UCI-ISR-04-1 April 2004

Design and Experiments with YANCEES, a Versatile Event Notification Service

Roberto S. Silva Filho [1] Cleidson R. B. de Souza [1,2] David F. Redmiles [1]

[1]Department of Informatics
University of California, Irvine
Irvine, CA, USA
+1 949 824-4121

[2]Departamento de Informática
Universidade Federal do Pará
Belém, PA, Brasil
+55 91 211-1405

{rsilvafi, cdesouza, redmiles}@ics.uci.edu

ABSTRACT

Publish/subscribe infrastructures, specifically event notification services, are used as the basic communication and integration framework in many application domains. The majority of these services, however, provide poor or no extension mechanisms as well as insufficient configuration capabilities. As a consequence, different event notification servers have been developed previously to support the requirements from different application domains, resulting in implementations that are almost always incompatible with one another. They lack mechanisms that allow their use in multiple hardware and software platform configurations, and the flexibility to support different application domains, which have requirements in constant evolution. The YANCEES (Yet ANother Configurable Extensible Event Service) framework was designed to address these versatility issues, relying on a pluggable architecture. We demonstrate our approach, showing how the YANCEES framework can be used in the implementation of different services to attend the demands of many application domains.

Keywords
Publish/subscribe systems, versatility, notification servers, event-based middleware, dynamic architecture, pluggable architecture, extensibility.

1. INTRODUCTION

In recent years, the publish/subscribe paradigm has provided the foundation for many distributed software applications including user and software monitoring [1], groupware [2], collaborative software engineering [3], workflow management systems [4], mobile applications [4], among others. As a consequence, many notification servers have been developed, providing services and special features to support the needs of each application domain.

The space of research or commercial notification services is relatively large. At one extreme, "one-size-fits-all" infrastructures such as the CORBA Notification Service (CORBA-NS) [5] or READY [6] provide a very comprehensive set of features that support a broad set of applications. At another extreme, domain-specific notification servers provide special functionalities to support their target applications. Examples of such systems include Khronika [7] and CASSIUS [8], which are especially designed to support groupware and awareness applications; or also Yeast [9] and GEM [10], designed to support advanced event processing for local networks and distributed applications monitoring, respectively. More recently, distributed publish/subscribe services such as Siena [11] and Elvin 4 [12] provide an intermediate approach. They achieve a trade-off between specificity and expressiveness of the subscription language, required by Internet-scale event notification services.

In spite of the availability of such a range of notification servers, including standardized solutions such as CORBA-NS [5] or JMS (Java Message Service API) [13], new notification servers continue to be developed to address the needs of novel applications. There are a few reasons. First, beginning with the most straightforward reason, the publish/subscribe paradigm appears seductively simple. A basic service can be programmed quickly before the complexities of the application it serves reveal themselves, then forcing extensive extensions already designed into sophisticated servers. Less obvious, a second deterrent is that current publish/subscribe infrastructures are not designed to be extensible, which hinders addition or customization of services to support new applications. For example, CORBA-NS does not support mobility protocols, and the addition of such feature usually requires changing its source code or the implementation of this concern by the client application. Third, with rare exceptions such as READY [6], current solutions are not configurable with respect to both set of features they provide, or the place where event processing happens in a distributed setting. For example, applications such as software monitoring [1], require the execution of event processing in the application site where the events are collected, whereas applications running on mobile devices may need a restricted set of services and components.

Hence, as a result of this lack of versatility, when new application requirements need to be provided, developers usually have three options: (1) build a new publish/subscribe infrastructure from scratch, with almost no reuse of existing components; (2) use an existing solution and implement the required functionality as part of the client application, which makes the functionality difficult to be reused, or (3) if the source code is available, build this new feature in the notification service itself. Since notification services are not usually designed with extensibility as one of their primary requirements, this may not be an easy task. Another side effect of this continuous recreating of new solutions is the proliferation of incompatible infrastructures that do not interoperate. The integration of such systems, to satisfy large organizations demands, requires an additional development effort. In other words, current publish/subscribe systems lack versatility, i.e. the ability to cope with different application requirements, by being able to change and evolve according to their requirements.

In this paper we present the YANCEES (Yet Another Extensible Event Service) framework, a new and experimental infrastructure designed to provide versatility to event notification services. YANCEES is based on the key observations that publish/subscribe systems can have their design dimensions generalized, following an extended version of the conceptual design framework proposed by Rosemblum and Wolf [14]. Even though Rosemblum and Wolf's model describes all the basic interactions involved in a publish/subscribe service, it lacks a dimension to express other services demanded by current notification services such as mobility, persistency, registration, authentication and so on. To capture that fact, we extend this model to include a protocol dimension. Another key observation in our approach is that, in a publish/subscribe system, the information follows a process of publication, routing (based on subscriptions) and notification, that can also support additional protocols. This process defines specific extension points where plug-ins can be installed. Moreover, subscription, notification and protocol functionality can be represented as a composition of functionality provided by different plug-ins, in an incremental way, allowing the reuse and combination of those components, on demand. Another key aspect of our framework is that current publish/subscribe infrastructures can be wrapped and extended using abstraction layers. This modular approach makes possible the fast extension and configuration of YANCEES to attend different applications. Therefore, the main benefit of the use of a common, configurable and extensible framework is an incremental framework where new functionality can be added reusing existing plug-ins to support the constant changes required by today's constant and fast-paced software evolution. Additionally, the common model provided by YANCEES allows the framework to be used as an integration mechanism among different event notification services. Finally, we state *versatility* as a goal in this work because our we see our primary, long-term goal as providing a notification service that is adaptable to human concerns in different collaborative, workplace settings, as well as a service that is applicable to varied distributed system architectures.

2. BACKGROUND

Notification servers implement the distributed implicit invocation design pattern, brokering the communication between publishers (information producers) and subscribers (information consumers). This architectural style allows the insulation between these two kinds of components, coping with the dynamism, heterogeneity, scalability and changeability requirements of many distributed applications. It also facilitates the combination and integration of information from different sources as they get produced. Publish/subscribe assume the existence of two main roles, at one side: publishers produce pieces of information that are sent, or published, to a logically-centralized service, also called notification service; on the other side, information consumers, or subscribers that express their interest on those pieces of information, which is usually performed using subscriptions. Subscriptions are generally represented as logical expressions that operate over the content, type and order of events. More advanced systems allow more expressive constructs, permitting the correlation of events, for example. Hence, in a publish/subscribe system, whenever an event is published, it is matched against the available subscriptions posted in the endpoints of the service. Whenever a subscription is matched, the event is routed to the subscriber as a notification.

