Preserving Versatility in Event-Based Middleware

Roberto S. Silva Filho and David F. Redmiles
Department of Informatics
School of Information and Computer Science
and Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425 USA
{rsilvafi, redmiles}@ics.uci.edu

ABSTRACT
Existing commercial and research event-based middleware are limited in their ability to evolve in order to support the requirements of novel applications. As a result, new infrastructures are being proposed to attend to widening scope and changing demands. In this paper, we define the concept of versatility generally and survey existing approaches that can be used to develop and preserve versatility in middleware. We discuss our research in designing, implementing, and preserving versatility in event-based middleware using YANCEES, a versatile event notification service we are developing. The design of the system is briefly presented and the lessons learned discussed.

INTRODUCTION

According to the Cambridge Advanced Learner's Dictionary, versatility is the ability "to change easily from one activity to another" or to be "able to be used for many different purposes." In the context of software engineering, versatility can be defined as the ability of a computational system to serve multiple purposes or to accommodate the requirements of different use situations. In this article, we describe current technologies and approaches for providing versatility in software infrastructures. Specifically, we discuss our experiences providing and maintaining versatility in event-based publish/subscribe middleware, including a case study of a more general approach we have been developing.

Middleware refers to the software layer between applications and the network protocols and supports software engineers in developing distributed applications. Historically, middleware has been used to address issues related to heterogeneity, communication, and distribution of components, relieving software engineers of the burden of solving low-level, network issues, such as concurrency control, transaction management, distributed object location, and communication, among others. Thus, middleware allows software engineers to focus on the actual application requirements [1]. Because of these advantages, middleware has become very popular. In fact, in recent years, standardized solutions such as OMG CORBA (Common Object Request Broker Architecture) and SUN JMS (Java Message Service) along with their many implementations have been used in the development of a large spectrum of applications. While CORBA is a distributed implementation of the remote method invocation paradigm, JMS and their implementations are examples of message-oriented middleware (MOM), which integrate components in a distributed system through the exchange of asynchronous messages.

Our team at the University of California Institute for Software Research has been researching the design and development of collaborative software engineering environments, which generally require the integration and coordination of different components such as editors, document repositories, awareness tools, and application monitors, among others. In such environments, the use of the publish/subscribe architectural style copes with the scalability, dynamism, and loose coupling requirements and allows the integration of the heterogeneous components. The publish/subscribe architectural style is usually implemented through notification services (or servers). Notification services mediate the communication between event sources (information producers) and their interested parties (event consumers). Event sources publish information in the form of events. Event consumers express interest in subsets of events using subscriptions, which generally are logical expressions specifying the content and order of events. Notification services ensure the events of interest are passed to the appropriate event consumers.

When surveying the existing standards and infrastructures for implementing publish/subscribe architectures, we did not find a single solution that simultaneously provided all the services required in our research. We required a group communication infrastructure and a loosely coupled integration mechanism that could support and integrate application components in various domains, including groupware, awareness, software and user monitoring, and testing. We encountered various limitations. Content-based infrastructures such as Siena [2] did not support event delivery policies such as pull, nor some event services, such as persistency, demanded by awareness applications [3]. The CORBA notification service (CORBA-NS), although it supported many needed features, such as persistency of events and pull delivery mechanisms, did not have support for mobility and intermittent connections. Standardized message oriented middleware, such as JMS implementations, did not provide high-level event processing, such as sequence detection, demanded by user monitoring tools (event sequence detection, rules and event abstraction). In fact the existence of so many different commercial and research publish/subscribe infrastructures confirmed our observation that in spite of the availability of standardized solutions such as CORBA-NS or JMS, new notification servers continue to be developed to address the needs of novel applications such as Internet-scale event routing, mobile applications, inter-process communication, groupware and others.

The proliferation of specialized solutions reveals limitations on the way publish/subscribe architectures are designed and implemented. 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, which hinders the addition or customization of new application services. For instance, CORBA-NS does not support mobility protocols. Such an addition would require changing the publish/subscribe service source code or even aspects of the client application. Third, with rare exceptions such as the READY [4] (a CORBA compliant notification service), current solutions are not configurable with respect to the place where event processing happens in a distributed setting. For instance, some applications such as software monitoring [5], require the execution of event processing on the application side where the events are collected, whereas applications running on mobile devices may need a restricted set of services and components. Finally, with the exception of a few research prototypes, none of existing event-based middleware approaches support the selection and customization of features that the middleware should provide.

