Striving for Versatility in 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 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 along last years to support the requirements form different application domains. These solutions are usually not versatile in their configuration, being also incompatible with each other. They lack mechanisms that allow their use in multiple hardware and software platforms, and the flexibility to support different application domains, which requirements are in constant evolution. The YANCEES (Yet ANother Configurable Extensible Event Service) framework was designed to address these issues. It provides a pluggable and dynamic architecture that allows the customization and extension of the different aspects of event notification services. This paper presents the main challenges faced in designing extensible and configurable event notification services, proposing YANCEES as one possible solution to this problem. Extension examples and performance tests using YANCEES shows the feasibility of the architecture, which provides a relatively low performance overhead to the applications using this infrastructure.

1. Introduction

In recent years, many publish/subscribe services in special notification servers have been used as the basic infrastructure to implement of distributed applications. They have been applied in the development of applications in different domains such as: user and software monitoring [1], groupware [2], awareness for software engineering [3], workflow management systems, and 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. At one extreme, "one-size-fits-all" servers 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 the other extreme, domain-specific notification servers provide special functionalities demanded by their target applications. Examples of such systems include Khronika [7] and CASSIUS [8], which are especially designed to support groupware and awareness applications. Other examples include Yeast [9] and GEM [10], which were designed to support advanced event processing for local networks and distributed applications monitoring, respectively. More recently, servers such as Siena [11] and Elvin 4 [12] provide an intermediate approach. They were designed to address the trade-off between specificity and expressiveness of the subscription language in an Internet-scale event notification service.

In a more general sense, the popularization of publish/subscribe services has demanded increased versatility in the set of features they need to support. This versatility is concentrated in the following sub-areas:

- **Support for multiple hardware platforms**. Notification services are currently required to integrate applications running on mobile (PDAs, cell phones, lap tops), desktop, and rack mount (server) systems.
- **Support for multiple kinds of application domains**: Requirements, coming from different domains, are constantly demanding new services. Examples include Internet-scale event routing, mobility, groupware, awareness and software monitoring.
- **Usability**. As described by Nielsen [13], "different users have different needs." This differentiated use demands support at the service level for the implementation of different user interaction policies. 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 de-

signed 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 a software design perspective, in order to be versatile, a system must support the following characteristics:

- **Extensibility**: The ability to extend the infrastructure with new functionality. Extensibility is required in a constantly evolving environment, motivated by the introduction of new technologies and services. Even though this requirement is important, it is poorly supported in current notification services. The most common way to extend a server presently is by directly changing the source code or by providing the required feature as part of the client application.
- **Functional Configurability**: The ability to combine different functionality to attend the needs of specific application domains. Of importance are 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.
- **Distribution Configurability.** With rare exceptions, such as READY, current event notification service provide no support for selecting which components to execute on the client side (event producers and consumers) and which ones to execute on the server side. This requirement is especially important for mobile applications and some software monitoring tools, such as EDEM [1], which collects and processes end-user events on the client side.
- **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 interoperate. The lack of extensibility 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.

In order to address these issues, we designed and developed YANCEES, an extensible, configurable, distributable, and dynamic event-notification framework. YANCEES is based on an extensible and configurable architecture that can be customized with the use of plug-ins, filters, and services built on top of a common set of extensible languages. Initial tests with our prototype proved the feasibility of the approach, demonstrating its ability to provide different services, addressing different application domain requirements with a low-overhead performance penalty.

The main benefits of the use of a common, configurable and extensible framework is its ability to support the constant changes required by the evolution of software. The use of dynamic plug-ins improves the reuse of features in the system, allowing the build of new functionality based on composition of plug-ins. Moreover, the common model provided by YANCEES, allows the framework to be used as a basic integration mechanism between different event notification services.

