Striving for Versatility in Publish/Subscribe Infrastructures

Roberto S. Silva Filho   David F. Redmiles

Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA, USA 92697-3430

{rsilvafi, redmiles}@ics.uci.edu

ABSTRACT

Publish/subscribe infrastructures are used as the basic communication and integration framework in many application domains. The majority of those infrastructures, however, fall short of mechanisms that allow their customization and configuration to comply with the requirements of those application domains. In other words, they are not versatile enough to support new and evolving requirements demanded by different applications. The YANCEES (Yet ANother Configurable Extensible Event Service) addresses these versatility issues by relying on a combination of plug-in oriented architecture and extensible languages decomposed over different design dimensions of a publish/subscribe infrastructure. We demonstrate our approach, showing how the YANCEES platform can be useful in reducing the customization, extension and implementation effort of different publish/subscribe infrastructures to attend the demands of many application domains.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain Specific Architectures; D.2.13 [**Reusable Software**]: Domain Engineering; H.4.3 [**Communications Applications**]: Information Browsers;

General Terms

Design

Keywords

Publish/Subscribe, notification servers, event-based middleware, flexible architecture, plug-ins and extensible languages application.

1. INTRODUCTION

Publish/subscribe infrastructures (a.k.a. Event Notification Services) have been used as the basic communication and integration infrastructure for many application domains such as user and software monitoring [15], groupware [7], collaborative software

engineering [25], workflow management systems and mobile applications [4], among many others. This wide range of applications has required new services from the publish/subscribe infrastructure such as advanced event processing (event sequence detection, abstraction, and summarization); event persistency, mobility support, transactions, secure communication channels, and a whole new set of domain-specific functionality. As a consequence, in spite of the availability of standardized solutions such as CORBA-NS (CORBA Notification Service) [22] or JMS (Java Message Service) [29], new publish/subscribe infrastructures continue to be developed to address the needs of novel applications.

In this context, the proliferation of specialized solutions reveals limitations on the way event-based infrastructures are being designed and built. First and foremost, the publish/subscribe paradigm appears seductively simple. A basic service can be programmed quickly before the complexities of the application it serves reveal themselves. Then, when complexities manifest, they require significant extensions already implemented in existing, sophisticated infrastructures. A second deterrent is that current publish/subscribe infrastructures are not designed to be extensible nor programmable, which hinders the addition or customization of new application services. For instance, CORBA-NS does not support event source discovery protocols, such as those provided by CASSIUS [17]. The implementation of this feature using CORBA-NS would require the direct change of this publish/subscribe infrastructure source code or even aspects of the client application. Third, with rare exceptions such as the READY [14] (a CORBA compliant notification service), current solutions are not configurable with respect to the place where event processing happens in a distributed setting, a feature important in some application domains. For instance, some software monitoring applications as EDEM [15], require the execution of event processing on the application side, where the events are collected and abstracted; whereas applications running on mobile devices may need a restricted set of services and features. Forth, with the proliferation of specialized infrastructures, interoperability becomes a problem. In large organizations, for the reasons previously mentioned, it is common to find different event-driven applications, designed for specific purposes, that rely on different event-based infrastructures. Due to differences in purpose and scale, they usually do not interoperate. For example, server monitoring applications, e-mail servers, workflow management systems and others, that do not share a common data format, data schema or even computing platform. Finally, with the exception of a few research prototypes, discussed at the related work section of this

paper, none of existing publish/subscribe infrastructures support the customization and evolution of the services and features they provide.

In this paper we present YANCEES (Yet ANother Extensible Event Service), a new and experimental framework designed to provide versatility to publish/subscribe infrastructures. YANCEES' goal is to support existing and new requirements demanded by current collaborative and software engineering applications, a field in constant evolution. YANCEES is based on the key observations that publish/subscribe systems can have their functionality decomposed over different design dimensions following a design framework that captures the main aspects of such systems. YANCEES allows the customization, extension and programming of such dimensions with the use of extensible languages and plug-ins [31]. It builds upon custom-made or existing publish/subscribe infrastructures such as Elvin [12] and Siena [2], allowing their extension with additional features. This strategy results in an infrastructure that can be easily tailored to the needs of different applications through the reuse of existing components and notification servers.

1.1. Publish/subscribe design dimensions

In order to understand the concerns involved in the design of a publish/subscribe infrastructure, Rosenblum and Wolf proposed an analytical design framework described in [24]. 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. As proposed by Cugola et al. [4], we combine the naming and observation models in the subscription model.