DEFINING VERSATILITY

In light of the above discussion, we proceeded to research ways of providing and maintaining good software engineering qualities. We adopted the term *versatility* in order to embrace an extensive set of qualities. Moreover, we sought a new term that could imply that that these qualities 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 middleware and publish/subscribe architectures, versatility comprises the following requirements.

Extensibility is the ability to augment middleware with new functionality. The most common way to extend a publish/subscribe infrastructure is to directly change its source code or to build in the required features as part of the client application. Aside from being cumbersome, this approach usually incurs extra delays in the overall application development. More importantly, the lack of extension mechanisms usually results in ad-hoc extensions that cause architectural drifting and non-interoperable solutions.

Functional Configurability is the ability to combine and select different functionality corresponding to different application needs or heterogeneous hardware and software constraints. Note also that some interdependence may exist between the functionalities provided. For example, in order to support pull event delivery, a notification service usually needs to provide persistency. Being able to represent functional dependencies is also important in fostering reuse of existing solutions.

Distribution Configurability defines the selection of the place to perform event processing, whether in the publisher (producer), the event router (notification service), or in the subscriber (consumer). This requirement is especially important for instance for mobile applications, which may not be continuously in contact with a central server or for monitoring tools where part of the processing is performed in the event producers [5], a way to minimize network traffic.

Reuse supports common requirements by referring to existing designs and implementations of services and components. Examples of the most common reusable features are content-based routing, event representation, and push or pull notification models and mechanisms. More advanced, though also commonly required features include event correlation, abstraction and persistency, among others.

Usability refers to the ease with which software engineers apply an infrastructure as well as the usefulness of that infrastructure's functionality. It also refers to the ease and functionality the infrastructure provides end users with in interactive applications. For instance, usability applies to the experience software engineers have defining new subscription languages and extensions as well as the experience end users have with editing and changing specific subscriptions, possibly through graphical user interfaces.

Besides the above qualities of versatility, publish/subscribe infrastructures need to support the essential middleware requirements of scalability, interoperability, heterogeneity, network communication and coordination [1]. Moreover, the versatility qualities presented above must not interfere with those essential requirements. In our work, we are especially interested in two of those requirements:

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

Scalability is the ability to support design and implementation choices required to cope with issues of magnitude, issues of quantity and size. For instance, scale may refer to the number of nodes in a distributed system or to the size and capability of hardware devices. Internet-scale notification services must deal with different timing, quality of service, delays, and other issues that are a consequence of the large number of publishers and subscribers of the system [6]. Scalability also requires integration and execution on different hardware platforms such as mobile devices, desktop computers and rack mounted systems (servers).

COMMON BUT LIMITED APPROACHES TO VERSATILITY

Versatility is not usually designed into or implemented in current publish/subscribe infrastructures. As a result, many workarounds are employed. Most commonly, missing features are added into the middleware by application programmers or accommodation is made in the application itself. In this situation, long-term maintenance is not always the highest priority. Consequently, maintenance and further evolution come at a high cost. Below are some approaches to middleware for publish/subscribe architectures that are common, but in a sense sidestep the issue of versatility.

One-size-fits-all

The most obvious approach to provide versatility is to provide the most comprehensive set of features imaginable. This strategy is adopted by standard implementations such as CORBA-NS and their extensions, such as READY. The CORBA-NS, for example, is an extension to the CORBA event service (CORBA-ES) that allows the definition and management of different event channels between CORBA distributed objects. Events can be typed or un-typed, persistent or non-persistent. The subscription language permits the definition of event sequence detection expressions with content-based filtering. The event delivery and observation can be performed using pull or push approaches. Secure channels can be established between publishers and subscribers; and scalability is addressed using federation of servers. Some implementations also provide event persistency. This approach, even though effective suffers from two fundamental problems: (1) the implementation cannot be easily extended to support new features (consider mobility, for example which requires support for publishers and subscribers temporary disconnection and migration); (2) nor the set of functionality provided by the service can be reduced to accommodate resource-constrained devices, such as PDAs or embedded systems.

Application-specific infrastructures

Another commonly used approach has been to develop application specific infrastructures, providing the characteristics demanded by each application domain. For example, CASSIUS [7], an awareness-centric notification service, was specifically designed with services to support those kinds of applications, providing event source hierarchy representation and discovery; graphical subscriptions, as well as event persistency and notifications summarization. This approach, however, still suffers from the same extensibility and configurability problems of one-size fits all solutions, even though they are better fit to the applications they serve. An undesired side effect of the adoption of many application-specific implementations is their lack of interoperability: different applications use different services that not necessarily communicate with each another.