This paper is organized as follows. The next section presents the analytical design framework used as the basis for the design of the extensibility mechanisms, section 3 present the main design issues, and section 4 provides the more detailed design. A case study, showing how to customize the architecture is presented in section 5, and performance tests are presented in section 6. Finally, in the last sections, related work and conclusions are presented.

2. Analytical design framework

To approach the design of notification servers, Rosenblum and Wolf [14] have proposed a framework that captures the most relevant design dimensions (or models) of these 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 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.

In our design, we considered a specialization of this framework similar to the one adopted by Cugola et al. [4]. In this model, the naming and observation dimensions are combined in what we describe as a subscription model. In this case, the way resources are identified (naming) is part of the subscription language. In addition to these models, a protocol model is introduced to describe interactions with the service that are not the common publish and subscribe operations. Mobility protocol primitives (move-in/move-out) and the event poll command are some examples.

3. Design issues

In the design of the system, the challenge was to define an architecture that allows the customization and extension of the dimensions presented in the design framework without a very significant loss of performance. The architecture also needs to cope with different configurations. In short, the architecture needs to:

- Support different requirements associated to the models of the design framework, especially the event, subscription, notification, and resource and protocol models.
- Provide extensibility through mechanisms that allow a programmer to define and implement new capabilities to the models defined above.
- Support different configurations, sets of services (or features) that may work together to provide the functionality necessary for the application domains.
- Permit the distribution of components and services among the publishers and subscribers to comply with device limitations, prevent performance bottlenecks, and implement services that require some degree of distribution, such as mobility.

4. Design

The YANCEES framework was designed to provide different extension points around a common publish/subscribe core. These points correspond to the dimensions proposed by the design framework discussed in section 2, being extended by the combined use of languages extensions and plug-ins. Plug-in instances are dynamically created and combined, based on the syntactic rules of these languages. This dynamic allocation of plug-ins is performed by special parser components.

Additional support to extensibility is provided by shared services and input and output filters, which can be used by the plug-ins to implement their functionality.

The publish/subscribe core provides a common event model. The subscription model used in the core is content-based, and the events are represented as attribute/value pairs, which provide generality to the system. As a consequence, the timing model of the whole system is a function of the central publish/subscribe core, which, in our prototype, is provided by the event dispatcher component.

In summary, the extensibility and configurability of the system is achieved by the use of the following strategies:

- Extensible languages to each design dimension
- Dynamic allocated plug-ins
- Parsers that combine and allocate plug-ins according to the extensible languages syntax.
- Input/output filters and shared services as auxiliary elements in the implementation of new extensions
- An architecture configuration manager to statically or dynamically load configurations of plug-ins, services, and filters to each application domain
- A central publish/subscribe core providing basic content-based filtering

The components that implement this strategy and their correlations are described in more detail in the next section. A more detailed description on how to extend the languages and how to write plug-ins, filters, and services are described in section 4.3.

4.1. General YANCEES architecture

The main components and interfaces of the architecture are presented in Figure 1. The static components and APIs are drawn in gray, whereas dynamic allocated components are depicted as dashed line boxes.

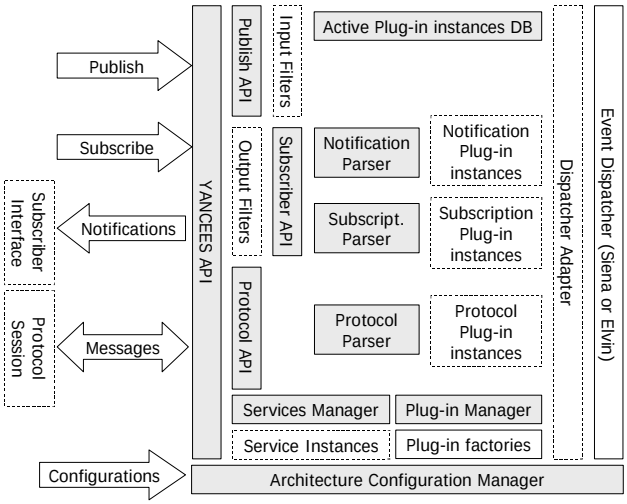

Figure 1 General framework static and dynamic components

4.1.1. Extensible languages

The interaction with the system occurs through message exchanges. These messages are defined according to different language specifications. The main role of the framework is to efficiently process and route these messages from publishers to subscribers. The architecture is able to process different message categories (events, subscriptions, notifications, and protocol messages), which correspond to the design dimensions presented earlier.