This design framework, however, does not consider additional services, other than the publication and subscription of events. Current publish/subscribe infrastructures has demanded support for different interaction protocols, as the example of mobility [4], Internet-scale event notification systems [2], context-aware applications [1], peer-to-peer networks [5], and the wide use of publish/subscribe infrastructures in different application domains. Hence, we build upon Rosenblum and Wolf's framework by introducing a new dimension to this model, the protocol. The protocol model is necessary to capture other forms of interaction with the publish/subscribe infrastructure that goes beyond the common publication, routing and notification of events.

Finally, this design framework did not account for differences between interaction and infrastructure aspects of the publish/subscribe infrastructures. The interaction with the publish/subscribe system is generally not only mediated by simple publish/subscribe programmatic APIs, but also through subscription and notification languages. This dual characteristic (language and APIs) makes the study of versatility in the context of publish/subscribe infrastructures a challenging endeavor, since one must match the configuration, variability and evolution of the infrastructure with the evolution and variability of these languages.

2. VERSATILITY MOTIVATION

As observed by Parnas [23], the majority of software systems are not designed for change and evolution. Instead, they are built to solve specific and well defined problems which end up hindering their ability to evolve, resulting in high maintenance and evolution costs. Publish/subscribe infrastructures are not an exception to this observation. In the light of this problem, Parnas proposes the concept of **flexibility,** which states that software must be designed and implemented not as a single program, but as a family of programs that can be extended and contracted according to different application needs. Our notion of versatility is based on this original definition of flexibility, and incorporates additional design properties that are important to current publish/subscribe infrastructures. Parnas' observations, even though still current and valid did not explicitly mention nor predict other kinds of concerns such as runtime (dynamic) change, load balancing or distribution of processing (between client or server sides for example), and usability. The first two issues are central to distributed systems and publish/subscribe middleware, whereas the latter is essential for the acceptability and usefulness of the proposed approaches. Based on this motivation, we proceed to present our concept of versatility.

In the light of the above discussion, we proceeded to research ways of providing and maintaining good software engineering qualities that allows the customization, expansion and contraction of publish/subscribe infrastructures in a usable way. For such, we adopted the term, versatility. Moreover, we sought a new term that could be applied not only to technical needs but to the varying needs of human stakeholders and application workplace settings. Hence, from a software engineering perspective, and more specifically in the context of publish/subscribe infrastructures, versatility comprises the following requirements.

Evolution Support allows the publish/subscribe service to incorporate changes due to new (functional and non-functional) application-level requirements. Evolution is accomplished by support for *extensibility, programmability* and *reuse*. **Extensibility** encompasses all classes of enhancements that can be made in the system without changing the existing functionality (or contract), for example, the addition of new functional behavior, such as advanced event processing, which adds to the current subscription language, while maintains backward compatibility with existing publish/subscribe API. **Programmability** allows the customization and modification of the behavior of existing software. For example, the reconfiguration of the publish/subscribe infrastructure to support different event representations (such as records, objects, or tuples), or the built of new subscription language, based on regular expression queries and different event delivery mechanisms. Programmability can also be used to define new federation and interaction protocols with the notification service. **Reuse** allows the modularization of certain aspects of software, permitting the incorporation of existing functionality, wrapped as special software pieces (or components), in the construction of new software [18]. For example, existing subscription filtering functions can be reused in the implementation of more advanced filtering and event processing commands.

Variability (or scalability) allows the contraction and expansion of software in order to support different functional and non-functional requirement sets. For example, allowing the built of thin pub/sub infrastructures to run in hardware restricted devices, or more complex systems, to run in fast server machines; as well as the ability to distribute event processing over client or server sides.

Usability. In order to be useful, and fulfill its purpose, software must be usable [21]. The cost associated to learning, and using a customizable and extensible piece of software must not exceed the total cost of building an application-specific infrastructure from scratch.

Besides the above qualities of versatility, publish/subscribe infrastructures need to support the essential middleware requirements of scalability, interoperability, heterogeneity, network communication and coordination [10] which must co-exist with the versatility properties we propose.

3. APPROACH

Our approach strives to provide the versatility properties discussed in the last section. It is based on the combination of plug-in and extensible languages in the implementation of new publish/subscribe infrastructures that better meet existing and new application requirements.

Plug-in based software development can be defined as a special case of component-based software development which supports the evolution and customization of the features of the application by the use of plug-ins [31]. A well known example is the Eclipse IDE[1] which can be extended with new software tools. Plug-ins rely on configuration management provided by the system runtime environment (or kernel), rather than the user, allowing graceful upgrading of systems over time without requiring application restart [3, 20]. The runtime environment manages: (1) plug-in runtime activation and deactivation; (2) plug-in registry, a list of installed plug-ins; and (3) inter-plug-in dependencies management. Optionally, the kernel can support other services such as logging, security, and so on. For such characteristics, plug-ins have been used in applications demanding modularization and footprint control, extensibility, as well as runtime change and upgrade.