Publish/subscribe systems evolved over time, incorporating new functionality and interaction models. First generation systems use either group based (also known as channel-based) or subject-based (also known as topic-based) event dispatching mechanisms. In the former approach, producers broadcast events to groups or channels, whereas consumers subscribe to one or more of these channels to receive events. An example of such systems is the CORBA Event Service (CORBA-ES) [15]. The second approach, subject-based, is more flexible: each event is annotated with a special field, usually a string or token, called subject (or topic) to describe its content. This special field allows filtering of events, implementing a fast switching mechanism on top of event channels. An example of such systems is the Talarian SmartSockets [16] and the JMS [13] implementations. These systems provide fast switching channels between information producers and consumers. They are mainly used as distributed application connectors where additional services such as transactions, persistency and filtering can be provided.

Recently, a more general mechanism called content-based routing (or dispatching) has been used. Content-based services allow event consumers to perform advanced queries over the whole content of an event or sets of events. For their sophistication and generality these systems usually face a trade-off between expressiveness and efficiency [17]. They usually have to process, route and combine events coming from different sources. Examples of such systems include Siena [11], Jedi [4] and Elvin [12]. Their expressiveness allows the construction of more sophisticated applications where filtering of information is important as for example, awareness, security, monitoring, content management and so on.

While many publish/subscribe systems provide a similar set of features, the publication and subscription of events, they differ in the specific set of features they provide. While systems as Siena are focused on internet scalability, other systems as CASSIUS and Khronika are designed to address specific issues of collaborative applications such as awareness. In other words, their functionality is mainly motivated by the application domain's requirements they serve. These requirements usually demand very specific services related but not limited to: performance, expressiveness, scalability, mobility or other very specific issues such as event source browsing for awareness. Most of those different publish/subscribe implementations do not interoperate and are usually not adequate in serving other applications needs. On the other hand, the use of general-purpose solutions such as CORBA-NS, even though provides a standard model that supports interoperability, usually does not address the specific requirements that many applications require.

Therefore, as can be observed, versatility is usually not a primary concern in current publish/subscribe systems. This drastically limits the application domains each notification service can serve, and also results in extra and repetitive effort each time a new functionality needs to be supported. Extensibility is usually a matter delegated to the application level and configurability is usually not provided.

3. YANCEES' REQUIREMENTS

In striving to address an extensive set of good software engineering properties and requirements, we adopted the term versatility for its ability to express these concerns in a generic way. Moreover, we sought a new term that we could use to imply that these properties involved human stakeholders and application workplace settings, an on-going challenge we set for our work. As defined by The American Heritage Dictionary, versatility is the *"quality of having varied uses or serving many functions"*. From a software engineering perspective, we express versatility in publish/subscribe infrastructures as a set of requirements as follows:

Extensibility. The ability to augment the infrastructure with new functionality. Extensibility is required in a constantly evolving environment, motivated by the demand of new technologies and services. Currently, the most common way to extend a publish/subscribe infrastructure is by directly changing its source code or by providing the required features as part of the client application.

Functional Configurability. The ability to combine and select different functionality to attend the needs of specific application domains. Of importance is not only the ability to extend the system, but also the ability to customize which features to support, as a result of hardware and software constraints. Currently, for example, neither "one-size-fits-all" solutions such as CORBA-NS nor application-specific systems such as Khronika and Yeast provide support for this requirement. . Note also that some functionally are dependent on others. For example, in order to support pull notifications, a server need to provide persistency mechanisms. Being able to represent these functional dependencies is important so that one might later reuse these functionality.

Distribution Configurability. With rare exceptions, such as READY [6], current event notification services provide no support for selecting which components to execute on the client side (where event producers and consumers are located) and which ones to execute on the server side. This requirement is especially important for mobile applications, which may not be in contact with a central server all the time, and some software monitoring tools, such as EDEM [1], which collects and processes end-user events on the client side, minimizing network traffic.

Interoperability. Because different event notification infrastructures are used to support heterogeneous application domains, interoperability may become an issue. For example, in a large organization, different subscription, notification, and protocol requirements need to coexist and inter-operate. The lack of interoperable formats in current notification servers forces the use of different services by different applications. As a consequence, the integration of these services may become a non-trivial task.

Reusability. Every time a new set of requirements is needed from a publish/subscribe service, a new system is usually built from scratch. Common features to most of current systems, such as content-based routing, event representation, and push or pull notification models are included in this effort. More advanced features such as event correlation, persistency and others are usually required and need to be re-implemented as well. Currently, there is few or no support for the reuse of such functionality, or for their incremental incorporation in the publish/subscribe service.

Usability. As described by Nielsen [18], "different users have different needs." This user-centered view requires different services and functionality from the underlying infrastructure. For example, services such as GEM are designed to be used by system administrators and programmers, that directly use the facilities of this language, whereas CASSIUS and Khronika were designed to be used by groupware applications end-users, that interact with the system through GUIs that hide the complexity associated to the operation of the system.

In the scope of this paper, we show how these requirements are achieved, with the exception of the usability requirement. As the title of the paper suggests, the addressing of such software qualities is still on process, and we plan to achieve all of them in a close future.

In striving to achieve the requirements described above, we designed a versatile framework based on the concept of plug-ins that can be installed in the different design dimensions of a publish/subscribe service. These dimensions are discussed in the next session.

4. ANALYTICAL DESIGN FRAMEWORK

In the YANCEES framework, the extensibility points are inspired in an extended version of the framework proposed by Rosenblum and Wolf [14]. Using this framework allows us to address and analyze the basic aspects that need to be considered in the design of an event-notification server. In this framework, the object model describes the components that receive notifications (subscribers) and generate events (publishers). The event model describes the representation and characteristics of the events; the notification model is concerned with the way the events are delivered to the subscribers; the observation model describes the mechanisms used to express interest in occurrences of events; the timing model is concerned with the casual and temporal relations between the events; the resource model defines where, in the distributed system architecture, the observation and notification computations are located, as well as how they are allocated and accounted; finally, the naming model is concerned with the location of objects, events, and subscriptions in the system.