Event Language. Defines the representation of the events to be processed.

Protocol Language. Defines messages used to implement different system interaction protocols.

Subscription Language. Allows the expression of queries, represented as event content-based expressions.

Notification Language. Specifies policies and mechanisms to be used in the delivery of the events to subscrib-

ers.

Specific parsers were created to interpret messages defined according to these languages. These parsers are associated to different dimensions of the design framework.

4.1.2. Main components

The main components of the system are described in more detail as follows.

Parsers. Parsers are the primary message processing components. They are used to dynamically allocate plug-ins that extend the notification, subscription and protocol models. Plug-ins are used to implement the semantics of extensions in the subscription, notification, or protocol languages. Parsers are used to hierarchically assemble plug-ins in evaluation trees based on the content of subscription messages. This strategy allows the more abstract plug-ins to depend on the functionality provided by simpler plug-ins, thus permitting the incremental extension of the system and reusing previously installed functionality.

For example, consider a subscription with a pull notification policy as follows (described in an algebraic form):

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

This expression is parsed in the subscription evaluation tree, using different plug-in instances as described in Figure 2.

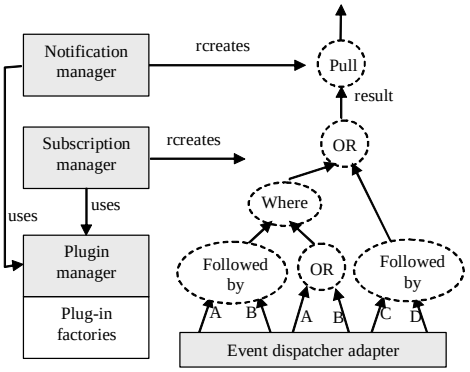

Figure 2 Subscription and notification evaluation tree using plug-ins

As depicted in Figure 2, the subscription evaluation tree comprises nodes, which are plug-in instances that communicate using the simple publish/subscribe design pattern [15]. In this tree, each level subscribes to its children nodes' results. The subscription "`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 registers itself as a listener to the results of each tree root plug-in. These results can be a new event produced due to the evaluation of an expression, or a set of events that matched the specific expression.

Plug-in manager. This component manages and creates instances of all plug-ins required by a design dimension. It also copes with runtime change, allowing new plug-ins to be installed at runtime. It makes possible the dynamic allocation of plug-ins, by registering plug-in factories under specific language keywords. For example, if a sequence detection plug-in is installed under the `followed-by` keyword, whenever this element is found in a subscription message, the plug-in registered under this name is allocated to process this expression.

Dispatcher adapter. The Dispatcher Adapter isolates the system from the idiosyncrasies of the event dispatcher component. The adapter acts as a translator between the internal event representation and the components and the event dispatcher component.

Event dispatcher component. The event dispatcher component provides the basic event and timing models used throughout the system. It provides content-based filtering and has its own attribute/value pair event representation. The distinction between this component and the Dispatcher adapter allows the use of notification servers such as Elvin or Siena as this component.

Service manager. Whereas plug-ins are dynamically allocated, services are static components. Their main purpose is to support the features implemented by the plug-ins. For example, services can be defined to provide context and shared resources or can be used to intermediate the inter-plug-in communication.

Filters. Input and output filters can be installed in order to perform pre and post processing on incoming events and out coming notifications, as well as to enforce system-wide policies, collecting statistics and other extensions.