Another emerging approach to versatility that has been used in different application domains, is the combination of extensible languages such as XML (the Extensible Markup Language), and plug-in based software development [31]. This combined approach is usually motivated by the need to cope with different languages, tailored at different application domains that share a common infrastructure. While language extensions allow the expression of domain-specific concerns, plug-ins are used to implement this functionality in the underlying infrastructure.

Together, the use of extensible languages and plug-in oriented software development is particularly attractive to publish/subscribe infrastructures domain since they combine the extensibility of languages such as XML with the runtime change and dynamic characteristics of plug-ins. The extensible languages also can be syntactically and semantically parsed at runtime, pre-

---

[1] http://www.eclipse.org

venting inconsistent combinations of plug-ins to take place, thus avoiding the definition of conflicting plug-in configurations. YANCEES demonstrates the application of those techniques to the publish/subscribe versatility problem as described in the next session. A general overview of the approach is presented in Figure 1 as follows.
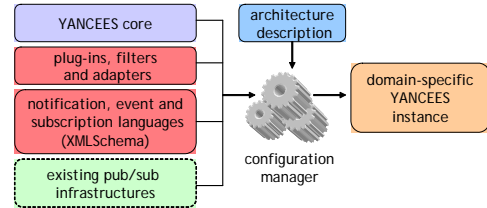

Figure 1. YANCEES general approach summary

3.1 YANCEES architecture overview

The YANCEES framework was designed based on the key observation that (1) the design dimensions of a publish/subscribe system represent variability points where different combinations of functionality can be plugged; (2) and that the required functionality subset is governed by composition rules expressed in the form of subscription and notification languages. YANCEES makes these extension points explicit, allowing their extension and programming through the combined use of extensible languages and plug-ins. Those extensions are wrapped up into configurations, allowing the fine-tuning of the infrastructure according to different application requirements. The whole process is automated by the use of dynamic message parsers and configuration managers. A general picture of the YANCEES framework is shown in Figure 2.
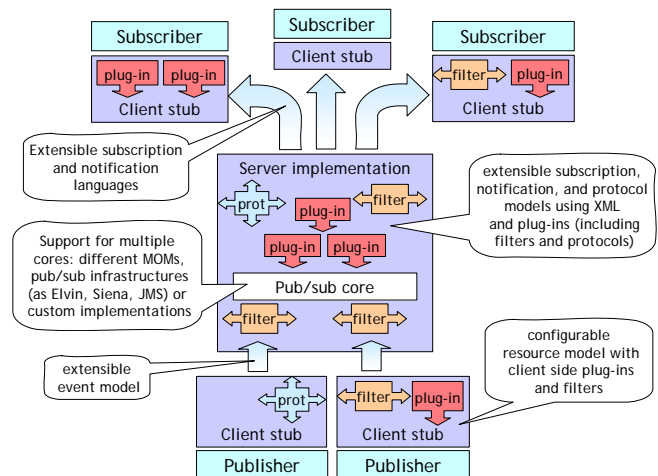

Figure 2. General view of the YANCEES framework.

In YANCEES plug-ins can be defined to perform event correlation (sequence detection, aggregation, abstraction and so on), notification policies (such as push and pull), and protocols (mobile primitives such as move-in/move-out). Plug-ins can be either installed in the server side or in the client side (publishers or subscribers), allowing the distribution of event processing functionality among these sites. The publish/subscribe core can integrate custom or existing publish/subscribe infrastructures such as Elvin and Siena.

3.2 Main YANCEES Components

Internally, YANCEES is programmed by the dynamic composition of different kinds of components used to augment existing or new publish/subscribe cores. Those components are defined ac-cording to four main categories: subscription and notification plug-ins, protocol plug-ins, filters and adapters. They implement or manipulate three kinds of extensible languages: event, subscription and notification.
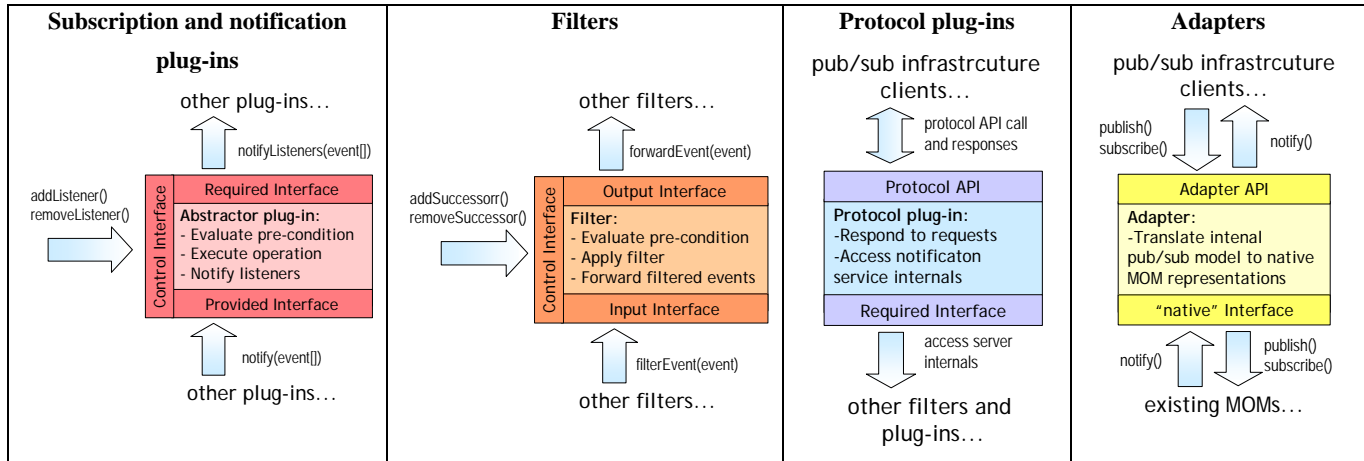