This model, however, does not consider additional services a publish/subscribe system can provide. Hence, in the design of YANCEES, we considered a variation of this model proposed by Cugola et al [4], with the addition of a new model, the protocol. The protocol model is defined as a way to address the need to capture other forms of interaction with the notification service that goes beyond the common publish/subscribe interaction. This extension to Rosenblum's and Wolf's framework is necessary to express the ability that a notification services has to handle different functionality other than the common publish and subscribe activities, such as: guaranteed delivery, mobility and roaming protocols, security messages, event source discovery primitives and other possible interaction mechanisms with the service. As proposed by Cugola and colleagues., we combine the naming and observation models in the subscription model.

5. APPROACH

The YANCEES architecture defines different extension points based on the dimensions described in our design framework. It is based on the key observation that the design dimensions of a publish/subscribe system express both a process where events are routed from producers and consumers, and extension points where new functionality can be plugged. YANCEES makes these extension points explicit, allowing the addition of subscription, notification, protocol and event components that, together, compose the whole functionality of an application-specific publish/subscribe service.

This design model assumes an underlying process of publication, routing (using subscriptions), and notification. In our approach, an important aspect of these three dimensions is the use of languages to describe events, to post subscription queries and to specify notification policies. This is also true for the protocol model, which expresses sets of primitive commands that the service responds to. Hence, the extensibility of such dimensions requires a dual aspect: the extension of the language and the implementation of these extensions as a plug-in. A general picture of the YANCEES framework is shown in Figure 1 as follows.

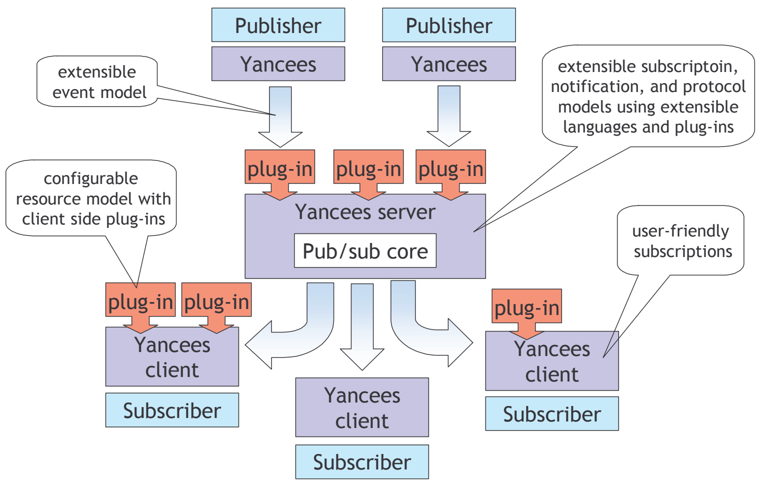

Figure 1: General view of the YANCEES framework

In the YANCEES framework, plug-ins represent modules that implement a particular and unique functionality. They can be dynamically composed to extend the subscription, notification and protocol languages. For example, plug-ins can be defined to perform event correlation (sequence detection, aggregation, abstraction and so on), notification policies (such as push and pull), and protocols (mobile primitives such as move-in/move-out). Plug-ins can be either installed in the server side or in the client side (publishers or subscribers), allowing the distribution of event processing functionality among these sites. By doing that, we are addressing an important requirement of versatile event-notification server systems, the distribution configurability (see section 3).

5.1 Internal YANCEES architecture

Internally, extensibility and configurability in the YANCEES framework are achieved by the use of three kinds of components: plug-ins, filters and services, and three kinds of languages: subscription, notification and protocol. The main internal components of the architecture are presented in Figure 2.

Subscription, notification and protocol plug-ins. Plug-ins are the dynamic components of the subscription, notification and protocol design models. Plug-ins implement specific command sets in those languages. This fine-grained extension approach improves the reuse of existing plug-ins and permits the dynamic composition of these components. This approach also guarantees that only the components used by active subscriptions are loaded in memory at one point.

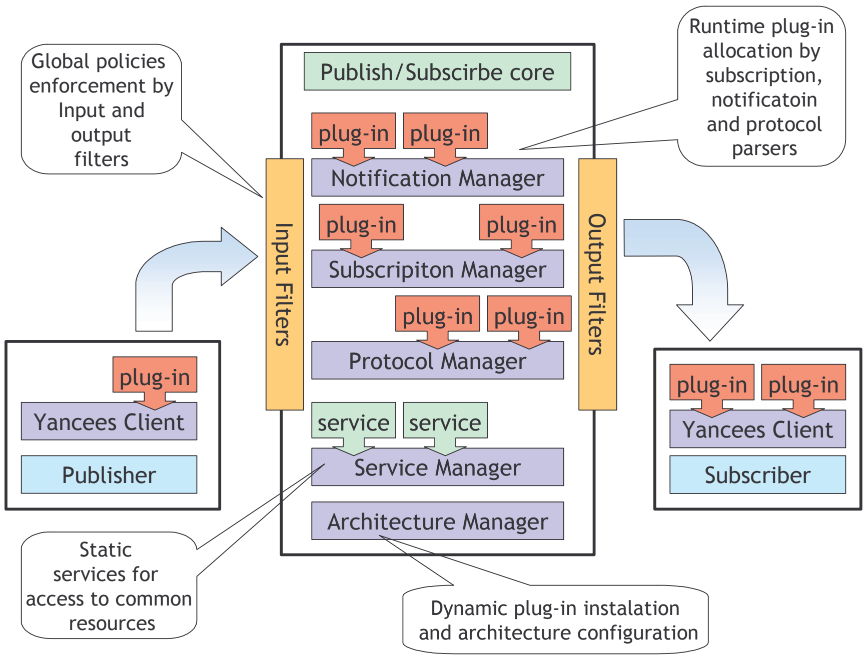

Figure 2 General components of the architecture

Dynamic parsers. Since events, subscriptions and notifications are expressed in terms of languages (end-user ways of interacting with the system), these languages need to be extensible, that is, able to incorporate new commands. For example, a new sequence detection subscription function may be expressed by a new <*sequence*> command. This command is added to the subscription language and needs to be implemented by a plug-in. Hence, there is a direct relation between the functional components (plug-ins) and the commands in this language. The binding between these two concerns is promoted by dynamic parsers (notification manager, subscription manager, protocol manager), according to the design dimension the plug-in will extend. An illustration of the dynamic parsing mechanism is presented in Figure 3. More details about the implementation of this dynamic binding are discussed at [19].

