite state systems are well-established and several automatic tools are available, the case of infinite states is still a research subject and only a few results are available in some restricted cases [9, 10].

The approach we followed for SIENA is based on a *simulation environment*. In the initial development phase, we used this simulation environment to verify and validate the algorithms. In a following phase, we analyzed the scalability of the architectures by means of systematic simulations using several models of wide-area network scenarios. In this chapter we will describe the simulation environment we used and then we will concentrate on the evaluation of scalability. We will discuss the network models that we used and the metrics we examined and we will present the initial synthetic results, corresponding to nearly 2200 simulations of scenarios.

It should be clear that, simulating distributed systems at the scale of a wide area network like the Internet is an extremely challenging endeavor. A complete simulation is just not feasible because of the immense amount of data and interactions that are involved at different levels of granularity. According to an estimate by Ahn and Danzig [1], "five minutes of activity on a network the size of today's Internet would require gigabytes of real memory and months of computations on 100 MIPS uniprocessors". The biggest difficulty becomes the choice of the right models that would allow us to simplify the simulation without loosing much in the way of accuracy. In particular, as pointed out by Paxson in [61], the idea of shifting the focus from a packet-level simulation to a simulation of an application-level model seems to be a very promising and motivates even more the approach we take in SIENA.

5.1 Simulator

We implemented a message-based discrete-event simulator [2, 3]. In this class of simulators, *physical* processes are realized with *logical* processes and the interaction among physical processes is modeled with the exchange of messages between the corresponding logical processes[1]. A logical process is defined simply by an object in the host programming language. The process object defines a procedure that handles incoming messages. Messages are objects themselves that may contain any type of information. Every process, in addition to all the functionality offered by the host programming language, can perform one of the following operations:

[1]In this context, the term *message* is not to be confused with the concept of network communication unit. Instead, the term has the semantics defined in discrete-event simulation literature (for a survey on this topic, see [52]).

1. create other processes
2. send messages to other processes
3. wait for the some time to pass
4. wait for a message to arrive
5. terminate itself

The simulator makes sure that messages are properly queued for the destination process and that time-outs are set according to processes' requests. Whenever a time-out expires for a process or when a process has pending messages, the server yields control to that process.

The simulator of SIENA is implemented as a classical sequential discrete-event simulator biased towards the conceptual framework that we defined for SIENA. Note that the conceptual framework of SIENA maps directly onto the elements of a discrete-event simulator. In particular, objects —interested parties, event servers, etc.— are already modeled and implemented as *reactive* processes, in fact we described server algorithms in terms of the routines that respond to notifications, subscriptions and advertisements. Also, event notifications, subscriptions, and advertisements are already represented as network messages that are very naturally implemented as messages within the simulator. In addition to these basic abstractions, the simulator of SIENA implements a network model with sites and links that mimics the real communication infrastructure.

We preferred a custom simulator to a general purpose one like *ns-2* [76] because it gives us more freedom in the implementation of the components to simulate and in abstracting away details of the network model. Specifically, the simulator implements the network model that considers only the end-to-end characteristics of a link hiding the underlying packet switched network. This abstraction greatly simplifies the simulation without introducing a significant distortion in the global results [35].

The SIENA simulator is written in C++ and most of its data structures are realized by means of the Standard Template Library (STL) [57]. Processes are objects of a sub-class of a *process* class that defines three virtual methods: *initialize*, *process*, and *terminate*. Every process objects has a reference to the *site* on which it executes. A *clock* variable holds the *current* time of the physical system.

The core data structure that drives the execution of the simulator is list of *action* items. Every action has a time-stamp that indicates the scheduled time for that action and the list is sorted in ascending time order. The simulator cycles extracting the first element of the actions list and executing the corresponding action. When extracting an action item, the simulator advances the clock to the schedule time of that action. An action can be one of:

deliver_message(m,p) the simulator yields control to the destination process p passing the message m as a parameter;

timeout_expired(p) the simulator yields control to the process p that set the time-out;

initialize_process(p) the simulator yields control to the initialization function of the new process p;

terminate_process(p) the simulation executes the termination function for process p and then destroys p.

Within the body of its main functions, a process has access to the services of the simulator. The basic services are:

create_process_at(p,s) the simulator defines *p.site* **:=** s and then schedules a **initialize_process**(p) action at time $T = clock$;

create_process(p) the simulator defines *p.site* **:=** *current_process.site* and then schedules a **initialize_process**(p) action at time $T = clock$;

create_process(p) the simulator schedules an **initialize_process**(p) action at time $T = clock$;

send_message(m,p) the simulator schedules a **deliver_message**(m,p) action at time $T = clock + transmission\_time(current\_process.site, p.site, m)$;

exit the simulator schedules a **terminate_process**(*current_process*) action at time $T = clock$.

In addition to these basic services, every process can access other information, including the current time, i.e., the value of the clock variable, and some network information about its neighborhood.

5.2 Scenario models

Simulations are executed on *network scenarios*. A network scenario is the description of a network with its population of components. Figure 5.1 shows the layered structure of our network scenarios. Before entering its main loop, the simulator reads a scenario description according to which it defines network resources and it instantiates objects.

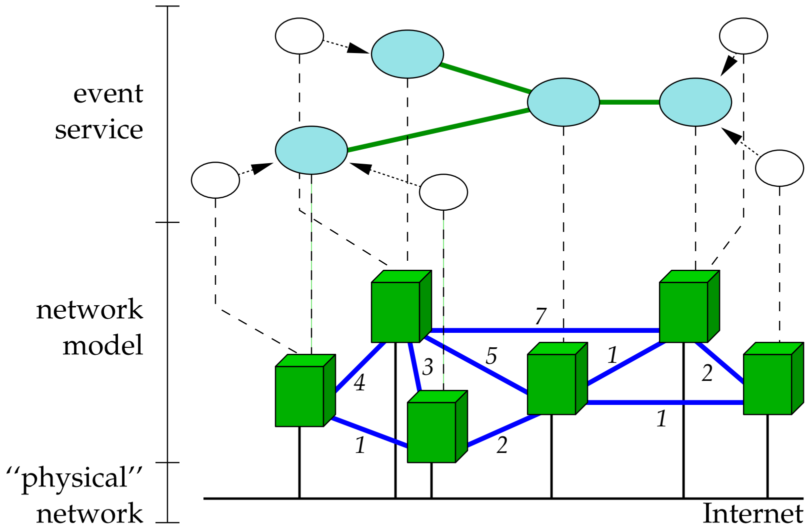

Figure 5.1: Layered network scenario model

5.2.1 Network model

At the lowest level of a network scenario we have the model of the network. This layer sits on top of what we consider the "physical" connections among hosts. Here the model defines *sites* and *links*. Sites are computers or more likely local networks connecting sets of computers. Links are the high-level descriptions of connections between two sites. Links are implemented by the "physical" layer by means of some communication protocol (e.g., TCP/IP), however, the simulator abstracts away from the details of this protocol and uses a simple characterization: a link that connects site s_1 and s_2 is has three parameters: $latency(s_1, s_2)$, $bandwidth^{-1}(s_1, s_2)$, and $cost(s_1, s_2)$. Latency is measured in time units, $bandwidth^{-1}$ is measured in time units per bit, and cost is measured in money units. Links are assumed symmetrical, i.e, $latency(s_1, s_2) = latency(s_2, s_1)$. An analysis performed by Paxson [60] shows that although the end-to-end behavior of the Internet has become less predictable, anomalies are still at an acceptable level and, what is particularly important for our assumptions, the majority of routing paths is stable for long periods of time and the reliability of the measure increases significantly when we consider short distances.