Figure 3. Kinds of plug-ins supported in the YANCEES platform

Subscription and notification plug-ins are used to program and extend the notification and subscription models. As described in Figure 3, they are responsible for implementing specific command sets in the subscription and notification languages respectively. They implement the listener pattern [13], a simple publish/subscribe model that allow their dynamic composition into event processing hierarchies (or threes). Using this approach, plug-ins can be composed to perform advanced event processing operations. For instance, plug-ins such as event sequence detection can depend on other plug-ins that perform content-based filtering with the help of the pub/sub core.

Filters (second column of Figure 3) are used to select or restrict specific events from an event stream. Hence, they are used to extend the event model or to perform filtering in the subscription and notification models. For example, they can be used to evaluate content-based event expressions, apply security policies throughout the system, enforce event type checking (thus extending the event model), invoke operations in other plug-ins in response to some special and many other operations. Filters are basically composed in sequence, according to the chain of responsibility design pattern [13].

Protocol plug-ins (third column of Figure 3) allow the extension and programming of the infrastructure to support new protocols. They are dynamically loaded components that can be used to implement different interaction protocols with the server. They have access to the internals of the infrastructure and can interact with other plug-ins. Example of protocol plug-ins include mobility protocols (move-in, move-out primitives implementation), user authentication, and event source discovery as supported by CAS-SIUS notification server. Protocol plug-ins can also be configured as **static plug-ins** (a.k.a. services) that, instead of being activated over demand, persist over many activations. An example of a static plug-in is an internal data model or JDBC connection.

Adapters (forth column of Figure 3) allow the integration, extension and programming of the event model and cope with interope-rability. They translate the internal YANCEES event representation to different, core-specific event models. They also provide a standard publish/subscribe interface. Combined those features allow the system to interact with existing publish/subscribe infrastructures such as message-oriented middleware as JMS, or notification servers as Siena and Elvin, or to implement new routing algorithms. For example, YANCEES can be configured with one or more pub/sub cores, connected by different adapters, in order to extend existing systems with advanced event processing features, or as a way to integrate different publish/subscribe networks.

Configuration managers and dynamic parsers. Those components manage the variability of the system. Whereas adapters, protocol plug-ins and filters are statically combined following user-defined configurations, interpreted by YANCEES configuration manager; subscription and notification plug-ins are loaded at runtime, by dynamic parsers, as the users express their interest in events by posting subscriptions. More details on those particular components and their algorithms are provided elsewhere [28].

3.3 Implementation

The YANCEES prototype is composed of four main sub-parts: (1) the extensible framework itself, which provides automated subscription, notification and event parsing; as well as configuration management; (2) a set of generalized publish subscribe adapters, plug-ins and filters, (3) the subscription, notification and event languages, and (4) existing or custom-made pub/sub cores (see Figure 1).

The framework was implemented in Java 1.4 and the Java API for XML Processing (JAXP), which supports XMLSchema that provides inheritance and extensibility mechanisms used to define the subscription, notification languages, as well as different event representations. It also allows subscriptions, events and notification languages to be validated and grammatically checked, preventing inconsistencies and misuses of the language. In our cur-

rent prototype the communication between clients (publishers and subscribers) and the YANCEES service is performed by using Java RMI.