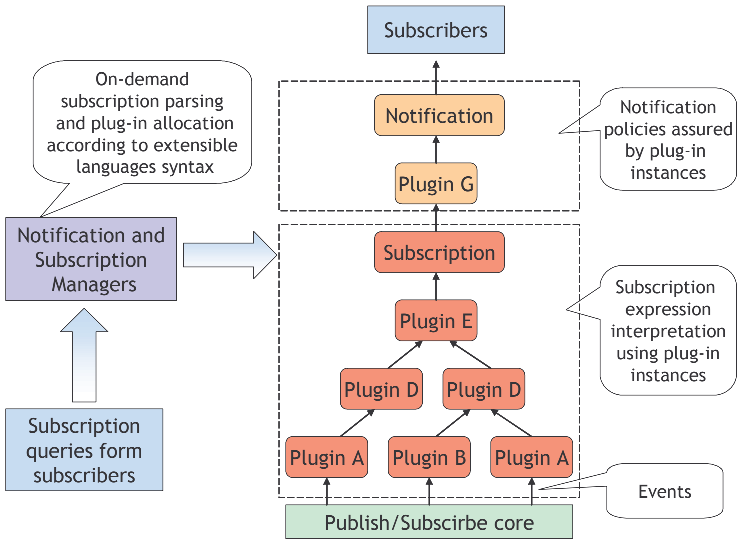
Subscribers
On-demand subscription parsing and plug-in allocation according to extensible languages syntax
Notification
Plugin G
Notification policies assured by plug-in instances
Notification and Subscription Managers
Subscription
Plugin E
Subscription expression interpretation using plug-in instances
Plugin D
Plugin D
Subscription queries form subscribers
Plugin A
Plugin B
Plugin A
Events
Publish/Subscirbe core

Figure 3 Dynamic composition of plug-ins to attend subscription and notification commands

In response to the posting of a new subscription message in the notification server, the subscription and notification managers allocate plug-in instances according to the subscription and notification commands requested. This creates an evaluation tree as described in Figure 3 (a subscription expression with an associated notification delivery policy is presented in Table 6 of section 8). In this tree, higher-level plug-ins depend on functionality provided by lower-level ones. For example, in this figure, the functionality of a plug-in of type E is dependent of the results provided by other plug-ins of type D, which, themselves, depend on the results from plug-ins of types A and B. An example of this composition is sequence detection. A <*sequence*> plug-in depends on results provided by two or more filters, that subscribe to events of certain types in the publish/subscribe core.

The use of dynamic parsers and subscription, notification and protocol plug-ins, in our approach, addresses the reuse and functional configurability requirements previously discussed in section 3.

Filters. Some functional extensions and policies may apply to all events being routed by the system, for example, security restrictions, event type checking, persistency, and so on. In order to facilitate the implementation of these extensions, filters can be used to inspect or restrict the flow of events both in the input and output phases of the processing. Filters can also be installed in the publisher and subscriber sites as a way to reduce the flow of events in the system. Filters are basically inserted one after another, according to the chain of responsibility design pattern [20], and are invoked at two moments, the publishing and the notification of the events.

Services. Services are special persistent components. Differently from plug-ins, that are dynamically allocated and composed on response to subscriptions, notifications and protocol requests, services are used to represent shared resources and to implement external application access points. For example, event persistency may require access to external databases; security protocols may require access to an authentication server, and so on. All these are implemented as services, which can be used by plug-ins and filters.

The components previously discussed address the extensibility of the publication, notification, subscription and protocol models. Orthogonal to these dimensions is the resource model that defines where, in the distributed architecture, the observation and notification computations are located, and how they are allocated and accounted. Performance or communication issues (in the case of mobility), may require the execution of some of the server functionality in the client or server sides. These requirements are addressed by the use of client (publishers or subscribers) components.

Finally, the event model is implemented by a combination of an event language, which describes how an event is represented, and the event dispatcher core, that performs the basic marshaling/un-marshaling and routing of the events, based on its content or other strategy, for example, using a topic-based or content-based approaches. The timing model, which deals with event ordering, is addressed by these dispatchers.

The ability to select which components (plug-ins, services and filters) to have in the framework and where to install them is orchestrated by the **configuration manager** component. Hence, in building a publish/subscribe system using YANCEES, a designer can define a combination of such components to provide the desired functionality set of the system. These components can also be distributed over client and server sites in order to distribute processing or to cope with special hardware or application limitations and requirements. An additional benefit of our approach is the ability to install plug-ins, services and filters at runtime. Since the dynamic parsers allocate the plug-ins on demand, plug-ins can be installed or upgraded as necessary, allowing the runtime evolution of the notification service functionality. This last process is coordinated by the architecture manager. A list of the main requirements addressed by our approach, and the strategies to achieve these properties is presented in Table 1 as follows.

Table 1`Summary of requirements and how they are addressed in our approach

Property	Approach
Extensibility	Extensible XML-based subscription, notification and protocol languages; implemented by plug-ins, filters and services; Support for dynamically updating or installing new functionality in the notification service.
Functional configurability	Support for different configurations: sets of plug-ins, filters, services and dispatcher components
Distribution configurability	Ability to define client-side (publisher and subscriber) or server-side plug-ins, filters and services;
Interoperability	Support for different notification servers, installed as dispatcher components.
Reuse	Support for building new functionality based on existing plug-ins, filters and services; Support for existing notification servers
Usability	***Future*** *support for GUI-based subscription editors and generic subscription and event programming language representations.*

5.2 Discussion

Clearly, successful operation of YANCEES depends on the way it is used. It is critical, for instance, that compatible plug-ins, services and filters be selected, since YANCEES itself does not detect when the selected components have conflicting semantics. It is currently the designer’s responsibility to assure a coherent selection of components.

Another challenge in using YANCEES is how to address crosscutting design concerns such as security, general quality of service, or even mobility. In our approach, these issues must be addressed by the coherent combination of components that, together provides the desired design concern. For instance, security may be achieved by combining dispatchers, filters, notification policies, and subscriptions that, together, provide the desired global functionality to the system.

6. IMPLEMENTATION

YANCEES is composed of four parts, the extensible framework itself, a set of publish subscribe dispatchers, a set of basic plug-ins, services and filters, and the configuration files, that specify the different compositions using those components, in order to build notification servers to attend specific requirements.

The framework was implemented in Java 1.4 and the Java API for XML Processing (JAXP) v1.2.3, which supports XMLSchema [21]. The XMLSchema provides inheritance and extensibility mechanisms used to define the subscription, notification and protocol languages. In our current prototype the communication between clients (publishers and subscribers) and the YANCEES service is performed by using Java RMI.