Given the above definitions, the simulator computes the time it takes to deliver a message m from site s_1 to s_2

$$transmission\_time(s_1, s_2, m) = latency(s_1, s_2) + length(m) \times bandwidth^{-1}(s_1, s_2)$$

Note that the characteristics of a link, and thus the transmission time on that link, do not depend on other factors such as the time of day, the traffic already routed through that channel, etc. This is not really a limitation of the simulator since, from the viewpoint of the simulator, it is just a matter of introducing these new parameters in the cost and latency functions. The real issue is rather that we do not know what the right model is for describing the dynamics of a generic link over the Internet at this level of granularity.

To compile realistic network topologies that approximate the behavior of real wide-area networks, we used a generator of random network graphs that implements the Transit-Stub model [82] (other models for random network graphs are Waxman's non-hierarchical graphs [78], and Tiers [25]. A discussion on these models can be found in [83]). Although we experimented networks of 500 up to 1000 nodes, the bulk of simulations we present here were performed on a graph of 100 nodes. Figure 5.2 shows this network topology.

5.2.2 Event service model

The event service model describes the topology of servers with respect to the topology of the network. In practice, once we have a description of the network, we must populate sites with event servers and we must configure their interconnections.

We do not use a random layout for servers, but instead we determine how many servers will be set up and where they will reside by matching their type with the layout of the network. In other words, we assume that location and connections of servers are more or less an image of the underlying network model. So we will have one server per site, and every server will be configured to connect to the servers residing on its neighbor sites. For example, if a network topology has three nodes, say "uci", "colorado", and "polimi" with two links connecting "uci" with "colorado" and "colorado" with "polimi" respectively, we will set up three servers, one for each site, and we will configure them so that the server at "uci" will talk to the server at "colorado" (and vice versa) and the server at "colorado" will talk to the server at "polimi" (and vice versa). This assumption significantly reduces the variable space thus simplifying the set of scenarios that we will cover, and it is also very reasonable since it reflects the structure of domains that characterizes the Internet [14].

The scenarios we are simulating at this time include only *homogeneous* event service topologies. In addition to the distributed topologies, we simulate also a "reference" centralized topology that we use as a baseline for our comparison.

centralized in this case there will be one server for the whole network and it will be randomly placed on one node of the network;

hierarchical this topology creates one server for each node of the network. The connections between servers (the *parent* server pointer) will be configured

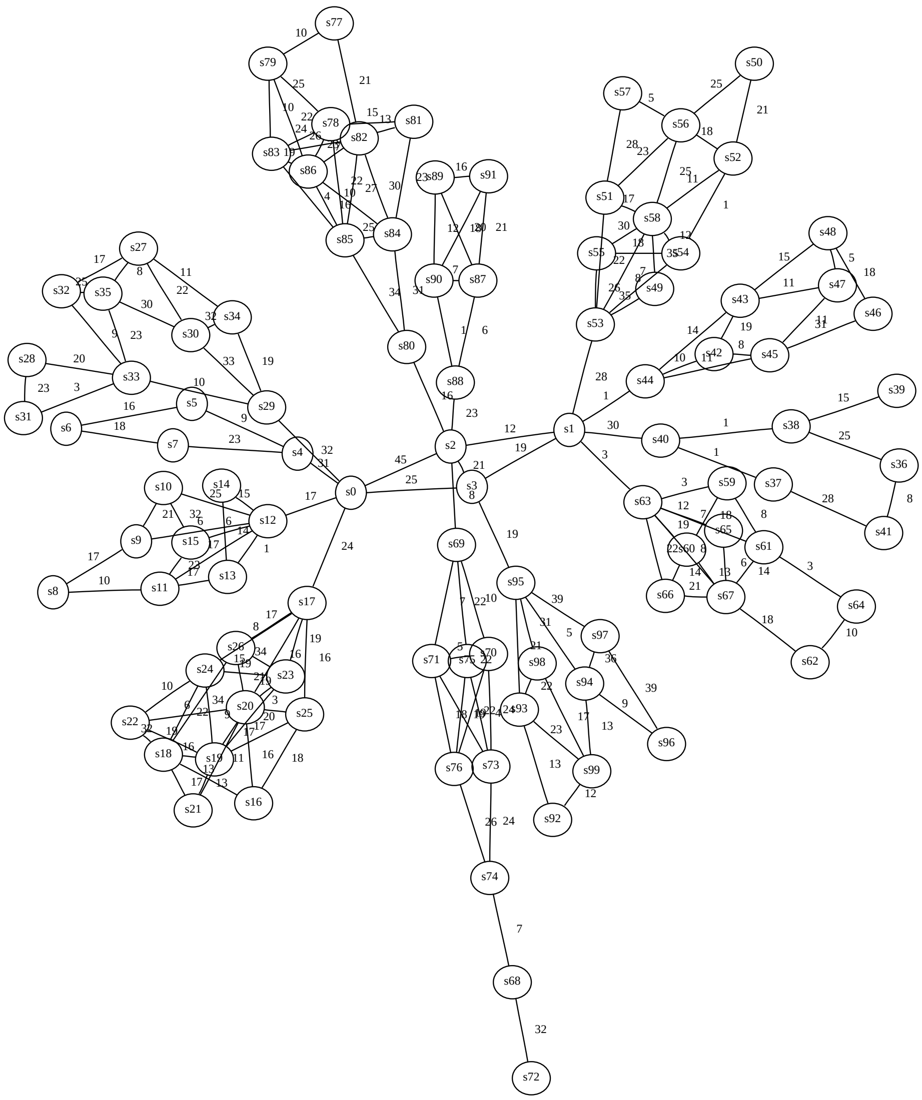

Figure 5.2: Randomly generated network topology

so that the tree of servers will correspond to the minimal spanning tree rooted in node 0 of the network. Note that the algorithm that creates the network cost function produces a quasi-hierarchical network in which the first nodes are at the highest level in the hierarchy, thus the choice of using node 0 as a starting point;

acyclic peer-to-peer this topology is exactly equivalent to the hierarchical topology. I.e., it has one server per node and the connections between servers are laid on a minimal spanning tree rooted in the first node of the network (the difference is in the type of connections);

generic peer-to-peer this topology of servers matches the topology of the net. Each node has one server and connections among servers are an exact replica of the connections of the network topology.

