Providing Extensibility and Configurability to Event Notification Services

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

[1]*School of Information and Computer Science*
University of California, Irvine
Irvine, CA, USA

[2]*Informatics Department*
Federal University of Pará
Belém, PA, Brazil

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

Abstract

Publish/subscribe services, specifically notification servers, are used as the middleware for the implementation of a large set of collaborative software engineering tools. With their popularization, many notification servers are being developed to support specific application domains. At the same time, general-purpose notification servers provide a large set of functionality for a broad set of applications. As a result, developers face the dilemma of choosing between application-specific or general-purpose notification servers. In both cases, however, the set of features provided by the servers are neither extensible nor configurable, hindering the use of these infrastructures in different application domains, making it difficult to add support for new functionality. In this work, we describe a more flexible approach – a customizable, extensible and dynamic architecture for notification services, which allows the configuration of the notification service to different application domains. The design and implementation of the system are discussed, as well as configuration examples.

Keywords: *Notification servers, event-based middleware, dynamic architecture, pluggable architecture.*

1. Introduction

Collaborative software development is an activity that relies on a broad set of tools, many of them groupware applications, designed to support virtually every aspect of a software development process. Examples of such applications include instant messaging, configuration management databases, e-mail, software monitoring, shared workspaces and many awareness tools. Many of these applications rely on services provided by event notification infrastructures. Indeed, several popular applications such as instant messenger systems, groupware applications [1], software monitoring [2], application gauges [3], CM awareness tools [4], and workflow management systems [5] have been built using some sort of event notification infrastructure. This infrastructure is often implemented in the form of notification servers that are responsible for the routing and runtime query of events.

Due to its increased popularity, event notification services need to cope with new requirements, coming from different application domains. In fact, a broad spectrum of research and commercial tools are available nowadays. At one extreme, "one-size-fits-all" approaches, such as adopted by CORBA Notification Service [6] or READY [7] strives to address new applications requirements by providing a very comprehensive set of features, able to support a broad set of applications. At the other extreme, specialized notification servers tailored to application-specific requirements provide novel but specific functionalities. Examples of such specialized systems include Khronica [8] and CASSIUS [9] which are specially designed to support groupware and awareness applications; or even Yeast [10] and GEM [11] which are specialized in advanced event processing for local networks applications, and distributed applications monitoring respectively. Finally, servers such as Siena [12] and Elvin [1], even though designed with special domains in mind, strive for a balance between specificity and expressiveness of the subscription language and event model they support.

Therefore, in the development of event-based collaborative software applications, developers face the dilemma of specialization versus generalization: to use a generalized infrastructure, that can support and integrate different applications, but may not provide all the necessary functionality for specific application domains; or to use one event-based infrastructure for each application domain, having "the right tool for the right problem", but loosing the uniformity and integration of a single solution. For example, in the development of awareness tools, one can use either a comprehensive solution such as CORBA Notification Service (CORBA-NS) that provides a large set of features and services, or a domain specific solution such as CASSIUS, that provides specific services for awareness applications. That's important to note that services as CORBA-NS, even though are very comprehensive in its set of features, do not provide direct support for all the requirements of awareness applications. It does not allow event source hierarchy and browsing, for example. CASSIUS, on the other hand, provides this feature and allows the easy discovery of information sources and the subscription for events from different source components.

If CORBA-NS were used in an awareness application demanding this feature, this service would need to be implemented by the application itself.

Another problem of the currently available event-based infrastructures is the weak support for selection and customization of the services to be provided, which is important for applications that run on resource-limited devices such as the ones common to mobile applications. If a notification service as CORBA-NS is used for such applications, the whole set of features provided by this server would be available for use, whether the application requires it or not.

Moreover, current event-based infrastructures lack mechanisms to support the easy extensibility of its functionality. The only extension mechanism is usually the (understanding and) change of their source code, or the implementation of the service by the client. Additionally, due to restrictions in their event or subscription models, the addition of new functionality may constitute a very difficult task [3].

This work was motivated by the problems discussed above, faced when adapting current notification servers to different collaborative software development scenarios. In an organization, there is a clear need of a single server model that can be adapted and customized to different applications. Moreover, due to the constant evolution of the applications and tools used in these environments, the event-based infrastructure must provide ways to add new features, when necessary, to evolve with the organization's needs.