The framework provides a generalized implementation of the publish/subscribe model, externalizing methods such as PUBLISH, SUBSCRIBE and UNSBSCRIBE to the end user. These methods operate over generic types that encapsulate XML messages representing subscriptions, notifications and events. Protocols are handled by the special SENDMESSAGE method in the YANCEES API. Internally, YANCEES implements the parsing and dynamically allocation of plug-ins in response to those methods, and manages the active protocols and subscriptions. From the point of view of the programmer, YANCEES provides standardized interfaces and abstract objects that represent the main extension points such as: plug-ins, filters and services. It also provides access to services and filters from an internal API, as a way to support the implementation of plug-ins. The publication and subscription APIs are multithreaded and can handle thousands of events and subscriptions per second, as will be discussed in section 7. The programmatic interface has more functions than described here. However, space limitations prohibit us from discussing them all.

The event dispatcher components used in the implementation were Siena 1.5.0, Elvin 4.x and an in-house developed topic-based event switcher. Both Siena and Elvin support content-based subscriptions and federation of servers, and both represent events as attribute/value pairs.

The configuration files are XML documents that specify which components to install in a given configuration, which include a set of dispatchers, plug-ins, services and filters. Internally, a configuration file is interpreted by the configuration manager that bootstraps the service with the required components.

We have implemented subscription plug-ins that provide event correlation (event sequence detection) (257 LOC) as well as topic-based event routing (268 LOC) and content-based event filtering using integrating Elvin and Siena (3000 LOC). Plug-ins that support push and pull notification policies are also available (50 LOC). Protocol plug-ins that support the polling of events are also implemented (200 LOC). Services as event persistency and CASSIUS-like event hierarchy are provided (1100 LOC), as well as loggers and type checking filters. More plug-ins are being developed.

7. PERFORMANCE TESTS

One important question to be answered when using the extensible framework is to determine the computational overhead of that approach with respect to existing systems.

In this section, we compare the performance of two existing systems with different YANCEES configurations: (1) YANCEES using Siena as its event dispatcher component, communicating via sockets; (2) YANCEES using a C implementation of Elvin as its dispatcher, also communicating via sockets; (3) YANCEES using Siena as a local (same process) dispatcher component, with no extra integration cost; and (4) YANCEES using a simple topic-based dispatcher. The system was compared with respect to its event and subscription throughputs, average subscription delay, and the delays between the publication and delivery of events. The results are presented in Table 2. All tests were preformed in a 3GHz Pentium IV with 1GB RAM, running over J2SDK1.4.2 from Sun.

Table 2 Comparative benchmarks using YANCEES, Elvin and Siena. x = XML events; n = native events

	Throughput (events/ second)	Throughput (subscrip./ Second)	Average subscirp. delay (ms)	Average pub./notif. delay (ms)
Elvin	5000	25	39	8
Siena	70	10	1	3
YANCEES over Siena remotely	454(x) 515(n)	142	4	217(x) 199(n)
YANCEES over Elvin remotely	500(x) 1500(n)	142	5	203(x) 141(n)
YANCEES over local Siena	3000(x) 4000(n)	166	4	70(x) 46(n)
YANCEES over fast switcher	2000(x) 5000(n)	142	4	137(x) 18(n)

In order to test the system throughput, a large number of evens (100,000) are published to the notification service. A consumer is provided that counts the number of events arrived per second. The goal of this test is to identify the maximum number of events per second the infrastructure can route1. In another test, the ability of processing subscriptions is measured. The goal of this test is to determine the average delay associated to the parsing of a subscription. Finally, the average delay between the publication and arrival of the events was also determined. The YANCEES configuration tests were performed using both XML events (marked with x in the table) and a native (Java) event representation (marked with n in the table). The advantage of the native representation is speed, since the XML parsing is not necessary. The results are presented in Table 2.

Since YANCEES provides multithreaded subscription handling, the average delay to post a subscription is minimized as well as the XML parsing latency associated with it. In YANCEES, the publication and notification of events is bufferized, which increased throughput (3000 to 4000 evens per second using simple content-based filtering). If topic-based is used, and the proper dispatcher installed, this number can raise up to 5000, which is comparable to Elvin throughput (a fast C implementation). This approach, however, increases the system latency. An initial latency (around 46-70ms) occurs between an event publication and its notification. This latency is explained by the XML parsing of the events and the internal processing of the plug-ins and filters through the subscription trees. Additionally, when Siena or Elvin are remotely used, this value increases due to inter-process communication delays (sockets). This is the price to be paid for the generality of the model, which is compensated, however, by its high throughput.

8. CASE STUDIES

In this section we present two evaluative case studies. We use the first case study for two purposes in this paper: 1) to show an evaluation of YANCEES in the context of supporting the application domain of groupware/awareness; and 2) to show the specific methods a user of YANCEES typically would employ in implementation. In the second case study, we present our experience using YANCESS to support a security visualization application. In the context of that application, we show how YANCEES achieves the properties of versatility described in section 3; however, we omit the lowest-level details of the implementation in this second case.

[1] Note that Siena alone was not able to handle this load in both subscription and publication of events but could handle it when used locally or remotely through YANCEES, an implementation problem of that prototype

8.1 Evaluative Case Study 1: Extending and configuring YANCEES to support awareness applications

From the programmer perspective the steps necessary to implement a new extension using the YANCEES framework are the following:

1. Extend the subscription, notification, or protocol model languages, as necessary, by using the extensibility facility provided by the XMLSchema language;
2. Implement the plug-ins that will be invoked to provide the functionality of the new commands in those languages;
3. Implement auxiliary filters and services, if necessary;
4. Update the respective configuration file to include the new components.
5. Restart the server (or clients) using the new configuration, or dynamically reconfigure them.

In some cases, the event model or the event dispatcher may also need to be changed. In such cases, besides changing the event language, the user may need to provide a new event dispatcher adapter.

This section shows how to adapt the YANCEES framework, from a programmer perspective, in order to address the needs of specific application domains. In this case study, we extend the framework to provide the main services required by awareness applications. The requirements of this domain are based on the set of functionalities provided by notification servers such as Khronika [7] and CASSIUS [8]. Hence, in order to be functionality-compatible with this domain, YANCEES needs to provide: event persistency and typing, event sequence detection, and the pull notification delivery mechanism. Moreover, a special feature provided by CASSIUS is the ability to browse and later subscribe to the event types that are published by each event source. This feature, called event source browsing, provides information about the publishers and their events. In the following sections we describe how each one of these features is implemented in the YANCEES framework.