5.2.3 Applications model

The applications model describes the applications that use the event service. In particular, we must determine number, location, behavior of objects as well as which event server will be their event service access point. As we have done so far, we model objects of interest and interested parties as separate objects.

An object of interest executes m sequences

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

For every object, we also give the average time delay between two messages $\bar{t}$.

An interested party executes the dual cycles of an object of interest, i.e., it executes m sequences:

$$\mathit{subscribe}, \overbrace{\mathit{recv\_notif}, \mathit{recv\_notif}, \ldots}^{n \text{ times}/t \text{ time units}}, \mathit{unsubscribe}$$

The length of the inner cycle can be specified directly in terms of the number of notifications that the interested party waits for before unsubscribing, or alternatively giving the mean time interval between a subscribe and the corresponding unsubscribe.

Both interested parties and objects of interest are steady objects. Thus they connect to one event service access point. Specifically, they connect to the nearest event server. Since the topology functions associate a default zero cost to local connections, i.e., connections between processes that execute on the same node, there can only be two cases:

- *centralized event service*: here objects and parties can connect to the only one server in the network, and

- *hierarchical, acyclic, and generic event service*: in this case, since there will be one server for each node, objects and parties connect to the server that runs on their site.

5.3 Simulation Results

5.3.1 Simulation traces

The result of a simulation is a trace file that contains an entry for every interaction between simulated objects. In the simulation of SIENA scenarios, all the interactions correspond to network messages exchanged by objects. The trace entry for a network message contains the following information:

message type determines which high-level request is carried by the message. It can be one of the messages that realize the interface functions of the event service (e.g., *notification*, *subscription*, *unsubscription*, etc.) or it can be one of the special messages implementing the server-to-server communication in peer-to-peer topologies.

message_id this is the unique sequence number of a message;

service_id this is the identifier of the service to which this message is a follow-up. When a client requests a service by sending a message to its access point, the access point uses the message id of that request to identify a service. Every message forwarded as a consequence of that request are marked with that service id;

source_object_id this is the identifier of the sender. Object identifiers are unique and they are not recycled when objects terminate their execution;

source_site this is the identifier of the site from which the message is sent (it coincides with the site on which the sender runs);

destination_object_id this is the identifier of the receiver;

destination_site this is the identifier of the site to which the message is sent (it coincides with the site on which the receiver runs);

departure_time this is the value of the clock at the time the message is posted by the sender;

arrival_time this is when the receiver gets the message, i.e., when the simulator schedules the response action;

cost cost of the message according to the cost function.

A service request can have a follow up of many messages of different types. For example, an unsubscription can result in many other unsubscriptions as well as new subscriptions.

5.3.2 Metrics and aggregation criteria

From the trace file produced by the simulation, we extract and aggregate some metrics that synthesize the performance of an architecture. The aggregation criteria are:

total all the messages exchanged for the whole simulation;

per-site all the messages sent or received by one site;

per-link all the messages sent through one link;

per-service all the messages with a specific service id.

On these groups of messages, we compute *cost* and *delay* metrics. The cost can be computed for all the groupings. Instead, the delay is computed using the per-service grouping only. The delay in a service request is the time it takes to complete a service request. For a service id s:

$$\mathit{delay}(s) = \max_{m \in G_s} \mathit{arrival\_time}(m) - \min_{m \in G_s} \mathit{departure\_time}(m)$$

where G_s is the group of messages having $\mathit{service\_id} = s$.

5.3.3 Simulation process

Figure 5.3 shows the simulation process. Ovals represent data. Boxes represent processing activities. Dark ovals or boxes represent information or activities that can be parametrized, i.e., the variables of our model. The "random" flash arrows introduce non-determinism, for example in the simulation process, non-determinism is given by the probabilistic behavior of objects of interest and interested parties. The two staring points are the topology meta-parameters and the scenario parameters. The topology meta-parameters are the input parameters for the topology generator[2]. These parameters are specific of the generation method (see [83] for details). The topology generator tool produces a graph using the Stanford GraphBase format [42] which we then translate into AT&T's *dot* format [43] for the subsequent processing.

Figure 5.4 shows a file containing scenario parameters[3]. The `servers` tag defines the architecture to be used in this scenario while the `objects` and

[2] We used the Georgia Tech Internet Topology Models (GT-ITM), a publicly available package for generating random graphs [31]

[3] Actually, in order to automate the simulation process, we used a description file parametrized with CPP macros