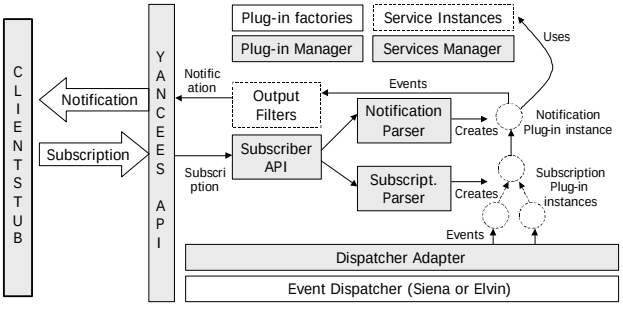

Figure 3 A generic subscription workflow

To illustrate the flow of events through plug-ins, filters and parsers, Figure 3 presents the parsing of a typical message (other components are omitted for clarity).

First, the YANCEES API receives a subscription message that also contains a notification delivery policy.

Then, the message is routed to the Subscription parser, which allocates plug-ins according to the commands in the message. At the same time, the notification manager parses the notification policies and allocates plug-ins to handle the event delivery. Finally, both trees are combined in order to compose a single subscription evaluation tree. This tree keeps active in memory, processing incoming events, while the subscription remains active. Once the subscription is evaluated successfully, the resulting notifications are sent back to the subscriber through the output notification filters. Examples of how to create and use plug-ins are provided in the next sections.

Configuration Manager. This component coordinates the installation and configuration of all the SUB-components of the system. It also allows the dynamic installation of services, filters and plug-ins, providing dynamicity and configurability to the framework.

Client Stubs. The location of the components of the system is not limited to the server-side. Plug-ins and parsers can be installed in the client-side to perform client-side event processing. This is important, for example, in mobile applications (which require disconnected operation), usability monitoring (which collects and processes events on the client-side) and load balancing (which distributes processing throughout the hosts of the system).

4.2. Implementation

A prototype of the YANCEES framework was implemented using Java 1.4 and the Java API for XML Processing (JAXP) v1.2.3, which supports XMLSchema [16]. The XMLSchema provides inheritance and extensibility mechanisms. When used with the Java DOM parser, an XML document defined according to an XMLSchema can have its syntax automatically validated. This resource is used in our implementation to guarantee the correct form of the messages.

The event dispatcher component used in the implementation was Siena 1.4.3. Elvin 4.x was also used in the tests. Both support content-based subscriptions and federation of servers, and both represent events as attribute/value pairs.

In our current prototype implementation, two component distribution configurations are possible: (1) the execution of all components on the client stub; or (2) the execution of all components on the server-side. In the former case the communication between client stub and server side (Siena or Elvin) is performed by using their native protocols; in the latter case, the communication between client stubs and the YANCEES server is performed by using Java RMI.

4.3. Extending YANCEES

In order to validate the system, some extensions were developed. This section presents some of these extensions, showing the steps necessary customize the YANCEES framework to provide awareness-related features.

The steps necessary to extend the system are as follows:

1) Extend the subscription, notification, or protocol model languages by using the extensibility facility provided by the XMLSchema language.
2) Implement the plug-ins that will be invoked to execute the new extension tags of the language by providing a factory to activate the plug-ins.
3) Implement auxiliary filters and services, if necessary.
4) Create or change an existing configuration file to include the new components and register their interdependencies.
5) Restart the server or dynamically reconfigure it.

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.

The components of the system (plug-ins, filters, services and event adapter) are used in terms of their interfaces. These interfaces are presented as follows.

5. Case study: implementing CASSIUS services with YANCEES

To demonstrate the main extensibility points of the YANCEES framework, it was adapted to support awareness applications. Notification servers such as Khronika [7] and CASSIUS [8] provide us with a list of the basic services needed by these applications. 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 service, called event source browsing, provides information about the publishers and their events.