8.1.1 Sequence detection

Event sequence detection requires the extension of the subscription language with a new keyword: <*sequence*>. It operates over a set of content-based plug-in filters (which are assumed to be already available), that interact with a content-based dispatcher, already installed in the YANCEES framework.

The first step is to extend the YANCESS subscription language with the new <*sequence*> tag, using the regular extension mechanisms of XML Schema [21]. The extension defines the syntactic relationship between this new tag and the existing ones in the subscription language, in order to represent subscriptions in the new format described in Table 6.

The next step is to implement the new functionality as a plug-in. This is usually accomplished by extending an implementing the *AbstractPlugin*, a convenience class that provides default implementations to the plug-in interface presented in Table 3.

Table 3 Plug-in interface

```
interface PluginInterface
      extends PluginListenerInterface {
  long getId();
  String getTag();
  String getFullContext();
  String getFullPath();
  Node getSubtree();
  void addListener (PluginListenerInterface
plugin);
  void removeListener
(PluginListenerInterface plugin);
  void addRequiredPlugin (PluginInterface
plugin);
  PluginInterface[]
getRequiredPluginsList();
  boolean hasChildren();
  void dispose(); }
```

From the programmer point of view, the interaction with the framework mainly takes place through the implementation of the *PluginInterface*. Note that this interface is also a listener to events produced in other plug-ins (it implements the *PluginListenerInterface* described in Table 4). This is the key feature used to implement our sequence detection plug-in since it allows the runtime composition plug-ins, in our case filter plug-ins. Hence, the sequence detection plug-in will operate over events published by the filters specified in its subscription. Filters are plug-ins already provided in the YANCEES framework, which allows the programmer to concentrate in the specific task of detecting the event sequence. The *PluginListenerInterface* provides two methods to collect events from other plug-ins as described in Table 4 as follows.

Table 4 Plug-in listener Interface

```
interface PluginListenerInterface {
  void receivePluginNotification
(EventInterface evt, PluginInterface
source);
  void receivePluginNotification
(EventInterface[] evtList, PluginInterface
source); }
```

A simple sequence detection implementation will collect, in the right order, events or patterns (event arrays) coming from each one of the filters it listens to. When a successful sequence is detected, the sequence plug-in returns the set of events of the pattern, publishing it to possible higher-level plug-ins (listeners). Note that we are assuming that the event dispatcher guarantees the in-order delivery of events. If this is not the case, more complex algorithms must be used.

In order to integrate the plug-in with YANCEES, a plug-in factory, implementing the interface presented in Table 5 must also be defined.

Table 5 Plug-in factory interface

```
interface PluginFactoryInterface {
  String[] getTags();
  PluginInterface createNewInstance (Node
subTree); }
```

A simple factory implementation will return a new instance of the plug-in each time the *createNewInstance*() method is invoked in its interface. The plug-in factory must then be registered under the "*sequence*" tag name in the YANCEES configuration file. The use of plug-in factories is important. It intermediates the creation of new plug-ins, coping with the runtime change ability of the system, allowing the transparent change of plug-in implementations.

After these steps, and registering the plug-in in the YANCEES framework, whether dynamically or statically in the configuration file, the plug-in is ready to be used. It will be activated each time a subscription is provided that uses the <*sequence*> tag in its body. It will produce plug-in evaluation trees as the one presented in Figure 3. An example of a subscription using this new extension is presented in Table 6.

Table 6 Example subscription using sequence detection and pull notification

```
<subscription>
  <sequence>
    <filter> <EQ>
        <name> status</name>
        <value> Fail </value>
      </EQ> </filter>
    <filter> <LT>
        <name> cooler Temp </name>
        <value> 90 </value>
      </LT> </filter>
  </sequence>
</subscription>
<notification>
  <pull/>
</notification>
```

8.1.2 Pull delivery mechanism

The Pull notification delivery policy requires subscribers to periodically poll (or inquire) the server for new events matching their subscriptions. For such, messages need to be stored in the server side. This mechanism copes with the requirements of some mobile applications, when subscribers can get temporarily disconnected for some periods of time.

In the YANCEES architecture, this mechanism is provided by a combination of notification and protocol model extensions. First, in the notification model, the <*pull*> language extension is defined. This policy activates a plug-in that, instead of sending the notifications directly to the subscribers, store the events in a temporary repository, implemented as a service component

In addition to the pull notification plug-in and the event persistency service, users need a way to periodically collect the events stored in the server. This functionality is provided by a polling protocol plug-in. It responds to subscriber's commands such as <*poll-interval*>, <*stop-polling*> and <*poll*>, which define different polling mechanisms. Responding to commands from a subscriber, the poll protocol plug-in gets the events stored in the persistency service, and delivers them to the subscriber in two different ways: periodically (poll-interval command) or over request (using the simple poll command). The periodic delivery can be deactivated (using the stop-polling command) in response to a temporary disconnection.

8.1.3 Special features

In addition to the features described in the previous sessions, CASSIUS provides event typing and the ability to browse through hierarchies of event sources.

The browsing of event sources in CASSIUS allows publishers to register events in a hierarchy based on accounts and objects. This model and the API required to operate the server are described elsewhere [8]. In the YANCEES framework, the CASSIUS functionality is implemented by the use of filters, a protocol plug-in and a *CassiusService* component.

The CASSIUS protocol plug-in interacts with the *CassiusService*, which allows the creation and management of objects, accounts, and their events. These operations include registering/un-registering accounts, objects, and events, as well as polling commands.

CASSIUS uses events with a fixed set of attributes, as opposed to the variable set of attributes an event can usually support in YANCEES (which depends on the dispatcher component used, in this case Siena). Hence, this restriction can be enforced by filters in the framework. CASSIUS events are marked with a special attribute indicating its type. Whenever the input filter detects this special attribute, the event attribute names and types are checked for correctness, according to the CASSIUS template format. This filter also helps in turning the events persistent. Once a CASSIUS event is identified and validated, it is sent by the filter to the *CassiusService*, which stores it in a database in its proper account/object record.

Polling of events, in this case, is handled by the CASSIUS protocol plug-in, which allows the collection of events by account, object, or sub-hierarchies. Note that this approach does not prevent the simultaneous installation of both services: the simple pull and the CASSIUS pull protocol.

8.2 Evaluative Case Study 2: Extending and configuring YANCEES to support a security visualization application

In this section, we present some of our experience in using YANCEES to achieve most of the requirements presented in section 3. (We are still researching appropriate criteria for usability). In this case study, we demonstrate YANCEES extensibility, configurability, interoperability and reusability in an application monitoring scenario as presented in Figure 4.