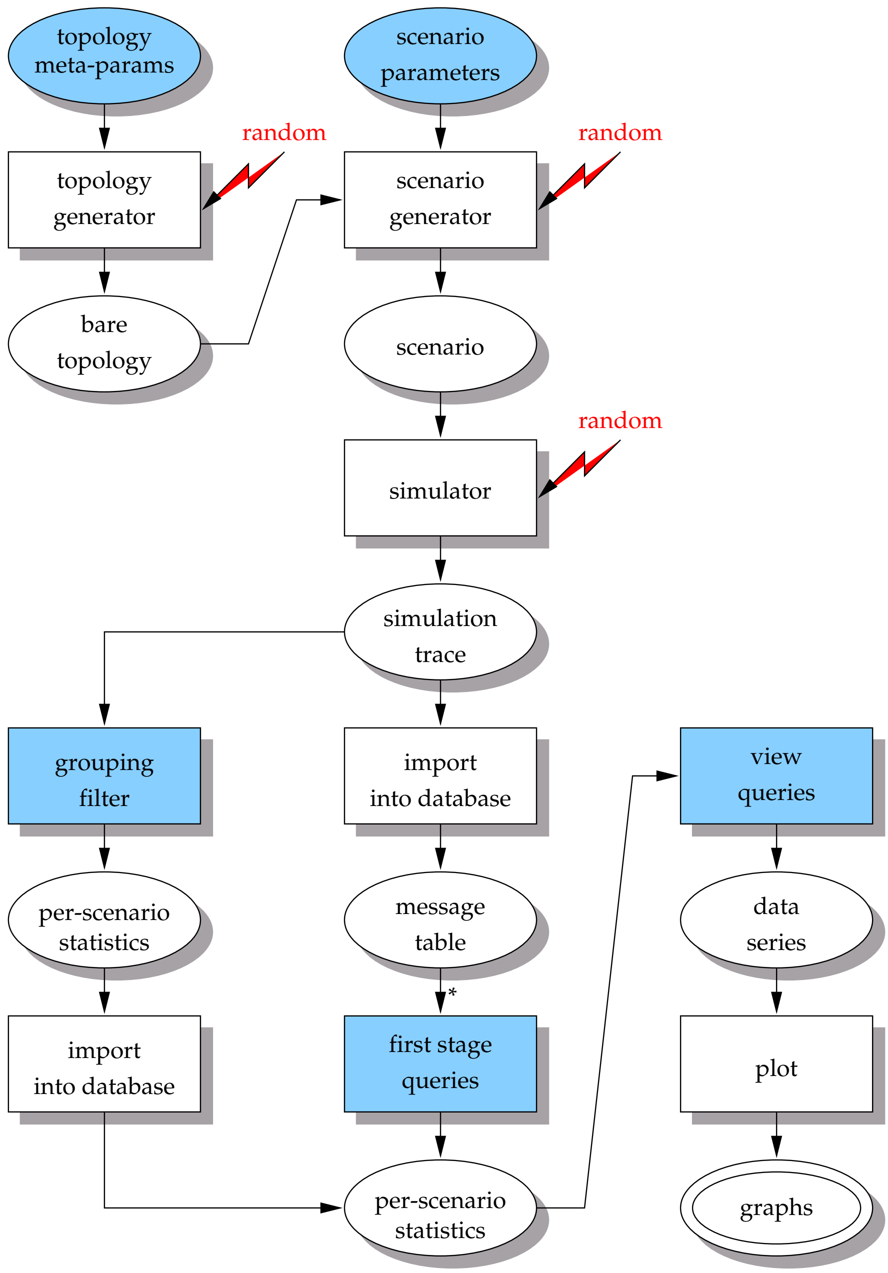

Figure 5.3: Simulation process

```
//
// scenario spec file
//
servers = acyclic adv_fwd;

objects {
   event = "A";
   number = 1000;
   pub/adv = 20;
   time/pub = 2000~2500;
   cycles = 10;
};

parties {
   pattern = "A";
   number = PARTIES;
   notif/sub = 15;         // these two are mutex
// time/sub = 10000~15000  //
   cycles = 10;
   init_delay = 2000~2500;
};
```

Figure 5.4: Scenario parameters

`parties` sections specify behavior and number of objects of interest and interested parties respectively. The number of objects of interest and interested parties is the total number for a scenario, objects are distributed randomly on all the sites. The behavior also defines which classes of events will be generated by objects of interest (with the tag `event`) and which pattern of events is of interest for interested parties (with the tag `pattern`). We use the symbol "A" as a shortcut to indicate "events of type A". Since we are initially interested in testing the consumption of network resources rather than computational resources, we have simulated only simple filters of this kind. The scenario parameters file can contain any number of objects and parties sections with possibly different behavior specifications. Note that some numeric parameters are expressed with a pair of *min* $\sim$ *max* values (e.g., `time/pub = 2000 2500`). This notation indicates a uniform distribution of values between *min* and *max*.

Scenario parameters are processed together with the bare topology to create the scenario file read by the simulator. Figure 5.5 shows an example of scenario description. This file contains the definition of sites and links (substantially a copy of the topology file), the allocation of servers, the definition and allocation of objects of interest and interested parties.

The simulation produces a stream of trace records. We process the simulation traces in two ways: we either import them into a relational database from which we extract aggregate metrics or alternatively, for extremely large simulations, we pipe them through a filter that computes the aggregate metrics and

```
//
// detailed scenario configuration
//
sites:
        s0 - s4 = 31;
        s0 - s2 = 45;
        s0 - s3 = 25;
        s1 - s3 = 19;
        s1 - s63 = 3;
        s2 - s80 = 16;
        s2 - s69 = 8;
// ...
//
// event servers type and conections
//
acyclic adv_fwd:
        S@s0;
        S@s2 -> S@s0;
        S@s3 -> S@s0;
// ...
        S@s69 -> S@s2;
        S@s80 -> S@s2;
// ...
//
// objects of interest
//
objects:
10 @ s0 server=S@s0 event=A cycles=10 count=20 delay=2185;
13 @ s1 server=S@s1 event=A cycles=10 count=20 delay=2042;
8 @ s2 server=S@s2 event=A cycles=10 count=20 delay=2126;
9 @ s3 server=S@s3 event=A cycles=10 count=20 delay=2403;
// ...
//
// interested parties
// (delay = 15N means cycle "after 15 notifications")
//
parties:
95 @ s0 server=S@s0 pattern=A cycles=10 delay=15N init_delay=2461;
92 @ s1 server=S@s1 pattern=A cycles=10 delay=15N init_delay=2029;
108 @ s2 server=S@s2 pattern=A cycles=10 delay=15N init_delay=2331;
98 @ s3 server=S@s3 pattern=A cycles=10 delay=15N init_delay=2124;
// ...
```

Figure 5.5: Scenario description file

then we import only these synthetic values into the database. After this first step, we obtain per-scenario statistics of aggregate metrics. From this information we select the scenarios and the corresponding data series that we intend to plot.

5.3.4 Synthetic results

In this section, we display and comment a number of graphs showing some aggregate metrics computed over several scenario simulations. In all the graphs, on the X axis we have the number of interested parties in logarithmic scale. Also, each single data point represents ten simulations of the same scenario and shows the average value and in some graphs the standard deviation. Note that the absolute values of the Y axis are somewhat irrelevant because they do not correspond to any real-world unit of measure. Instead, what we want to show are the trends of a metric with respect to the growth of interested parties (our primary measure of scale), or the relative positions of different curves representing different architectures.

In the graphs below we used the following aliases for event-service architectures: *ce*=centralized, *hs*=hierarchical (with subscriptions forwarding), *as*=acyclic peer-to-peer with subscriptions forwarding, *aa*=acyclic peer-to-peer with advertisements forwarding.

Comparison on total cost

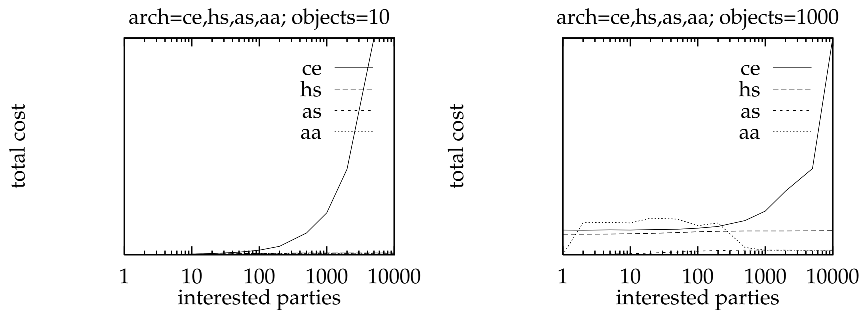

Figure 5.6: Total cost: comparison of architectures (*ce*,*hs*,*as*,*aa*)

Figure 5.6 compares the total cost of the simulation of a network of 100 sites with a population of 100 objects of interest. Clearly, the behavior of every distributed architecture (*hs*,*as*, and *aa*) is compressed as opposed to the explosive growth of the centralized architecture. We have obtained the same result when scaling up the number of objects of interests.

In order to be able to see the difference among distributed architectures, we eliminate the data serie of the centralized architecture and we obtain what is shown in Figure 5.7 and Figure 5.8.