In this work, we describe the design and implementation of an extensible and pluggable architecture for notification services that provides an alternative to the generalized versus specialized dilemma. It provides extensible event, notification, subscription and resource models (see [13] for a detailed description of these models), which at the same time supports the specialization, extension, and customization of the event notification service toward the needs of different applications. In order to provide this flexibility, we use the following main design strategies: a pluggable publish/subscribe core; the combination of this core with plug-ins; the ability to add consumer and producer side plug-ins and services, with the ability to customize their location; the adoption of an extensible subscription language, and the use of an extensible event model. These strategies provide a basic architecture that can be extended to support new functionality demanded by (new or) different application domains. It is also dynamic, allowing the addition and removal of services at runtime.

2. Publish/subscribe systems

A publish/subscribe service mediates the communication between publishers and consumers of information in distributed systems. Applications using this service are named event-based systems. In these systems, some components (or information sources) announce (or broadcast) events, while other components (information consumers) register interest in these events. An event expresses a state change in a (possibly distributed) component, or represents some temporal fact in the world. Computationally, events are usually represented as sets of attribute/value pairs, programming language structures (records), or even objects.

Information consumers express their interest in subsets of events by submitting predicates defined in terms of event content, type and correlations. These predicates are called subscriptions. A publish/subscribe service is responsible for receiving: (1) subscriptions, coming from event consumers, and (2) events, coming from their producers. With these pieces of information, it efficiently performs the matching of subscriptions with their corresponding subset of events, routing the results, as notifications, to the interested parties.

2.1. The evolution of publish/subscribe systems

Publish/subscribe systems evolved over time, incorporating new functionality and interaction models. First generation publish/subscribe 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 [14]. Subject-based systems are more flexible: each event is annotated with a special field, usually a string or token, called subject (or topic) to describe its content. Information consumers specify their subscriptions based on this specific field. An example of such systems is the Talarian SmartSockets [15].

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 [16]. They usually have to process, route and combine events coming from different sources. Examples of such systems include Siena [12], Jedi [5] and Elvin [1].

2.2. Message-oriented middleware (MOM)

MOMs provide a basic set of mechanisms and services for the construction of event-based distributed applications. Examples of such systems include the Microsoft® Message Queuing (MSMQ) [17], SonicMQ [18] and IBM MQSeries [19]; as well as the JMS (Java Message Service API) [20] specification from SUN Microsystems. These systems implement a basic message queuing mechanism providing a synchronous or asynchronous message-based communication channel, which allows the exchange of information between information producers and consumers. This message service is usually augmented with functionality and characteristics such as message persistency, transactions, cryptography and secure channels, load balancing, scalability support, XML messages; guaranteed message delivery and others.

MOMs provide typically asynchronous and peer-to-peer interaction channels between processes. A central service mediates the creation of a communication channel or queue between processes before they can start exchanging messages. Instead of providing content-based routing, as the most recent event-notification servers such as Siena, MOMs usually provide channel-based or subject-based addressing mechanisms. This approach copes with the scalability requirements of the application domains that use this technology, which favor efficient routing algorithms instead of expressive event processing capability.

2.3. General-purpose notification servers

General purpose notification servers such as READY [7], and the CORBA Notification Service [6] from OMG are usually designed to satisfy a large number of requirements that span different application domains, providing a broad spectrum of functionalities. The CORBA notification service, for example, is an extension to the CORBA event service that allows the definition and management of different event channels between CORBA distributed objects. Events can be typed or untyped, persistent or non-persistent. Subscriptions allow sequence detection and content-based filtering. The event delivery and observation can be performed using pull and push approaches. Secure channels can be established between publishers and subscribers. Scalability is addressed using federation of servers.

2.4. Optimized notification servers.

As an intermediate category of notification servers, Siena [12] and Elvin [1] provide a relatively simple but optimized set of functionalities, with the ability to efficiently process a large amount of events. Both of them allow the content-based subscriptions and a flexible event model based on event/value pairs. Elvin provides a quenching mechanism to speed-up the event routing, while Siena is designed to be scalable, maintaining its subscription expressiveness, including the ability to specify simple event patterns in its subscriptions.

2.5. Application specific servers

These servers are designed to support the requirements of specific application domains. For example Khronika [8] and CASSIUS [9], which are designed for supporting awareness; and Jedi [5] that supports mobile applications.

Jedi (Java Event-Based Distributed Infrastructure) was designed to cope with the special requirements of scalability and mobility. To cope with mobility, push and pull delivery policies are implemented. Scalability is achieved by server federation. Event ordering is guaranteed by the system. Roaming and special primitives for client migration are provided. Event sequence detection with regular expression is also supported.