5.1. Implementing sequence detection

Event sequence detection required the extension of the subscription language with a new keyword: <sequence>. It operates over a set of content-based filters, which are natively provided by the event dispatcher component.

The first step in to extend the YANCESS subscription language with the new `<sequence>` tag. This is illustrated in the code fragment of Table 1 as follows.

Table 1 sequence detection extension to the subscription model (code fragment)

```
<complexType name="SequenceSubscriptionType">
  <complexContent>
    <extension base="sub:SubscriptionType">
      <sequence minOccurs="0" maxOccurs="1">
        <element name="sequence" type= "Fil-
terSequenceType" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
<complexType name="FilterSequenceType">
  <sequence minOccurs="1" maxOccurs="unbounded">
    <element name="filter" type="FilterType"/>
  </sequence>
</complexType>
```

The next step is to implement the sequence detection plug-in, extending the *AbstractPlugin*, a convenience class that provides default implementations to the plug-in interface presented in Table 2 as follows.

Table 2 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(); }
```

Note that the *PluginInterface* is a listener to events produced in other plug-ins. This is the key feature used to implement our sequence detection plug-in. It implements the interface in Table 3 as follows.

Table 3 Plug-in listener Interface

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

A simple sequence detection implementation will collect, in the right order, events coming from each one of the filters it observes. When a successful sequence is detected, the sequence plug-in returns the set of events collected, publishing it to 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 in Table 4 must also be defined.

Table 4 Plug-in factory interface

```
interface PluginFactoryInterface {
  String[] getTags();
  PluginInterface createNewInstance (Node sub-
Tree); }
```

A simple factory implementation will return a new instance of the plug-in only 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 described in Table 5, followed by the restart or reconfiguration of the YANCEES service.

Table 5 Sequence detection plug-in configuration file (code fragment)

```
<architecture> ...
  <subscription>
    <plugin>
      <name> sequence.plugin </name>
      <mainClass>
        <javaClassName>
          plugin.sequence.SequencePlugin
        </javaClassName>
      </mainClass>
      <factoryClass>
        <javaClassName>
          plugin.sequence.SequencePluginFactory
        </javaClassName>
      </factoryClass>
      <depends> siena.plugin </depends>
    </plugin>
  </subscription> ...
</architecture>
```

The plug-in is then ready to be used. It will be activated each time a subscription is provided that uses the <*sequence*> tag in its body. 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 xsi:type="FilterSequenceType">
    <filter xsi:type="FilterType">
      <EQ>
        <name> status</name>
        <value> Fail </value>
      </EQ>
    </filter>
    <filter xsi:type="FilterType">
      <LT>
        <name> cooler Temp </name>
        <value> 90 </value>
      </LT>
    </filter>
  </sequence>
</subscription>
<notification> <pull/> </notification>
```

5.2. Implementing a pull delivery mechanism

Pull delivery allows subscribers to periodically poll (or inquire) the server for new events matching their subscriptions. This mechanism copes with the requirements of some mobile applications, for example, where subscribers usually get temporarily disconnected, allowing the storage of events during the disconnection period.

In the YANCEES architecture, this mechanism is provided by a combination of plug-ins. The first plug-in is the pull notification plug-in. Instead of sending the notifications directly to the subscribers, this plug-in stores the events in a temporary repository. This repository is implemented as another component, the event persistency service.

In addition to the pull notification plug-in and the event persistency service, users need a way to control when to collect and when to store the events being routed to them as a result of a subscription. This functionality is provided by a polling interaction protocol plug-in. In short, the implementation of a pull delivery mechanism requires:

- The extension of the notification language to add pull support
- The implementation of a pull notification plug-in
- The implementation of a persistency service
- The definition of a polling protocol
- The implementation of a polling protocol plug-in

The implementation of the pull notification plug-in follows the same steps required to implement the sequence detection plug-in, as previously described. For the pull notification plug-in, instead of extending the subscription language, the notification language is extended. This extension defines a new <pull> tag. Additionally, a factory to instantiate this plug-in must also be provided. As a consequence, after installing these components in the system, whenever a <pull> tag appears in the <notification> part of a subscription message, a plug-in instance will be created to handle this notification policy (see Table 6).