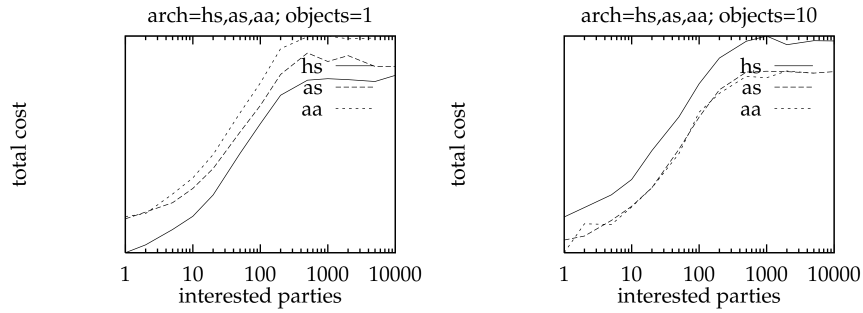

Figure 5.7: Total cost: comparison of architectures (*hs*,*as*,*aa*) with 1 and 10 objects

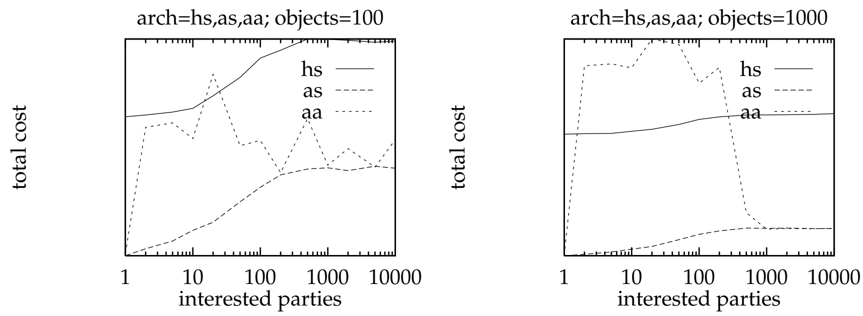

Figure 5.8: Total cost: comparison of architectures (*hs*,*as*,*aa*) with 100 and 1000 objects

What we see in figures 5.7 and 5.8 is that, except for the first chart that shows scenarios with only one object of interest, the hierarchical architecture with subscription forwarding is always more expensive than the acyclic topology with the same subscription forwarding algorithm. This behavior is consistent with what we can infer from the algorithms, in fact, while the peer-to-peer topology is penalized by its broadcast of subscription, the hierarchical algorithm —which propagates subscriptions only towards the root of the hierarchy— is also forced to propagate notifications towards the root server whether or not interested parties exist on that part of the network. This generates a traffic of unnecessary notifications.

The second observation that we can make by looking at these charts is that the advertisement forwarding algorithm has an irregular behavior for a low density of interested parties. Its cost returns to the level of the corresponding architecture with subscription forwarding when the number of interested parties saturates the network (i.e., more than one interested party per site). This effect becomes more and more evident as we increase the number of objects of interest.

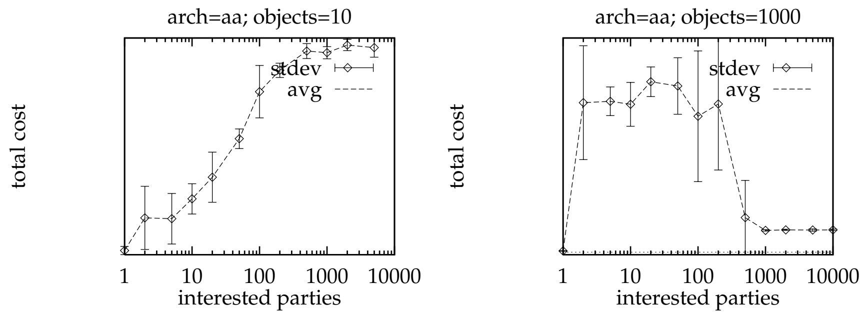

Figure 5.9: Total cost: acyclic peer-to-peer with advertisement forwarding

This unstable behavior at lower densities of interested parties is also evident in the chart of Figure 5.9. There we isolate the total cost of the acyclic peer-to-peer topology with advertisements forwarding. We also show error bars that report the standard deviation of this metric computed over ten runs of the same scenario.

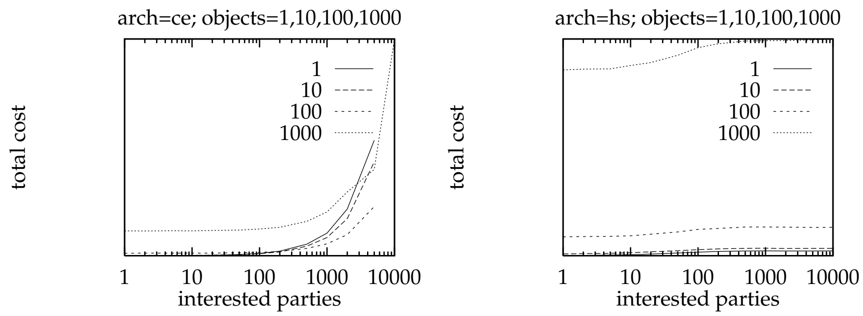

Figure 5.10: Total cost: centralized and hierarchical topologies

We can also examine the behavior of every single architecture with respect to the total cost. Figure 5.10 reports the behavior of the centralized architecture and the hierarchical topology with subscription forwarding. Figure 5.11

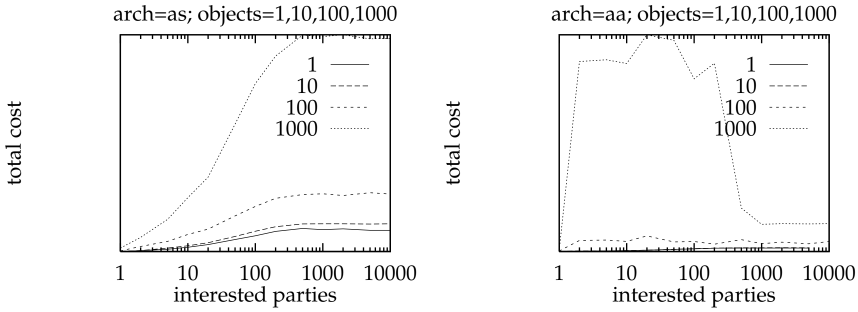

Figure 5.11: Total cost: acyclic peer-to-peer topology with advertisement forwarding and subscription forwarding

reports the behavior of the acyclic peer-to-peer topology with both subscription forwarding and advertisement forwarding. Remember that the values are normalized with respect to the maximum value in each chart, however, the absolute values of different charts are not related.