A distinctive feature of CASSIUS is its ability to model information source hierarchies, allowing end-users to browse through event sources and hierarchies. The level of disruptiveness of the notifications can also be configured according to different awareness styles. It allows the definition of content-based or type-based subscriptions. Support for mobile applications is made possible through the use of the ubiquitous HTTP protocol and by allowing information consumers to store events in the server during periods of disconnection.

3. Analytical Design Framework

In order to approach the design of notification servers, Rosenblum and Wolf [13] propose a framework that captures the most relevant design dimensions (or models) of those systems. 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, 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 model.

In our design, we will consider a specialization of this framework which is close to the one provided by Cugola et al. at [5]. In special, in our model, the naming and observation models are combined together in what we describe as *subscription model*. In this case, the way resources are identified (naming) is part of the subscription language. A *protocol model* is defined in order to describe other kinds of interaction with the service other than the basic publish and subscribe messages, allowing the server to support new interaction protocols.

4. Design

In order to support the whole spectrum of requirements listed in the previous section, with the flexibility to select the subset of features needed by each application domain, our event notification architecture must be configurable and extensible. This section describes our approach to provide such flexibility. One of the main challenges in our design was to specify an infrastructure that allows the addition of new event processing functions (functional requirements) such as event filtering, sequence detection and abstraction, while allowing the incorporation of generalized (non-functional) aspects such as support for security and mobility.

The extensibility needs to embrace all the design dimensions presented in the design framework. The basic elements used in such extensibility are the representation of events, subscriptions and messages in an extensible language (XML); the use of smart parsers and the dynamic instantiation of plug-ins to attend the requirements of each subscription. This section will describe in more detail the extension mechanisms for each one of those models.

4.1. Architecture overview

The high-level architecture of the system is presented in Figure 1.

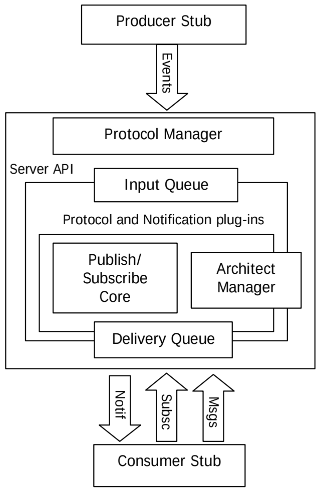

Figure 1: The high-level architecture of the extensible and configurable notification server.

In this architecture, a **publish/subscribe core** allows it's the extension of its notification, event and subscription models. One of the key elements to this extension is the use of plug-ins. The subscription query and notification mechanism are performed using a dynamic combination of plug-ins with a simple publish/subscribe core.

The architect manager orchestrates the addition and removal of plug-ins, and the allocation of components throughout the system, providing runtime change capabilities to the resource model. The protocol manager handles not only the publish/subscribe messages, but also every other communication, such as those related to the security (authentication) and mobility protocols. Each protocol is provided by a specific plug-in added to this component.

Non-functional requirements are addressed by the combination of those mechanisms in configurations that are instantiated by the architect manager. Each one of those approaches and main component of the architecture is described in more detail as follows.

4.2. The publish/subscribe core

The publish/subscribe core, described in Figure 2, is defined by a set of sub-components that make possible the handling of different subscriptions, using the available plug-ins.

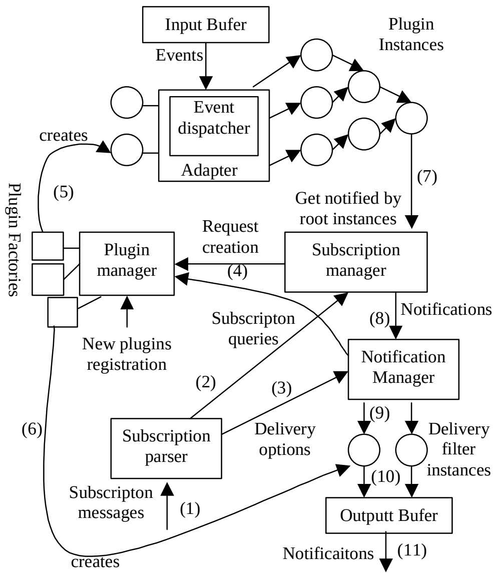

Figure 2 Architecture of the publish/subscribe core.

Subscription Parser. Subscriptions are defined as messages according to an extensible XML schema, which allows the definition of queries and event notification options. The subscription parser validates and extracts the subscription queries and delivery options from subscription messages, forwarding this information to the notification and subscription managers (1), (2) and (3).