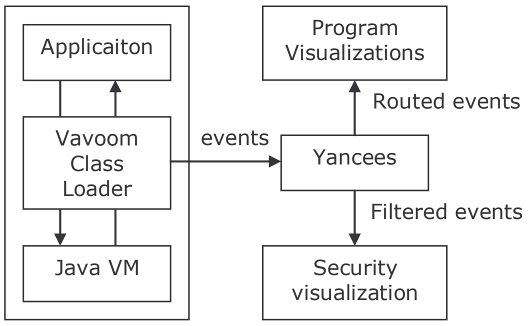

Figure 4 Application monitoring scenario

In this scenario, an enhanced version of Vavoom software visualization tool [22] was extended to use YANCEES as the basic communication infrastructure. Vavoom is a visual java virtual machine that allows the monitoring and visualization of java programs executions. The visualizations are based on program execution events such as object creation, method invocations, variable changes and so on. Vavoom was extended to provide a security visualization tool that displays information about the network activity of monitored applications.

In this application, YANCEES plays two roles: In one hand, it is used to route events to program visualizations that display the current execution of the application. This requires fast routing of events and no content-based filtering; On the other hand, it also needs to provide content-based event filtering and sequence detection capabilities for a security visualization, that displays and monitors the activity of current socket connections the application may open. The Vavoom class loader produces Java opcode-level events with information, including object creation and destruction, method invocation and exceptions. This information, even though sufficient for Vavoom program visualizations, is insufficient for the security visualization tool. The visualization uses network-related information such as method invocations, with certain parameters, in a certain order. Hence these low-level events, produced by the class loader, must be filtered, combined and generalized to provide higher-level network activity information to the security visualization.

Extensibility. The scenario demanded two different qualities of service from YANCEES: fast switching of events to be used by the real-time visualizations and content-based filtering with sequence detection, to be used by the security visualization. This former requirement is fulfilled in YANCEES by the use of a topic-based dispatcher developed for this scenario (268 LOC). The second requirement was provided by correlation language extensions, with their respective plug-ins (257 LOC), which provided different sorts of event sequence detection that operated over timing and event order constraints. The implementation of the correlation plug-ins was facilitated by the reuse of filtering plug-ins and their language extensions that were already implemented in the system at that time. This allowed the implementation to focus on detecting sequences, and not on filtering.

Configurability. For this specific software monitoring scenario, both dispatchers (content-based and topic-based), with their respective plug-ins were installed together in the same configuration. They operate at the same time, being dynamically selected based on the subscription originated from both kinds of visualizations: the security visualization and the application visualization.

Interoperability. In the initial state of our prototype, for performance reasons, Elvin was used as the event dispatcher component of YANCEES. Later on, with the implementation of the fast switch router dispatcher component, Elvin could be completely replaced. This was accomplished with no further change in the visualizations. This demonstrates the ability of YANCEES to integrate different event notification services. In other words, from the interoperability point of view, YANCEES provides an abstraction layer that operates over different notification servers, which are wrapped as event dispatcher components in the YANCEES framework. This indirection layer makes the change in the underlying infrastructure to be transparent to the client applications. Moreover, using this approach, allows events published using Elvin native API to be visible to YANCEES and vice versa.

Reusability. This scenario also illustrates the reuse of current solutions: for example, the reuse of content-based filtering capabilities (plug-ins and subscription language extensions) already provided by YANCEES; and the reuse of existing content-based routers, as Elvin and Siena.

9. RELATED WORK

The general idea of extensible and configurable software architectures is not new and has been proved successful in a wide variety of domains. Their use in publish/subscribe services, however, has been ignored. This section presents some of the previous work in the area of distributed systems and middleware that, at some extent, inspired our approach.

The Click Modular Router [23] defines a basic architecture for the definition of flexible and modular Internet routers. In this architecture, software modules can be arranged according to an IP routing workflow, allowing the expression of different policies and configurations that coordinate the proper routing of IP packages. Promile [24] is another system that extends the Click Modular Router configurable architecture, adding to it the runtime change capability. It uses a graph (workflow), described in XML, to specify the interconnection between modules. Modules work as filters and policy enforcers that are inserted in the main event stream of the router in a pipe and filter architecture style. A special process called the graph manager controls the dynamic change (insertion and removal) of these components in the package flow of the router.

Even though the problem of routing Internet packages does not provide the full content-based filtering of a publish/subscribe model, it provides good insight on how to provide dynamic change by using a modular architecture, as well as the service priority arrangement provided by the workflow model. Filters in our model follow a similar approach.

Software architectures and event-based systems can be combined to provide a framework to support runtime configuration. Oreizy and Taylor [25], for example, propose the use of the C2 architectural style to support these changes. Likewise, event processing languages (such as GEM) and dynamic architecture languages (such as Darwin) can be used to implement configurable distributed systems at the application level [26].

Software architecture and architecture definition languages inspired the configuration manager of the YANCEES framework. The runtime composition of plug-ins with reuse of functionality through abstraction mechanisms was inspired in the composition of rules provided by GEM.

Configurable middleware services have been described in the literature. For example, TAO [27] allows the static configuration of services or the runtime change of strategic components in a CORBA ORB. TAO can be configured to cope with different real-time constraints by selecting the appropriate implementation of each component of the ORB. It also allows the definition of configurations where unnecessary components are not present, which addresses small footprint requirements of mobile devices or special real-time constraints. The motivation of this work is similar to ours: the need to cope with different requirements of specific application domains. In the case of TAO, real-time is the main application domain.

The Apache web server is another example of configurable middleware. It uses a pluggable architecture where modules providing different services can be added. These modules can be installed in distinct phases of the request handling, processing and response sending process. This approach has some similarities to the plug-ins in our notification server. However, differently from the pluggable publish/subscribe component in our architecture, Apache uses a very simple extension model, with no parallelism and distribution flexibility: each request and response follows the same workflow. It also does not allow the addition of new services at runtime.

In another approach [28], computational reflection is used to design an a configurable and open middleware implementation. This approach is applied to CORBA ORBs as a way to intercept, modify and adapt calls to provide the desired functionality.

10. CONCLUSIONS AND FUTURE WORK

In this paper, we presented YANCEES, a framework and implementation of event notification services that strive for versatility. In sum, YANCEES provides an extensible, configurable, and dynamic framework that allows the implementation and customized of event notification services to support different application requirements. This approach permits the provision of the right notification service to the appropriate application domain.