The framework provides a generalized implementation of the publish/subscribe model, externalizing methods such as PUBLISH, SUBSCRIBE and UNSBSCRIBE to the end users. These methods operate over generic types that encapsulate XML messages representing subscriptions, notifications and events. Protocols are handled by the special CONNECT_TO_PROTOCOL method in the YANCEES API, which returns a remote reference to a specific plug-in. Preliminary tests show an additional overhead around 100ms in the subscription time (a consequence of XML technology), and an extra 50ms in the event routing if compared to Elvin [12] and Siena [2] alone (all measured on a 3GHz Pentium IV). The YANCEES event routing, however, is optimized, yielding a top throughput of 10000 events/second that, in our tests, was superior to Elvin and Siena, when events are sent in a high frequency [27]. This apparently conflicting result is a consequence of the buffering strategy of YANCEES, that minimizes traffic between publishers and subscribers by grouping and sending more than one notification at once (instead of one message per notification). This strategy is transparent to the users. It is also important to mention that those delays are compatible with the software engineering and groupware applications we are currently supporting. The programmatic interface has more functions than described here. However, space limitations prohibit us from discussing them all.

YANCEES supports both distributed and centralized configurations. When using Elvin and Siena, for example, it inherits their ability to support Internet-scale publish/subscribe networks. If necessary, different federation protocols can be implemented in the infrastructure, with the help of protocol plug-ins. An example of tool that uses a special YANCEES configuration for peer-to-peer applications is discussed elsewhere [6].

We have implemented subscription plug-ins that provide event correlation (event sequence detection) (257 LOC) as well as topic-based event routing (268 LOC) and content-based event filtering based on Elvin and Siena pub/sub cores (3000 LOC). Plug-ins that support push and pull notification policies were also implemented (50 LOC); as well as protocol plug-ins that support the polling of events (200 LOC). Additional plug-ins for event persistency and CASSIUS-like event hierarchy management are provided (1100 LOC), as well as loggers and type checking filters. Some of those plug-ins are discussed in the case studies presented in the next section. More plug-ins are being developed. YANCEES is available for download in the website[2].

4. CASE STUDIES

In this section we present two evaluation case studies, showing how YANCEES design dimensions can be configured to support different application requirements. For the lack of space, we focus on describing the main plug-ins and their configurations, used in each case. A more detailed description on how to extend YANCEES from a programmer's perspective is described elsewhere [27]. In order to illustrate our approach, we assume YANCEES is initially configured with a simple content-based subscription

---

[2] http://www.isr.uci.edu/projects/yancees

model, based on tuple-based event model and push notifications similar to those provided by Siena [2] or Elvin [12].

4.1 Evaluation Case Study 1: Extending and configuring YANCEES to support awareness applications

In this case study, we extend the framework to provide the main services required by awareness applications. The requirements of this domain are based on the set of functionalities provided by notification servers such as Khronika [19] and CASSIUS [17]. Hence, in order to be functionality-compatible with those notification servers, YANCEES needs to provide: event persistency and typing, event sequence detection, and pull notification delivery policy. Moreover, a special feature provided by CASSIUS is the ability to browse and later subscribe to the event types that are published by each event source. This feature, called event source browsing, provides information about the publishers and their events, and requires an API for event source registering and browsing. A summary of the extensions implemented in YANCEES is presented in Figure 4. For simplicity, other components such as parsers, factories and so on are not represented.
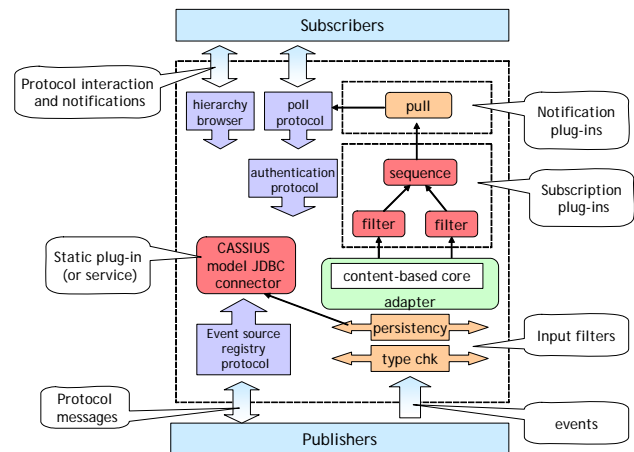

Figure 4. YANCEES configuration implementing CASSIUS-equivalent functionality.

Subscription model. An important feature in awareness applications is event sequence detection. In order to support this feature, the original subscription language is extended with a new keyword: <*sequence*>, which depends on the existing <*filter*> command that performs content-based filtering. In other words, grammatically, a sequence detector operates over a set of two or more filter commands. A sequence detection plug-in is then implemented and registered in the system. It operates over a set of content-based plug-in filters (which are assumed to be already available), that interact with a content-based publish/subscribe core, already installed in the YANCEES framework. Note that we assume that the content-based core (in this case Siena or Elvin) provide event ordering guarantees.