Subscription Manager. The subscription manager component is responsible for handling the subscription queries provided by information consumers. It performs the interpretation of these requests assembling subscription trees that will execute the appropriate query using different plug-in instances, created with the help of the plug-in manager component (4) and (5).

For example, Figure 3 describes the subscription tree produced by the parsing of the following expression (note that, in this example, an algebraic form is used instead of XML for clarity and readability):

```
(A followed by B where
    A.CPUtemp > 150 or
    B.status == “non responding”)
OR (C followed by D)
Notify: Pull
```

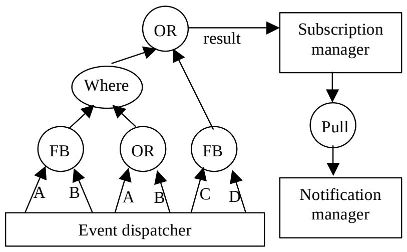

Figure 3 Decomposition of subscriptions by the subscription and notification managers using plug-ins.

As depicted in Figure 3, the subscription evaluation tree is composed of nodes, which are plug-in instances that communicate using the simple publish/subscribe design pattern [21]. In this tree, each level subscribes to its children nodes' results. A subscription like "`A.CPUtemp > 150 or B.status == 'non responding'`", for example, is evaluated by an OR plug-in instance that subscribes to events of type A and B. Once the tree is assembled and configured, the subscription manager subscribes itself to the results of each tree root plug-in (7). These results can be Boolean events, showing the occurrence or not of the subscription, a new event produced due to the evaluation of an expression, or a set of events that matched the specific filter.

This approach allows the subscription manager to perform optimizations such as the reuse of sub-expression parts between two or more subscriptions. For example, in an expression such as: "`(A followed by B or C)`" and another expression "`(D followed by B or C)`", the '`OR`' part of is common to both subscriptions. Since the evaluation uses the publish/subscribe model as described in Figure 3, subscription trees can be easily rearranged to share the same 'OR plug-in' output event.

Notification Manager. Similar to the subscription manager, the notification manager is responsible for parsing the notification options and allocating the appropriate plug-ins for the delivery of notification (4) and (6). For example, in Figure 3, pull notification is selected, which will be handled by the appropriate pull delivery plug-in.

Plug-in Manager. In order to assembly subscription and notification evaluation trees, both the subscription and notification managers use the plug-ins registered in the plug-in manager. Each plug-in is created by a registered factory [21], which is indexed by one or more operators (XML tags) that the plug-in is able to evaluate. For example, a '`<FOLLOWED BY>`' plug-in processes the ordered event sequences, being indexed by that special XML tag.

Event Dispatcher. A simple event dispatcher provides the basic event routing mechanism and dictates the event model of the notification server. If a content-based dispatcher is provided, and the event model is tuple-based, all the components will share this basic functionality and the event language will be built on this basic foundation.

In case an off-the-shelf event dispatcher is used, an adapter provides a standardized and simple publish/subscribe API that hides the idiosyncrasies of the selected component, including its event and subscription models.

Note that the event dispatcher adapter can also be used to extend the event model. For example, a tuple-based publish/subscribe core can provide types. Types in this case are special attributes in the tuple-based event, and should be enforced and managed by the adapter.

4.3. Protocol Manager

Non-functional services such as security and mobility usually demand other forms of interaction, other than the basic publish or subscribe commands. For example, security usually requires an authentication protocol and mobility is may require the extension of the server protocol with move-in/move-out commands [5]. The ability to support these and other additional services requires a mechanism to provide new interaction mechanisms to the server. The protocol manager allows the registration of plug-ins to handle these mechanisms. This component deals with aspects related to the protocol model defined in section 2.5.

4.4. Architect manager

The architect manager is the component responsible for the configuration of the diverse services and plug-ins in the architecture. It reads an XML specification with the topological arrangement of the components, and the list of available plug-ins, performing their proper installation.

Runtime changes in the plug-in factories and the components of the system are also managed by this component. The addition of new plug-ins consists on the registration of their factories in the plug-in manager. This process may require the interruption of the event flow. For such the input and delivery queues can be temporarily paused, while the new service is inserted in its appropriate place.

4.5. Extensibility

The extensibility of the architecture is achieved by the use of plug-ins, together with the configurability provided by the architect and plug-in managers, combined with the flexibility of the subscription language (query and notification languages) and the event model of the event dispatcher and its adapter.