Distribution of load among sites

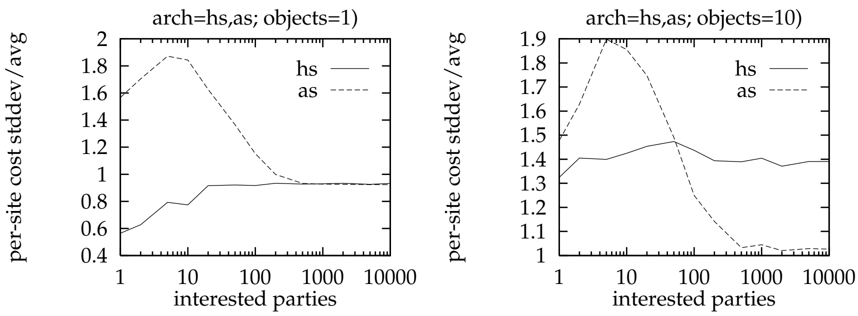

Figure 5.12: Variance of per-site cost: hierarchical vs. acyclic peer-to-peer (1,10 objects)

There is another fundamental difference between the hierarchical topology and the acyclic topology that also confirms our intuitions. The acyclic topology does a better job in distributing the load over all the servers in the net.

In Figure 5.12 and in Figure 5.13 we plot the standard deviation of the per-site cost, normalized with respect to the average per-site cost. For small numbers of objects of interest, i.e., when there are a few emitters of notifications,

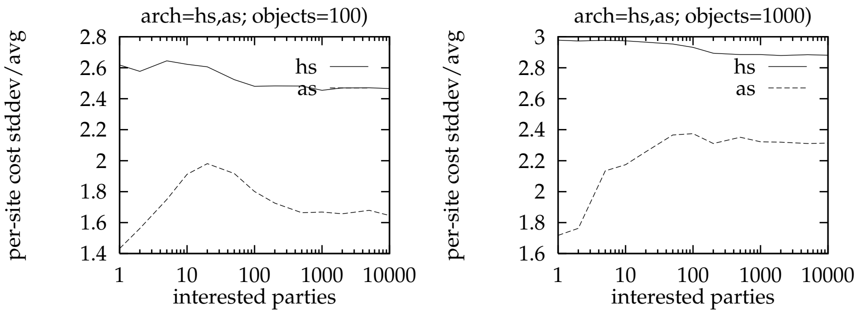

Figure 5.13: Variance of per-site cost: hierarchical vs. acyclic peer-to-peer (100,1000 objects)

and for small numbers for interested parties, the hierarchical topology has a better distribution of costs among sites. This is probably due to the overhead of broadcasting subscriptions in the acyclic topology. However, when scaling up interested parties and objects of interest, the acyclic topology reacts with a more homogeneous distribution of communication load.

Cost per service

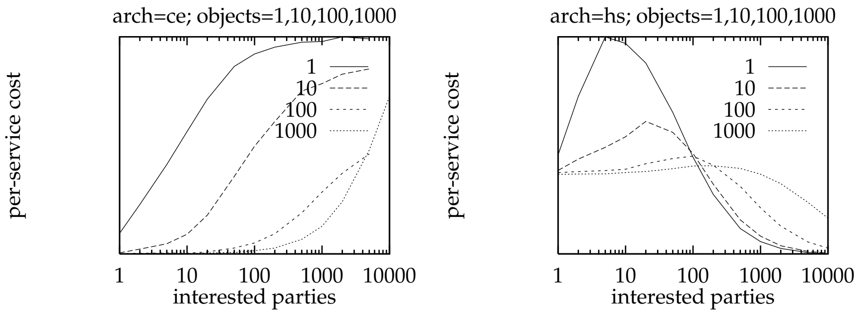

Figure 5.14: Average cost per service: scalability of every architecture (*ce*, *hs*)

The total cost metric hides the overhead introduced by the routing messages sent by event servers with respect to the messages that are strictly required for the implementation of each client request. The per-service grouping criterion tries to give a better characterization of the cost of architecture. More specifically, an event service is efficient if it can amortize the cost of the realization of a client request with the information set up by previous requests.

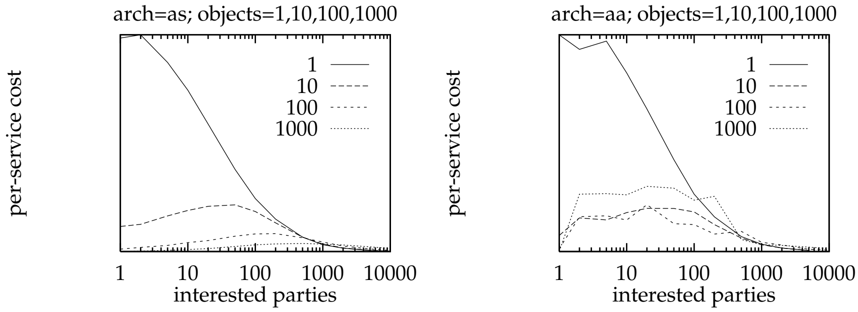

Figure 5.15: Average cost per service: scalability of every architecture (*as*, *aa*)

Figure 5.14 and Figure 5.15 show how every architecture scales up by amortizing the per-service cost. Figure 5.16 and Figure 5.17 examine the same data the

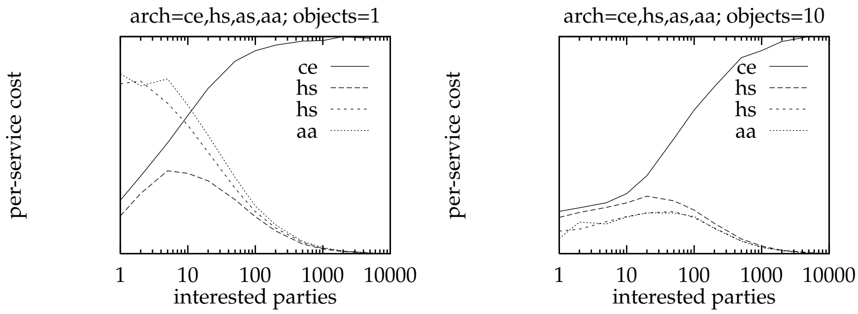

Figure 5.16: Total cost: comparison among architectures (1,10 objects)

other way around, i.e., by comparing the behavior of different architectures at various levels of scale.

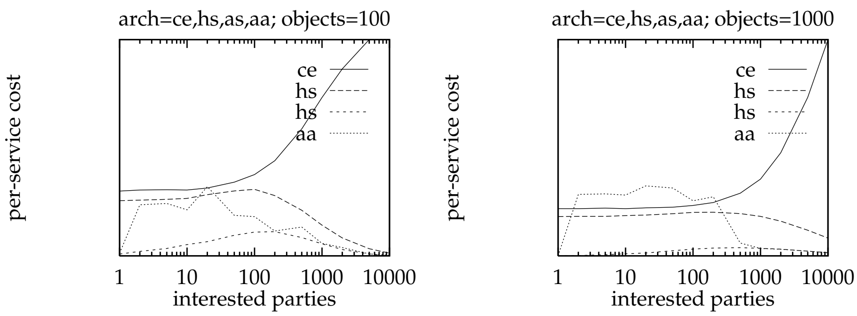

Figure 5.17: Total cost: comparison among architectures(100,1000 objects)

Chapter 6

Conclusions and Future Work

Software systems of a significant dimension, especially those that are distributed over a computer network, are often engineered by means of the integration of *components*. A promising approach to support component-based software architectures is the so-called *event-based style* whereby the interaction of components is modeled with *events*. Components emit events to inform other components of a change in their internal state or to request services from other components. Upon detecting the occurrence of events, components react by executing some actions and possibly emitting other events. The glue that ties components together in an event-based architecture is an infrastructure that we call *event service*. The event service registers the interests of components and then dispatches event notifications accordingly. The advantage of using an event service instead of other "classical" integration mechanisms such as direct or remote invocation is that this method increases the degree of de-coupling among components thus eliminating static dependencies and improving interoperability.