The pull notification plug-in implementation is very simple; it directs the events to the persistency service component and registers them under their target subscriber interface.

Finally, the poll protocol plug-in must also be implemented. It responds to commands such as <poll-interval> <stop-polling> and <poll>, which define different polling mechanisms. It collects the events stored in the persistency service, and delivers them periodically to the subscriber (poll-interval command); it then collects the notifications whenever requested (using the poll command) or deactivates the periodic delivery (using the stop-polling command) in case of a temporary disconnection.

These sets of plug-ins define a configuration, a set of components that need to be present in order for a service to operate. The dependencies between these are checked by YANCEES with the help of the `<depends>` clause in the configuration file (see Table 5). An example with all the plug-ins discussed here is presented in Figure 4.

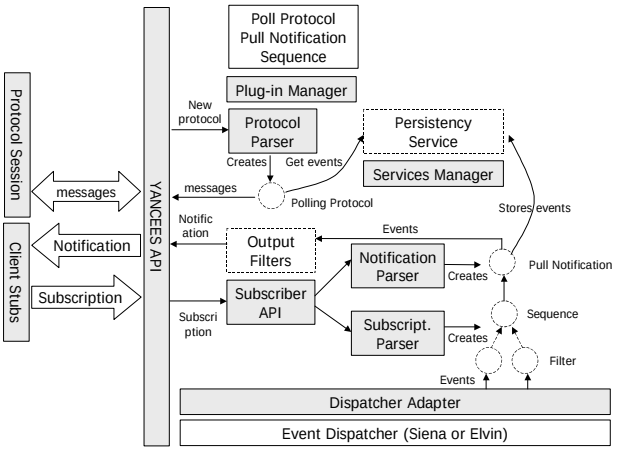

Figure 4 Example configuration using sequence detection, pull notification and poll protocol

In the example of Figure 4, filters perform content-based filtering, interacting directly with the dispatcher adapter. When filters subscriptions are matched, the events are sent to the sequence plug-in, which checks for the order these events are produced. If the sequence is right, a notification with all matched events is sent to the pull notification plug-in, which stores it in the persistency service. The polling protocol plug-in interprets the protocol messages coming from the end-user through a protocol session. Whenever a poll message is received, the events stored for a given subscription are collected and sent to the end user.

5.3. Implementing CASSIUS 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.

These events can be easily identified and checked for correctness by an input filter. This filter performs type checking on all incoming events, identifying and validating events represented according to the CASSIUS template format. Once a CASSIUS event is identified and validated, it is copied 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.

The poll mechanism is not the only way to collect CASSIUS events. At any time, subscriptions can also be performed on regular CASSIUS events.

6. Performance tests

The YANCEES architecture was evaluated with respect to the additional computational overhead of the framework. It was compared to Siena and Elvin as a way to estimate the computational delays introduced by the use of the framework.

In our tests, we analyzed the delays associated with the basic publish/subscribe operations as well as the notification delay: the time between the publishing of an event and its notification to the subscriber. The tests were performed on a Pentium III 750MHz system with 139MB RAM, running MS-Windows 2000.

The tests used a simple event with two attributes and a simple subscription that queries for events with those attribute names/values. The following tests were performed with Siena, Elvin, and YANCEES using Elvin and Siena as dispatcher components:

Publish: 1000 events were published in batch, and the average time to process the *publish*() requests was measured.

Subscribe: 1000 subscription requests were posted, and the average time to process the *subscribe*() command was measured.

Notification: a simple subscription was posted, and the average elapsed time between publishing and receiving each of the 1000 events was measured.

In all of the tests, all YANCEES components were installed in the server-side. The communication between publishers and the server was done using Java RMI.