4.5.1. The Event model

Events can be represented in many different ways. The most popular forms are tuples, records and objects [5]. The event model used by the system dictates the possible extensions the system will be able to have. For example, type checking is a feature available in record or object-based models, but not in tuple-based systems. The use of one of these models is dependent on the requirements the system will be customized to attend.

The event model is a native property of the event dispatcher and should be matched by a subscription language and a special plug-in to handle it. For example, if Siena [12] is used as the event dispatcher, the event model of the system will be tuple-based. The Siena subscription, which is content-based, will need to be described according to an XML schema. A special plug-in, registered with the plug-in manager to handle the tags in this language will also have to be provided.

4.5.2. The subscription model

In order to be able to handle different functionalities, the subscription language of the server is extensible. It is described using an XMLSchema that defines a set of basic tags that can be extended to provide new functionality. For example, support for out of order event detection can be expressed by the addition of a new element in the language: "`(A or-sequence B)`" which is mapped to a special tag which use is illustrated as follows:

```
<OR-SEQUENCE>
     <EVENT> A </EVENT>
     <EVENT> B </EVENT>
</OR-SEQUENCE>
```

The occurrence of this new tag in a subscription request, will instruct the subscription manager to look-up the '<OR-SEQUENCE>' tag in the plug-in manager. The correspondent plug-in for this tag is instantiated and the whole expression between <OR-SEQUENCE> and </OR-SEQUENCE> is passed to this plug-in. Plug-ins may need results from other plug-ins, which issues in the subscription evaluation tree in Figure 3.

In case a tag is not registered, indicating that its corresponding plug-in factory is not installed, the plug-in manager redirects the request to a **generic plug-in** that can search the plug-in implementation repository, or the Web, for that specific factory implementation. If available, the service can be downloaded and installed at runtime, allowing the plug-in instance to compose an expression. In case of failure, the subscription process is not completed, and an error control event is produced.

An example of an event processing language and service, with a configuration focusing on the support for awareness application, is discussed in [22].

4.6. The notification model

Extensions to this model are defined in the same way as the subscription model. For each new XML Schema extension, with the use of new tags, a plug-in needs to be defined and registered with the plug-in manager to interpret that extension. The plug-ins are them combined in a publish/subscribe evaluation tree. Example of notification plug-ins includes push or pull delivery and event persistency.

4.7. The protocol model

A protocol is defined as a set of primitive messages defined in XML. For each protocol, a plug-in is defined to handle the set of messages (or tags) in the protocol messages. In an authentication extension for example, an authenticator plug-in will be responsible for the validation of the messages, the keeping of the authorized users database, and the denial or concession of access to each message in the system.

4.8. The resource model

Some application domains may require the execution of part of the subscription activities in the producer or consumer sides. This approach usually results in the distribution of processing and the reduction of messages exchanges with the central service, which is important in mobile or large-scale applications, for example. Our model copes with these requirements by allowing the partial execution of subscriptions in both the producer and consumer processes.

Consumer and producer stubs mediate the interaction of the application with the notification service. Publishers interact with their stubs in order to send events, whereas consumers provide subscriptions and implement a listener interface defined in the consumer stubs.

4.8.1. Publisher and Consumer plug-ins and filters

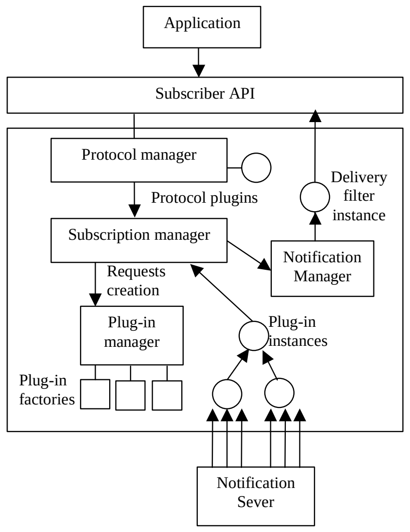

Figure 4 An example of the pluggable consumer stub configuration.

Figure 4 shows an example of a subscriber stub internal architecture that can be extended using plug-ins and notification filters. The use of a local subscription manager component allows the evaluation of subscriptions in the consumer address space. The purpose of this evaluation at the client-side permits, for example, that only well-formed subscriptions get to the server. This is helped by local instances of the plug-in manager and the subscription manager components.