We envision a ubiquitous event service, accessible from every site on a wide-area network, and particularly suitable for supporting highly distributed applications that require a fine grained interaction. Such an event service complements other middleware services such as point-to-point communication mechanisms by offering a many-to-many communication and integration facility. In this dissertation, we presented the project of an event service called SIENA that has precisely these goals.

The focus of SIENA is on scalability of the service and on the trade-offs existing between the expressive power of the event service and its ability to scale up. In particular, SIENA has been designed to absorb gracefully the network traffic induced by the explosion of connections between components established in the event-based interaction at the scale of a wide-area network such as the Internet.

Summary of the work done

The first step in this research is the formal definition of the SIENA event service which includes the formulation of a conceptual framework, the specification of the data model, the interface, and the semantics of the event service. In particular, we extended the publish/subscribe framework with another primitive called *advertise*, we introduced the observation of sequences of events, and we defined two alternative semantics, one based on subscriptions, the other one based on advertisements. The critical point in this phase was to devise a service flexible enough to balance the modeling capabilities offered to the application designer with the opportunities to apply optimizations to reduce the communication overhead introduced with the implementation of the service.

The following step is the realization of the event service as a distributed structure of *event servers*. This effort lead to the definition of a number of alternative implementations, each one having a different architecture with different requirements on the semantics of the service and different features for scalability. Every architecture results from the composition of a topology for the layout of the network of event servers with an algorithm for the dispatching of event notifications. We initially formulated two basic optimization principles that allow the amortization of communication overhead and resource consumption. Then we applied these principles to each topology. In this phase, we give a detailed description of the algorithms and data structures that realize the dispatching and the recognition of sequences of events. The algorithms also show the critical aspects of the event model and of the event service interface with respect to scalability.

The last part of this work is dedicated to the quantitative assessment of the proposed solutions. In line with other research efforts in communication networks, we used a simulation environment that served us both as a design and development tool, to verify and validate algorithms and topologies, and as a model for the evaluation of our scalability goals. The construction of the simulation environment involved the implementation of a network simulator specially targeted at application-level simulations. For the assessment of SIENA we utilized randomly generated networks, populated by several types and densities of applications, served by event services with different architectures. The results that we obtained show that all our distributed architectures are able to compress the communication overhead generated in a reference centralized architecture. In particular, when scaling up the number of interested parties and objects of interest, the new distributed topologies that we propose amortize the cost of services much more than the centralized architecture that has an explosive behavior, and also more than the hierarchical architecture that is adopted in several distributed systems similar to SIENA. Another quite evident result differentiates the hierarchical topology from the acyclic one by showing how the latter one does a better job in distributing the load over all the event servers.

Future work

This thesis is an initial step towards the realization of the quite ambitious goals of SIENA. Much work remains to be done to improve what we have achieved so far and to explore new solutions. As far as the capabilities of the event service, we have the opportunity to enrich the vocabulary of operators available for single filters as well as for the combination of patterns. While new filters operators pose little conceptual and practical problems, new combinators of patterns are likely to have a significant impact on the algorithms that realize the factoring and delegation of the observation of sequences. We already outlined some alternative heterogeneous topologies and some additional optimization strategies that could improve the dispatching and monitoring algorithms. These could be implemented and new ones can be devised. In particular, we would like to improve the compression of routing tables on event servers.

An area of this work that definitely needs a deeper understanding and more research is the assessment of performances. The experiments we ran so far demonstrate that our approach is on the right track, however the quantitative results are either not very surprising or not so informative for determining the variables of control for scalability. We will work on a more comprehensive evaluation of the architecture with more systematic simulations, trying to isolate the factors that affect scalability. Ultimately, we would like to distill a set of guidelines that characterize clearly the trade-offs between applications' needs and scalability of the event service.

Apart from the work with the simulated environment, we would like to gain more real-world experience. We indicate here some plans for future works that are well out of the scope of this this thesis and might follow up from this research. The first rather obvious idea is to realize a real implementation of SIENA beyond the current prototype. This means having an event server, possibly implementing all the different architectures we studied, that we can deploy on a significant number of nodes on the Internet. Another interesting development would be the study of the application side. This includes a characterization of the requirements that applications pose on an event service and ultimately a methodology for the design and implementation of event-based component-based software systems.

Bibliography

[1] J. Ahn and P. B. Danzig. Speedup vs. Simulation Granularity. *IEEE/ACM Transactions on Networking*, 4(5):743–757, Oct. 1996.

[2] R. L. Bagrodia, K. M. Chandy, and J. Misra. A Message-Based Approach to Discrete-Event Simulation. *IEEE Transactions on Software Engineering*, 13(6):654–665, June 1987.

[3] R. L. Bargodia and W.-T. Liao. Maise: A language for the design of efficient discrete-event simulation. *IEEE Transactions on Software Engineering*, 20(4), Apr. 1994.

[4] N. S. Bargouti and B. Krishnamurthy. Using Event Contexts and Matching Constraints to Monitor Spftware Processes Effectively. In *Proceedings of the 17th International Conference on Software Engineering*, Seattle WA, U.S.A., May 1995. IEEE Computer Society.

[5] 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 Requests For Comments (RFC) 1630, June 1994.

[6] D. Barrett, L. Clarke, P. Tarr, and A. Wise. A Framework for Event-Based Software Integration. *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, Oct. 1996.

[7] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[8] D. Bertsekas and R. Gallager. *Data Networks*. Prentice–Hall, Englewood Cliffs, New Jersey, 1987.

[9] O. Burkart. *Automatic verification of sequential infinite-state processes*, volume 1354 of *Lecture Notes in Computer Science*. Springer–Verlag, New York, 1997.

[10] O. Burkart and B. Steffen. Model Checking the Full Modal Mu-Calculus for Infinite Sequential Processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, Proceedings of 24th International Colloquium ICALP '97*, number 1256 in Lecture Notes in Computer Science, pages 419–429, Bologna, Italy, July 1997. Springer-Verlag.

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

[12] A. Carzaniga, E. Di Nitto, D. S. Rosenblum, and A. L. Wolf. Issues in Supporting Event-based Architectural Styles. In *3rd International Software Architecture Workshop*, Orlando FL, U.S.A., Nov. 1998.

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

[14] D. Clark. Policy Routing in Internet Protocols. Internet Requests For Comments (RFC) 1102, May 1989.