Notification and protocol models. The Pull notification delivery policy requires subscribers to periodically poll (or inquire) the server for new events matching their subscriptions. For such, messages need to be stored in the server side. In the YANCEES architecture, this mechanism is provided by a combination of notification and protocol model extensions. First, in the notification model, the <*pull*> language extension is defined; then a pull plug-in is implemented. Instead of sending the notifications di-

rectly to the subscribers, the pull plug-in stores the events in a temporary repository, implemented as a JDBC connector static plug-in. In order to allow end users to collect the persistent events, a poll protocol plug-in is also defined.

Event model. Finally, the event model also needs to be extended to support event typing. This is necessary since the existing pub/sub core performs content-based filtering and uses generic tuple-based events. CASSIUS events are marked with a special attribute indicating its type. In YANCEES, this functionality is implemented by a type checking input filter. Whenever this filter detects this special attribute, the event attribute names and types are checked for correctness, according to the CASSIUS template format.

Protocol model. Finally, 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 [17]. In the YANCEES framework, the CASSIUS functionality is implemented by the use of a persistency filter, the hierarchy browser, event source registry and authentication protocol plug-ins, which interact with the JDBC connector that intermediates 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.

4.2 Evaluation Case Study 2: Extending and configuring YANCEES to support a security visualization application

In this section, we present some of our experience in using YANCEES to achieve most of the requirements presented in section 2. (We are still researching appropriate criteria for usability). In this case study, we demonstrate YANCEES extensibility, configurability, interoperability and reuse in an application monitoring scenario. In this scenario, an enhanced version of Vavoom software visualization tool [8] was extended to use YANCEES as the basic communication infrastructure. Vavoom is a visual java virtual machine that allows the monitoring and visualization of java programs executions. The visualizations are based on runtime program execution events such as object creation, method invocations, variable changes and so on. Vavoom was extended to provide a security visualization tool that displays information about the network activity of monitored applications.
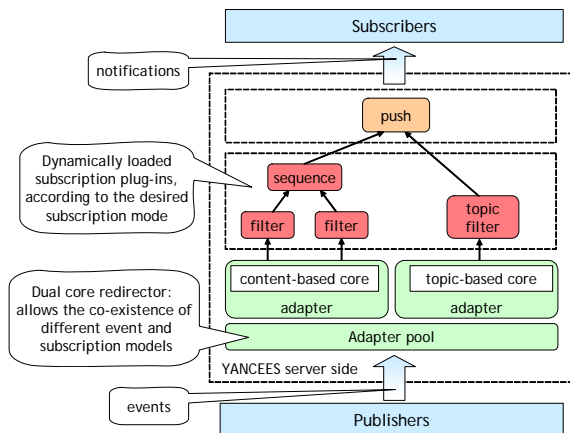

Figure 5. YANCEES configuration in supporting the security visualization scenario.

In this application, YANCEES plays two roles: In one hand, it is used to route events to program visualizations that display the current execution of the application. This requires fast routing of events and no content-based filtering; On the other hand, it also needs to provide content-based event filtering and sequence detection capabilities for a network activity visualization that displays and monitors the activity of current socket connections the application may open. The Vavoom class loader produces Java bytecode-level events with information, including object creation and destruction, method invocation and exceptions. This information, even though sufficient for Vavoom program visualizations, is insufficient for the security visualization tool. The visualization uses network-related information such as method invocations, with certain parameters, in a certain order. Hence these low-level events, produced by the class loader, must be filtered, combined and generalized to provide higher-level network activity information to the security visualization. A summary of the extensions and the YANCEES configuration for this application is presented in Figure 5.

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

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

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

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

5. RELATED WORK

The idea of modularizing and customizing different aspects of a publish/subscribe infrastructure is present in other systems at different degrees. Existing infrastructures however, are domain-specific versatile, limiting their flexibility to the exact amount demanded by the application domain they are designed to support, thus focusing on a subset of the design dimensions proposed in section 1.1.

The Modular Event System [11], for example, focuses on event transformations and interoperability, which is achieved by the modularization of publish/subscribe concerns in terms of components called scopes that can be statically composed to implement different infrastructure functionalities and policies under a standardized publish/subscribe API. Current implementation is based on Siena, and borrows from it its event model. There is no support for dynamic variability and for protocols. The subscription language is also fixed.

In another example, Shen and Sun propose a flexible notification framework (or FNF for short) [26] that support different requirements of collaborative applications. This is accomplished by the use of programmable message queues, where different ingoing and ongoing notification mechanisms can be installed. It allows the manipulation of incoming and outgoing event queues by controlling their event granularity and event forwarding frequency. It also allows the definitions of transformations of notifications (or events), for the implementation of application-specific concurrency control mechanisms. The FNF event model is topic-based; the tool is tailored for synchronous event-based groupware communication and the programming of the system is preformed by manually composing the message queues. There are no explicit subscription or notification languages.