Note that similar to the event dispatcher core, this consumer stub uses the simpler listener design pattern to assembly subscription evaluation trees locally. Simple events are received from the notification server. Functions as AND's, OR's or sequence detection, for example, are executed locally based on these events. Again, this is an *optional* feature that our architecture supports: the distribution of the event processing in the consumers or publishers. Of course, a developer might choose not to use this feature and let the server process the subscriptions.

Protocol plug-ins can be defined in order to implement client side security and mobility services. Another option is the local use of notification plug-ins. For example, producer-side persistency is useful in some mobile applications where events can be posted for delivery during disconnected operation, being sent when the connection to the notification server is reestablished.

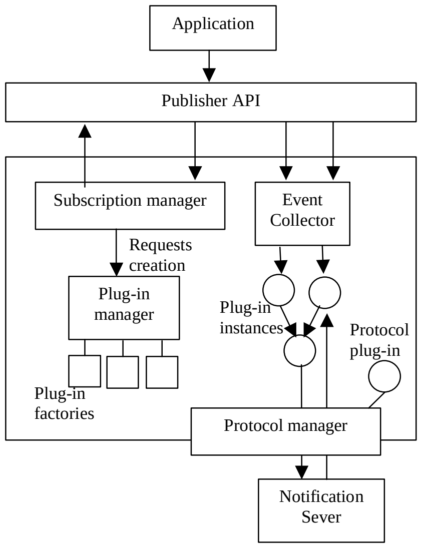

Figure 5 An example of a publisher stub configuration.

In another example, described in Figure 5, the publisher stub is extended with plug-ins used to collect event sequences and send higher-level events when the condition being evaluated is satisfied. In other words, these are composition or abstraction plug-ins. This approach requires download and execution of part of the subscription locally, those parts of the subscription evaluation tree that uses events produced in the current process. This does not limit the scope of these subscriptions, since events coming from other applications can be combined with local events, during expression evaluations, by allowing plug-in instances to subscribe to these events in the notification server. For example, for performance reasons, one can decide to perform the evaluation of the following subscription in the publisher side: "`(A and B and C and D and E)`", where A through D are locally produced events and E is produced by another application elsewhere.

The use of publisher plug-ins makes sense only when most of the events being processed come from the local host, as it is the case of event monitoring performed by systems like EDEM.

4.9. Extensibility summary

Table 1 presents the main design dimensions addressed in the system design, with their extension mechanism. Examples of features that can be added to the system using this approach are also presented.

4.10. Implementation

A prototype of the event-notification server architecture described in this work was implemented in Java (Sun J2SDK1.4). The server side components as including the plug-in infrastructure were implemented, as well as a simple version of the architect manager.

In our prototype, many simple content-based publish/subscribe systems could be used as the event dispatcher component in the publish/subscribe core. We are currently using Siena as this component for its simple event model, based on generalized attribute/value sets and its ability to perform content-based subscriptions.

Extensions to the basic Siena functionality as AND and OR event sequence detection and event rules were implemented using this framework. Other plug-ins are being developed to provide the functionality of CASSIUS and the EDEM event language in our architecture.

The extension of the core functionality with simple plug-ins is relatively easy. It requires the implementation of a component according to the plug-in interface and the extension of the subscription language to include the expressions (with their tags) that express the new functionality. The plug-in location is then informed to the system through the architect manager. Of course, more advanced features such as rule plug-ins, require a bigger amount of code to define the plug-ins, than a simpler sequence detector.

In our prototype, for the extensions performed, the overhead of the architecture was in the order of hundreds of milliseconds for each subscription received, which, for the sake of collaborative software tools, is not very expressive.

In parallel to the design and implementation of the system, we are also focusing in the subscription language to be used in our event-notification service. A language to address the specific problems of CSCW applications was proposed in [22]. A low-level representation of this language, using XML, is being developed.

Table 1 Adaptation points and features to extend according to the notification service model

Design dimension	How to Extend	Examples
Subscription (or event query) model	Extensible subscription language Provide feature specific event processing plug-ins	Event aggregation Abstraction Sequence detection
Event Model	Extensible event representation language Provide an event adapter for each dispatcher used Provide a plug-in to handle the dispatcher specific event language	Tuple based Record based (event typing) Object based (event typing)
Notification Model	Notification plug-ins (or filters) Extensible subscription language that allows the definition of notification policies	Push Pull (provides persistency)
Resource Model	Server configuration language and configuration manager that allows the distribution of event processing to server-side or client-side plug-ins	Centralized Partially distributed
Protocol Model	Extensible protocol language Protocol plug-ins to handle different protocols	Security protocols Mobility protocols Configuration protocols

5. Example configurations

In this section we present two examples of configurations of our architecture supporting representative application domains that use event-notification services, namely application and usability monitoring and awareness. These configurations were created to support features provided by notification servers used in these domains. A configuration of the architecture is a particular disposition of its components in order to attain a specific set of requirements or features. By presenting these configurations, we show that the elements of our architecture can be easily customized to implement application-specific notification servers.