The average times collected during the tests are presented in Table 7. The low delays associated with the invocation of publish and subscribe commands using Elvin and Siena alone is explained by the non-blocking implementation of these methods. The current YANCEES prototype does not provide non-blocking method invocations.

Table 7 Average delays for basic operations (ms)

	Publish	Subscribe	Notification
Elvin	0	0	132
Siena	1	0	98
YANCEES using Elvin	78	86	184
YANCEES using Siena	98	89	162

The times in Table 7 do not include the delays associated with the creation of the event and subscription objects for all notification servers tested. In the special case of YANCEES, this includes the parsing of XML documents into DOM trees in memory. Because this operation is performed on the client-side, before the invocation of the *publish*() and *subscribe*() methods, they were not computed in the tests. The average event parsing delay was 43ms; whereas the average subscription parsing delay was 84ms for the event and subscription XML files used in the tests respectively.

The tests show an increase from 60% (Elvin) to 70% (Siena) of the delay of the whole notification cycle, when YANCEES is used to perform the same functionality as Elvin or Siena. This is the computational price to be paid for the additional extensibility and configurability obtained by the use of the YANCEES framework.

7. Related work

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

7.1. Pluggable and programmable routers

The Click Modular Router [17] 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 [18] 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. The filters in our model follows a similar approach.

7.2. Configurable distributed systems

Software architectures and event-based systems can be combined to provide a framework to support runtime configuration. Oreizy and Taylor [19], 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 [20].

The 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.

7.3. Configurable middleware

Configurable middleware services have been described in the literature. For example, TAO [21] 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. 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.

8. Conclusions and future work

In this paper, we presented YANCEES, an extensible, configurable, and dynamic framework that provides a set of common and extensible models that, together, allow the implementation of customized event notification services to support different application domains. This approach permits the provision of the right configuration to the appropriate application domain. The framework is also dynamic, allowing the combination and installation of services at runtime.

The extensibility mechanisms are based on the design dimensions proposed by Rosenblum and Wolf [14], plus a protocol dimension that captures different interaction mechanisms. The framework dimensions are extended by use of extensible languages, plug-ins, filters, and services, which are dynamically combined by parsers.

The use of a single extensible infrastructure 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, that can be dynamically allocated and used by different plug-ins.

These benefits come through a price. Performance tests with a prototype of the framework showed that YANCEES is about 70% to 60% slower than current event notification servers such as Siena and Elvin. This delay in performance, however, is compensated by the gain in configurability and extensibility.

A case study explaining how to extend and adapt the framework to provide services implemented in other notification servers such as CASSIUS was presented, demonstrating the feasibility of the model.

As future work, we plan to experiment with the use of the YANCEES framework to integrate different event services over a common model, which will require the refinement of the framework to address issues related to this integration.

Another issue to be exploited is scalability. Exploiting the use of the framework in Internet-scale networks of notification servers, such as those supported by Siena.

The dynamic allocation of plug-ins also opens the possibility for applying subscription optimization algorithms such as the Rite algorithm and other approaches [22] that optimize the sharing of plug-in instances at runtime.

The use of XML as subscription and event representation languages jeopardize the usability of the system. Improvements in the user interface with the system are necessary. 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 improvement of the distribution of the components, which currently can be only be located entirely on the client or server sides, is another future goal.

9. 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.

10. 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] Anita Sarma and A. v. d. Hoek, "Palantír: Increasing Awareness in Distributed Software Development," presented at International Workshop in Global Software Development at ICSE'2002, Orlando, Florida, 2002.

[4] 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.

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

[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] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley Publishing Company, 1995.

[16] W3C, " XML Schema Part 0: Primer. W3C Recommendation," 2001.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] R. E. Filman and D. D. Lee, "Managing Distributed Systems with Smart Subscriptions," presented at Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), Las Vegas, Nevada, USA, 2000.