FULCRUM [1] is a publish/subscribe system designed to support context-aware applications. It applies an open implementation strategy, allowing the programming of different context-aware queries involving real-world aspects such as device (publishers and subscribers) physical distances. FULCRUM focus on subscription extensibility and programmability, and do not allow the configuration and programming of other aspects of the infrastructure.

The ADEES (Adaptable and Extensible Event Service) [30] is a client-side framework that allows the definition of different subscription and notification strategies. It supports different sets of notification and subscription operations, expressed in a meta-model (language). Operations are implemented by different components which can be composed to perform different transformations over the events. The infrastructure is implemented as an event processing layer on top of CORBA-NS. It provides a certain degree of client-side extensibility, programmability and configurability of the subscription and notification languages. However, other aspects of the publish/subscribe infrastructure such as server-side extensibility and configurability, support for different pub/sub cores, event models and protocols are not addressed.

FACET [16] is an extensible and configurable implementation of the CORBA Event Service. The extensibility and configurability of features are implemented using Aspect Oriented Programming [9], which allows the weaving of different features in the middleware. It was initially designed to provide specific configurations that can run on restricted conditions of embedded systems, and can support the real-time requirements of specific applications. The use of Aspects as the only way to compose both functional and nonfunctional features demands mechanisms for consistency checking. The model is too tight to the CORBA-ES standard, borrowing from it its subscription, notification and event models.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented YANCEES, an infrastructure for developing application-specific publish/subscribe infrastructures. YANCEES achieves this goal through the combined use of extensible languages and plug-in oriented software development, strategically layered upon the main design dimensions of a publish/subscribe infrastructure. This approach provides configurability, programmability, extensibility and reuse of both plug-ins and existing publish/subscribe infrastructures.

Theoretically, we show that publish/subscribe infrastructures can be decomposed in sets of components that follow an extended version of the Rosenblum and Wolf [24] design framework. This extended version makes explicit the interaction and infrastructure aspect of each dimension, and supports a new protocol dimension that captures interactions with the publish/subscribe system other than the common publication and subscription of events. Practically, we show how these dimensions can be extended by the dynamic combination of plug-ins, filters, adapters and extensible languages by describing two use cases where YANCEES was used to provide application-specific functionality. These two contributions combined provide a novel approach to cope with the versatility demanded by different applications.

The use of a single extensible infrastructure has many benefits. It provides a common model with which customized publish/subscribe infrastructures can be build. This model also allows the extension of existing publish/subscribe infrastructures, coping with interoperability. Moreover, it improves the reuse of existing components, speeding up the development of new solutions.

These benefits come through a price. As in any framework, its initial construction demands some effort in generalization and domain analysis; after that, a learning curve exists in order to understand its concepts and extension points. Another cost is associated to performance overhead. Preliminary performance tests with our prototype of the framework showed an increase in the routing latency of the events [27]. This delay in performance, however, is compensated by the gain in configurability and extensibility, and is amortized by the high-throughput and multithread support provided in the current prototype implementation.

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

The implementation of crosscutting functionality such as security is still not completely addressed in the framework. The YANCEES platform, however, provides a testbed where different approaches for the implementation of non-functional requirements

can be tested and further exploited. In special, we are studying the use of Aspect-Oriented Programming, in addressing those issues.

7. ACKNOWLEDGEMENTS

8. REFERENCES

1. Boyer, R.T. and Griswold, W.G. Fulcrum – An Open-Implementation Approach to Context-Aware Publish/Subscribe, UCSD, San Diego, 2004.

2. Carzaniga, A., Rosenblum, D.S. and Wolf, A.L. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, *19* (3). 332-383.

3. Chatley, R., Eisenbach, S. and Magee, J. Painless Plugins. Technical Report - http://www.doc.ic.ac.uk/~rbc/writings/pp.pdf, Imperial College London, London, 2003.

4. Cugola, G., Nitto, E.D. and Fuggetta, A. The Jedi Event-Based Infrastructure and Its Application on the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, *27* (9). 827-849.

5. DePaula, R., Ding, X., Dourish, P., Nies, K., Pillet, B., Redmiles, D., Ren, J., Rode, J. and Silva Filho, R. In the Eye of the Beholder: A Visualization-based Approach to Information System Security. *To appear in The International Journal of Human-Computer Studies (IJHCS) Special Issue on HCI Research in Privacy and Security*.

6. DePaula, R., Ding, X., Dourish, P., Nies, K., Pillet, B., Redmiles, D., Ren, J., Rode, J. and Silva Filho, R., Two Experiences Designing for Effective Security. in *Symposium On Usable Privacy and Security (SOUPS 2005)*, (Pittsburgh, PA, 2005).