5.1. Application and usability monitoring

In this application domain, software components and GUI design are monitored in order to detect design flaws and misassumptions with actual prototypes of the system being analyzed. In this context, an event-notification server needs to support the following subset of features: event sequence detection; event abstraction; content-based filtering; browsing of information sources and their events; and event persistency as a support for pull delivery of notifications. These features are provided by event monitoring applications as EDEM [2] and EBBA [23].

In this example, the publish/subscribe core of our architecture already provides content-based filtering. Sequence detection and event abstraction are implemented as plug-ins used by the subscription manager to extend the subscription model. In the EDEM approach, as discussed in section 4.8.1, the plug-ins are installed in the publisher stub; whereas in the EBBA approach, they are installed in the consumer stub. Our architecture supports both cases, since it allows plug-ins in both producer and consumer stubs. Furthermore, it supports the dynamic change from one approach to the other.

Event types are assured and implemented as an extension to the tuple-based model of the core. This is implemented in the event dispatcher adapter, and as an extension to the dispatcher language, provided by the event dispatcher plug-in. Persistency and pull delivery are plug-ins used by the notification manager. The browsing of information sources and their events is performed by plug-ins used by the protocol manager.

5.2. Awareness applications

Notification servers such as Khronika [8] and CASSIUS [9] are specially designed to support the development of awareness applications. These servers provide event persistency and typing, event validity (time-to-live), event sequence detection, and different notification delivery mechanisms. Moreover, a special feature in this case is the possibility to browse, and later subscribe to the event types that are published by each event source. This service, called event browsing, provides information about the publishers and their event types currently supported by the server.

This set of features is provided in our architecture by a set of plug-ins. Event typing is provided by the event dispatcher adapter which may extend the event model of the publish/subscribe core to provide this facility. Type checking is also provided by this component. Event browsing comprises the ability to advertise event sources types (like in Siena), organize this information in a database and provide this information to event consumers. This functionality is provided by protocol plug-ins. On the publisher side, another event browsing protocol plug-in is installed, forwarding queries about the event hierarchy to the server. Similarly to mobile applications, delivery policies and persistency are implemented as notification plug-ins. Content-based filtering is already addressed in the publish/subscribe core, but sequence detection is provided as a subscription query plug-in.

6. Related Work

The general idea of extensible and configurable software architectures is not new, being a research topic in different computer science fields. This section explores some of the previous work that inspired our approach.

6.1. Pluggable and programmable routers

The Click Modular Router [24] 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 [25] 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 provide good insights on how to provide dynamic change using a modular architecture.

6.2. Configurable distributed systems

Software architectures and event-based systems can be combined to provide a framework to support runtime configuration. Oreizy and Taylor [26] propose the use of the C2 architectural style to support these changes. Likewise, event processing languages (such as GEM) and dynamic architecture languages (as Darwin) can be used to implement configurable distributed systems in the application level [27].

Configuration managers and their use in extensible operating systems architectures is described in [28] and extensible middleware is described in [29]. Both require a way to describe the changes in the system with some kind of module interconnection language, and a service or automated tool to interpret and apply such description to the system. In this paper, we use an analogous approach to support different configurations of the event-notification service using fine-grained components.

6.3. Configurable middleware

Configurable middleware services have been described in the literature. For example, TAO [30] 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.

7. Conclusions And Future Work

In this work, we describe the design of a configurable, extensible and dynamic architecture for notification services, which provides the specificity necessary for the implementation of domain-specific applications and the generality necessary for supporting the requirements of a broad set of applications. The architecture provides (i) customization of the entire event notification service to meet the needs of different applications; (ii) extensibility to support the addition of new features; and (iii) dynamism allowing the introduction of these features at runtime. In order to provide these advantages, the architecture uses the following main strategies: a publish/subscribe core; the extension of this core with subscription, notification and protocol plug-ins; and the ability to add consumer and producer side plug-ins; the adoption of an extensible subscription, event, protocol and configuration languages based on XML.