[15] D. Clark and e. a. Joseph Pasquale. Strategic Directions in Networks and Telecommunications. *ACM Computing Surveys*, 28(4):679–690, Dec. 1996.

[16] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):191–230, July 1998.

[17] D. H. Crocker. Standard for the Format of ARPA Internet Text Messages. Internet Requests For Comments (RFC) 822, Aug. 1982. STD 11.

[18] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 98)*, Kyoto, Japan, Apr. 1998.

[19] 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, Milano, Italy, Sept. 1998.

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

[21] S. E. Deering. Host Extensions for IP Multicasting. Internet Requests For Comments (RFC) 1584, Aug. 1989.

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

[23] 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.

[24] S. E. 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, Apr. 1996.

[25] M. B. Doar. A Better Model for Generating Test Networks. In *Proceedings of Globecom '96*, Nov. 1996.

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

[27] W. Fenner. Internet Group Management Protocol, Version 2. Internet Requests For Comments (RFC) 2236, Nov. 1997.

[28] C. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–84, Nov. 1996.

[29] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, 1963.

[30] A. Fuggetta, G. P. Picco, and G. Vigna. Undestanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[31] Georgia Institute of Technology. College of Computing. Georgia Tech Internet Topology Models (GT-ITM).
http://www.cc.gatech.edu/projects/gtitm.

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

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

[34] C. Hedrick. Routing Information Protocol. Internet Requests For Comments (RFC) 1058, June 1988.

[35] P. Huang, D. Estrin, and J. Heidemann. Enabling Large-Scale Simulations: Selective Abstraction Approach to The Study of Multicast Protocols. In *Proceedings of the 6th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS98)*, pages 241–248, Montreal, Canada, July 1998.

[36] K. Ilgun, R. Kemmerer, and P. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3), Mar. 1995.

[37] D. B. Johnson and C. Perkins. Mobility Support in IPv6. Internet Draft, Nov. 1997. Mobile IP Working Group.

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

[39] B. Kantor and P. Lapsley. Network News Transfer Protocol – A Proposed Standard for the Stream-Based Transmission of News. Internet Requests For Comments (RFC) 977, Feb. 1986.

[40] Keryx WEB page. http://keryxsoft.hpl.hp.com, 1997.

[41] Kerysoft, Hewlett Packard. *Keryx Version 1.0a Release Notes and Documentation*, 1997. http://keryxsoft.hpl.hp.com/keryx-1.0a/html/index.html.

[42] D. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison–Wesley, Reading, Massachusetts, 1994.

[43] E. Koutsofios and S. C. North. *Drawing Graphs with* dot. AT&T Bell Laboratories, Murray Hill NJ, U.S.A., October 1993.

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

[45] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[46] J. Levine. An Algorithm to Synchronize the Time of a Computer to Universal Time. *IEEE/ACM Transactions on Networking*, 3(1):42–50, Feb. 1995.

[47] J. C. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. In *Proceedings of the IEEE INFOCOM '96*, pages 1414–1424, San Framcisco CA, U.S.A., Apr. 1996.

[48] S. Maffeis. iBus: The Java Intranet Software Bus. Technical report, Soft-Wired AG, Zurich, Switzerland, Feb. 1997.

[49] 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.

[50] D. L. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. Internet Requests For Comments (RFC) 1305, Mar. 1992.

[51] D. L. Mills. Improved Algorithms for Synchronizing Computer Network Clocks. *IEEE/ACM Transactions on Networking*, 3(3):245–254, June 1995.

[52] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–65, Mar. 1986.

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

[54] P. Mockapetris. Domain Names - Implementation And Specification. Internet Requests For Comments (RFC) 1035, Nov. 1987.

[55] J. Moy. Multicast Extensions to OSPF. Internet Requests For Comments (RFC) 1075, Mar. 1994.

[56] B. Mukherjee, L. Heberlein, and K. Levitt. Network Intrusion Detection. *IEEE Network*, pages 26–41, May 1994.

[57] D. R. Musser and A. Saini. *STL tutorial and Reference Guide*. Addison–Wesley, Reading, Massachusetts, 1996.

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

[59] OSF, editor. *OSF/Motif Programmers Guide*. Prentice Hall, Englewood Cliffs, 5 edition, 1991.

[60] V. Paxson. End-to-End Routing Behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, Oct. 1997.

[61] V. Paxson. Why We Don't Know How to Simulate the Internet. In *Proceedings of the 1997 Winter Simulation Conference*, Atlanta GA, U.S.A., Dec. 1997.

[62] C. Perkins. IP Mobility Support. Internet Requests For Comments (RFC) 2002, October 1996. Standards Track.

[63] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, Sept. 1991.

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

[65] M. T. Rose. *The Simple Book*. Prentice–Hall, Englewood Cliffs, New Jersey, 1991.

[66] D. R. Rosenblum, A. L. Wolf, and A. Carzaniga. Critical Considerations and Designs for Internet-Scale, Event-Based Compositional Architectures. In *Workshop on Compositional Software Architectures*, Monterey CA, U.S.A., Jan. 1998.

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

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

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

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

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

[72] Sun Microsystems, Inc., Mountain View CA, U.S.A. *JavaBeans 1.01 Specification*, 1998.

[73] TIBCO Inc. Rendezvous Information Bus. http://www.rv.tibco.com/rvwhitepaper.html, 1996.

[74] TIBCO Inc., Palo Alto CA, U.S.A. *TIBR+: a WAN Router for Global Data Distribution*, 1996.

[75] G. Vigna and R. Kemmerer. NetSTAT: A Network-based Intrusion Detection Approach. In *Proceedings of the* 14^{th} *Annual Computer Security Application Conference*, Scottsdale AZ, U.S.A., Dec. 1998.

[76] The VINT Project—UC Berkeley, LBL, USC/ISI, and Xerox PARC. ns *v2—Notes and Documentation*, Nov. 1997.

[77] D. Waitzman, C. Partridge, and S. E. Deering. Distance Vector Multicast Routing Protocol. Internet Requests For Comments (RFC), Nov. 1988.

[78] B. M. Waxman. Routing of Multipoint Connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, 1988.

[79] B. Whetten, T. Montgomery, and S. Kaplan. A High Performance, Totally Ordered Multicast Protocol. In K. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science. Springer–Verlag, New York, 1995.

[80] Workshop on Internet Scale Event Notification (WISEN). http://www.ics.uci.edu/IRUS/wisen/wisen.html, July 13–14 1998.

[81] M. Wray and R. Hawkes. Distributed Virtual Environments and VRML: an Event-based Architecture. In *Proceedings of the Seventh International WWW Conference (WWW7)*, Brisbane, Australia, 1998.

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

[83] E. W. Zegura, K. L. Calvert, and M. J. Donahoo. A Quantitative Comparison of Graph-based Models for Internet Topology. *IEEE/ACM Transactions on Networking*, 5(6), Dec. 1997.