Theoretically, we show that publish/subscribe systems can be decomposed in sets of components that follow an extended version of the Rosenblum and Wolf [14] design framework. This extended version supports a protocol dimension that captures interactions with the publish/subscribe system other than the common publication and subscription of events. Practically, we show how these dimensions can be extended by the dynamic combination of plug-ins, filters and services by describing two use cases where YANCEES was used to provide application-specific services. These two contributions combined provide a novel approach to cope with the versatility demanded by different applications.

The use of a single extensible framework has many benefits. It provides a common event model on top of which extensions can be built. This model increases the interoperability of event notification systems, opening the possibility for the integration of different notification servers under a common infrastructure. Moreover, it improves the reuse of the components of the framework, which speeds up the development of new solutions.

These benefits come through a price. Performance tests with a prototype of the framework showed an increase in the routing latency of the events. This delay in performance, however, is compensated by the gain in configurability and extensibility, and is amortized by the high-throughput and multithread support provided in the current prototype implementation.

The use of XML as subscription and event representation provide extensibility to those dimensions. However, form the point of view of the end user, the interaction with the system through the use of XML is cumbersome. Hence, improvements in the user interface with the system are necessary and will be provided in future versions of our prototype. In particular, we are exploring the use of graphical subscription editors and automatic configuration generators to facilitate the configuration and use of the system.

The implementation of crosscutting functionality such as security is still not completely addressed in the framework. The YANCEES platform, however, provides a testbed where the implementation of such non-functional requirements can be tested and further exploited.

11. ACKNOWLEDGMENTS

The authors thank CAPES (grant BEX 1312/99-5) for financial support. Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel

Command, USAF, under agreement number F30602-00-2-0599. Funding also provided by the National Science Foundation under grant numbers CCR-0205724 and 9624846. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Laboratory, or the U.S. Government.

12. REFERENCES

[1] D. Hilbert and D. Redmiles, "An Approach to Large-scale Collection of Application Usage Data over the Internet," presented at 20th International Conference on Software Engineering (ICSE '98), Kyoto, Japan, 1998.

[2] P. Dourish and S. Bly, "Portholes: Supporting Distributed Awareness in a Collaborative Work Group," presented at ACM Conference on Human Factors in Computing Systems (CHI '92), Monterey, California, USA, 1992.

[3] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: Raising Awareness among Configuration Management Workspaces," presented at Twenty-fifth International Conference on Software Engineering, Portland, Oregon, 2003.

[4] G. Cugola, E. D. Nitto, and A. Fuggetta, "The Jedi Event-Based Infrastructure and Its Application on the Development of the OPSS WFMS," *IEEE Transactions on Software Engineering*, vol. 27, pp. 827-849, 2001.

[5] OMG, "Notification Service Specification v1.0.1," Object Management Group, 2002.

[6] R. E. Gruber, B. Krishnamurthy, and E. Panagos, "The Architecture of the READY Event Notification Service," presented at ICDCS Workshop on Electronic Commerce and Web-Based Applications, Austin, TX, USA, 1999.

[7] L. Lövstrand, "Being Selectively Aware with the Khronika System," presented at European Conference on Computer Supported Cooperative Work (ECSCW '91), Amsterdam, The Netherlands, 1991.

[8] M. Kantor and D. Redmiles, "Creating an Infrastructure for Ubiquitous Awareness," presented at Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001), Tokyo, Japan, 2001.

[9] B. Krishnamurthy and D. S. Rosenblum, "Yeast: A General Purpose Event-Action System," *IEEE Transactions on Software Engineering*, vol. 21, pp. 845-857, 1995.

[10] M. Mansouri-Samani and M. Sloman, "GEM: A Generalised Event Monitoring Language for Distributed Systems," presented at IFIP/IEEE International Conference on Distributed Platforms (ICODP/ICDP'97), Toronto, Canada, 1997.

[11] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, pp. 332-383, 2001.

[12] G. Fitzpatrick, T. Mansfield, D. Arnold, T. Phelps, B. Segall, and S. Kaplan, "Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin," presented at European Conference on Computer Supported Cooperative Work (ECSCW '99), Copenhagen, Denmark, 1999.

[13] SUN, "Java Message Service API," vol. 2003: SUN, 2003.

[14] D. S. Rosenblum and A. L. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," presented at 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, 1997.

[15] OMG, "CORBA Event Service Specification (version 1.1)," OMG, 2001.

[16] Talarian, "Talarian: Everything You Need To Know About Middleware," vol. 2003: Talarian, 2003.

[17] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Challenges for Distributed Event Services: Scalability vs. Expressiveness," presented at ICSE '99 Workshop on Engineering Distributed Objects (EDO '99), Los Angeles, CA, USA, 1999.

[18] J. Nielsen, "What is Usability?," in *Usability Engineering (Chapter 2)*, J. Nielsen, Ed.: Morgan Kaufman, 1993, pp. 23-48.

[19] R. S. Silva-Filho, C. R. B. deSouza, and D. F. Redmiles, " The Design of a Configurable, Extensible and Dynamic Notification Service," presented at Second International Workshop on Distributed Event-Based Systems (DEBS'03), San Diego, CA, USA, 2003.
[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Publishing Company, 1995.
[21] W3C, " XML Schema Part 0: Primer. W3C Recommendation," 2001.
[22] P. Dourish and J. Byttner, "A Visual Virtual Machine for Java Programs: Exploration and Early Experiences," presented at ICDMS Workshop on Visual Computing, Redwood City, CA, 2002.
[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, pp. 263-297, 2000.
[24] M. Rio, N. Pezzi, H. D. Meer, W. Emmerich, L. Zanolin, and C. Mascolo, "Promile: A Management Architecture for Programmable Modular Routers," presented at Open Signaling and Service Conference (OpenSIG 2001), London, UK, 2001.
[25] P. Oreizy and R. N. Taylor, "On the Role of Software Architectures in Runtime System Reconfiguration," *IEE Proceedings - Software Engineering*, vol. 145, pp. 137-145, 1998.
[26] M. Mansouri-Samani and M. Sloman, "A Configurable Event Service for Distributed Systems," presented at Proc. Configurable Distributed Systems (ICCDS'96), Annapolis, MD, USA, 1996.
[27] D. C. Schmidt and C. Cleeland, "Applying a Pattern Language to Develop Extensible ORB Middleware," in *Design Patterns and Communications*, L. Rising, Ed.: Cambridge University Press, 2000.
[28] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An Architecture for Next Generation Middleware," presented at Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Lake District, UK, 1998.