Future work includes the improvement of the prototype, with the addition of client-side plug-ins; and the execution of tests using specific configurations, comparing their performance with available event-notification servers. Issues as timing and scalability still need to be explored in more detail in our design. The study of approaches to optimize the subscription processing, such as the use of the RITE algorithm and other approaches [31] are also part of our future plans.

8. Acknowledgements

The authors thank Eric Dashofy, Marcio Dias, Santoshi D. Basaveswara, and Max Slabyak for their contributions to the design of the system. We thank CAPES (grant BEX 1312/99-5) for the 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.

9. References

[1] 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," in European Conference on Computer Supported Cooperative Work, Copenhagen, Denmark, 1999.
[2] 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, Kyoto, Japan, 1998.
[3] C. R. B. de Souza, S. D. Basaveswara, and D. F. Redmiles, "Using Event Notification Servers to Support Application Awareness," presented at International Conference on Software Engineering and Applications, Cambridge, MA, 2002.
[4] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: Raising Awareness among Configuration Management Workspaces," presented at 25th International Conference on Software Engineering, Portland, Oregon, USA, 2003.
[5] G. Cugola, E. D. Nitto, and A. Fuggeta, "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.
[6] OMG, "Notification Service Specification v1.0.1," Object Management Group, 2002.
[7] R. E. Gruber, B. Krishnamurthy, and E. Panagos, "The Architecture of the READY Event Notification Service," in Proceedings of the 1999 ICDCS Workshop on Electronic Commerce and Web-Based Applications, Austin, TX, USA, 1999.
[8] L. Lövstrand, "Being Selectively Aware with the Khronika System," presented at European Conference on Computer Supported Cooperative Work (ECSCW '91), Amsterdam, The Netherlands, 1991.
[9] M. Kantor and D. Redmiles, "Creating an Infrastructure for Ubiquitous Awareness," in Eighth IFIP TC 13 Conference on HCI, Tokyo, Japan, 2001.
[10] B. Krishnamurthy and D. S. Rosenblum, "Yeast: A General Purpose Event-Action System," IEEE Transactions on Software Engineering, vol. 21, pp. 845-857, 1995.
[11] M. Mansouri-Samani and M. Sloman, "GEM: A Generalised Event Monitoring Language for Distributed Systems," presented at IFIP/IEEE International Conference on Distributed Platforms, Toronto, Canada, 1997.
[12] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," ACM Transactions on Computer Systems, 2001.
[13] D. S. Rosenblum and A. L. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification," 6th European Software Engineering Conference. Symposium on the Foundations of Software Engineering, Zurich, Switzerland, 1997.
[14] OMG, "CORBA Event Service Specificatoin (version 1.1)," OMG, 2001.
[15] Talarian, "Talarian: Everything You Need To Know About Middleware," vol. 2003: Talarian, 2003.
[16] 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, L.A., CA, USA, 1999.
[17] Microsoft, "Building Distributed Applications with Message Queuing Middleware," vol. 2003, 2003.
[18] Sonic, "Using SonicMQ® to Extend J2EE Application Server Capabilities," vol. 2003: Sonic Software, 2003.
[19] IBM, "Websphere MQ Family," vol. 2003: IBM, 2003.
[20] SUN, "Java Message Service API," vol. 2003: SUN, 2003.
[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software: Addison-Wesley Publishing Company, 1995.
[22] R. S. Silva Filho, M. Slabyak, and D. F. Redmiles, "Web-based infrastructure for awareness based on events," presented at Workshop on Network Services for Groupware - ACM Conference on Computer Supported Cooperative Work, New Orleans, LA, USA, 2002.
[23] P. C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," ACM Transactions on Computer Systems, vol. 13, pp. 1-31, 1995.
[24] 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.
[25] 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, London, UK, 2001.
[26] 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.
[27] M. Mansouri-Samani and M. Sloman, "A configurable event service for distributed systems," in Proc. Configurable Distributed Systems. Annapolis, MD, USA, 1996.
[28] M. Clarke and G. Coulson, "An architecture for dynamically extensible operating systems," in International Conference on Configurable Distributed Systems, Annapolis, MA, USA, 1998.
[29] V. Issarny, C. Bidan, and T. Saridakis, "Achieving middleware customization in a configuration-based development environment: experience with the Aster prototype," presented at International Conference on Configurable Distributed Systems, Annapolis, MA, USA, 1998.
[30] 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.
[31] R. E. Filman and D. D. Lee, "Managing Distributed Systems with Smart Subscriptions," in Proc. International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, 2000.