7. Dourish, P. and Bly, S., Portholes: Supporting Distributed Awareness in a Collaborative Work Group. in *ACM Conference on Human Factors in Computing Systems (CHI '92)*, (Monterey, California, USA, 1992), ACM Press, 541-547.

8. Dourish, P. and Byttner, J., A Visual Virtual Machine for Java Programs: Exploration and Early Experiences. in *ICDMS Workshop on Visual Computing*, (Redwood City, CA, 2002).

9. Elrad, T., Filman, R.E. and Bader, A. Aspect-oriented programming: Introduction. *Communications of the ACM*, *44* (10). 29-32.

10. Emmerich, W. Software Engineering and Middleware: A Roadmap. in Finkelstein, A. ed. *The Future of Software Engineering*, ACM Press, 2000.

11. Fiege, L., Mühl, G., C., F. and Gärtner Modular event-based systems. *The Knowledge Engineering Review*, *17* (4). 359 - 388.

12. Fitzpatrick, G., Mansfield, T., Arnold, D., Phelps, T., Segall, B. and Kaplan, S., Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin. in *European Conference on Computer Supported Cooperative Work (ECSCW '99)*, (Copenhagen, Denmark, 1999), Kluwer, 431-451.

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

14. Gruber, R.E., Krishnamurthy, B. and Panagos, E., The Architecture of the READY Event Notification Service. in *ICDCS Workshop on Electronic Commerce and Web-Based Applications*, (Austin, TX, USA, 1999).

15. Hilbert, D. and Redmiles, D., An Approach to Large-scale Collection of Application Usage Data over the Internet. in *20th International Conference on Software Engineering (ICSE '98)*, (Kyoto, Japan, 1998), IEEE Computer Society Press, 136-145.

16. Hunleth, F. and Cytron, R.K., Footprint and feature management using aspect-oriented programming techniques. in *Joint Conference on Languages, Compilers and Tools for Embedded Systems*, (Berlin, Germany, 2002), ACM Press, 38-45.

17. Kantor, M. and Redmiles, D., Creating an Infrastructure for Ubiquitous Awareness. in *Eighth IFIP TC 13 Conference on Human-Computer Interaction (INTERACT 2001)*, (Tokyo, Japan, 2001), 431-438.

18. Krueger, C.W. Software Reuse. *ACM Computing Surveys*, *24* (3). 131-184.

19. Lövstrand, L., Being Selectively Aware with the Khronika System. in *European Conference on Computer Supported Cooperative Work (ECSCW '91)*, (Amsterdam, The Netherlands, 1991).

20. Mayer, J., Melzer, I. and Schweiggert, F. Lightweight Plug-In-Based Application Development. in M. Aksit, M.M., R. Unland ed. *Lecture Notes in Computer Science*, Springer-Verlag Heidelberg, 2003, 87 - 102.

21. Nielsen, J. What is Usability? in Nielsen, J. ed. *Usability Engineering (Chapter 2)*, Morgan Kaufman, 1993, 23-48.

22. OMG. Notification Service Specification v1.0.1, Object Management Group, 2002.

23. Parnas, D.L., Designing software for ease of extension and contraction. in *3rd international conference on Software engineering*, (Atlanta, Georgia, USA, 1978), IEEE Press, 264 - 277.

24. Rosenblum, D.S. and Wolf, A.L., A Design Framework for Internet-Scale Event Observation and Notification. in *6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (Zurich, Switzerland, 1997), Springer-Verlag, 344-360.

25. Sarma, A., Noroozi, Z. and van der Hoek, A., Palantír: Raising Awareness among Configuration Management Workspaces. in *Twenty-fifth International Conference on Software Engineering*, (Portland, Oregon, 2003), 444-453.

26. Shen, H. and Sun, C., Flexible notification for collaborative systems. in *ACM conference on Computer supported cooperative work (CSCW'02)*, (New Orleans, Louisiana, USA, 2002), ACM, 77-86.

27. Silva Filho, R.S., De Souza, C.R.B. and Redmiles, D.F. Design and Experiments with YANCEES, a Versatile Publish-Subscribe Service - TR-UCI-ISR-04-1, University of California, Irvine, Irvine, CA, 2004.

28. Silva Filho, R.S., de Souza, C.R.B. and Redmiles, D.F., The Design of a Configurable, Extensible and Dynamic Notification Service. in *International Workshop on Distributed Event Systems (DEBS'03)*, (San Diego, CA, 2003), 1-8.

29. SUN. Java Message Service API, SUN, 2003.

30. Vargas-Solar, G. and Collet, C., ADEES: An Adaptable and Extensible Event Based Infrastructure. in *13th International Conference, DEXA 2002 Aix-en-Provence*, (2002).

31. Wilson, G.V. Extensible programming for the 21st century *ACM Queue*, 2004, 48-57.