Minimal core

Due to the Internet-scale requirements of current applications, content-based systems such as Siena [2] and Elvin [8] have became popular in the academia and industry. Through the federation of servers and a smart schema of subscription covering, they scale to Internet proportions. Also, by supporting event content-based filtering, they provide a very flexible subscription model, based on a generic event representation. Scalability, however, limits the subscription language expressiveness [6]. As the two approaches above, their lack of design for extensibility and reuse makes their functional extension and customization process a challenging task. Not having been designed with extensibility in mind, the addition of services such as event persistency, pull notification policy and mobile applications support protocols, for example, remains as sets of features to be implemented by the application developers [3].

ADVANCED APPROACHES TO VERSATILITY

In the last several years, advanced techniques and approaches have been developed which enable the development of middleware infrastructures with varying degrees of versatility. Those systems rely on the widespread adoption of object-oriented techniques and approaches such as object-oriented frameworks and design patters, as well as relatively new approaches such as computational reflection, aspect-oriented programming and component-based software architectures. In this section, we briefly discuss these approaches, their benefits and limitations, and some examples on how can they be applied to design and implement more versatile middleware.

Software patterns and frameworks

Software Frameworks are skeletal groups of software modules that can be tailored for building domain-specific applications. They provide reuse in the form of pre-programmed logic and adaptation points. They are usually implemented in object-oriented languages, being described in terms of concrete and abstract classes, which together specify the way instances of those classes collaborate [9]. As studied by Roberts and Johnson, the use of frameworks can reduce the cost of developing an application by an order of magnitude since it promotes the reuse of both design and code. Moreover, they have been adopted in a large set of applications and, for being built upon existing object-oriented programming languages and techniques, they do not require new technology [10].

Recently, frameworks have been used in the development of configurable middleware, as the example of the TAO ORB [11] and the Apache web server. The TAO ORB implements a CORBA ORB as an extensible framework. The system is modeled in terms of its basic components, allowing the static configuration of services and the runtime change of its strategic components. TAO can be configured to cope with different real-time constraints of applications by selecting the appropriate implementation of each component of the ORB. It also allows the definition of configurations where only necessary components are present, which addresses small footprint requirements of mobile devices or special real-time constraints. In another example, the Apache web uses a pluggable architecture where modules providing different functionality can be added. These modules or plug-ins can be installed with the help of hooks and an internal API, over the different stages (request reception, request translation, authentication, resource handling (using MIME types), response generation, logging, response) of the internal dataflow-based architecture of the HTTP server. Apache supports plug-ins and extensions that can handle different protocols such as WEBDAV and SSL, and externally-invoked applications such as CGI scripts.

Reflexive middleware

Computational reflection has been employed in designing configurable and open middleware. An example is Open ORB [12]. A reflective system is one that is capable of reasoning about itself. This implies that the system has some representation of itself in terms of its runtime programming structures. Reflection also provides access to the basic execution mechanisms of the system. Through the Meta Object Protocol (MOP), programs can intercept and adapt the middleware execution environment, including behavior such as message arrival, marshaling and un-marshaling of messages, thread creation and so on. Reflection is a powerful mechanism that allows the fine-grained extension of applications. However, it has some potential performance and integrity problems. Reflection itself does not impose restrictions on when and how to extend the system, every point in an application is a potential extension point. Hence it must be supplemented by architectural restrictions in the system and usually require the deep knowledge of the middleware internals.

Aspect-oriented approaches

Aspect-oriented programming (AOP) stems from the observation that in the development of applications, separate concerns such as security, logging, persistency and other "ilities" are hard to modularize. In programming paradigms such as object-oriented and functional programming, different, "cross-cutting" concerns are entangled in code across many modules. AOP aims to solve this problem by allowing the separation of individual concerns in different implementations (or aspects). Aspects are defined in an aspect language and are then interwoven in an application with the help of special compilers. AspectJ, for example, is an aspect weaver (or compiler) that allows the definition and weaving of aspects in Java programs.

AOP has been used to model and provide "ilities" to middleware. Filman et al. [13], for example, demonstrated how to extend ORBs with non-functional requirements such as security and fault-tolerance. In more recent work, Zhang and Jacobsen [14] showed how to extract, model and implement the different functionalities of an ORB as AOP cross-cutting concerns. Such separation allows the further rebuilding of an ORB with a custom set of such concerns.

As with the reflexive middleware, the AOP approach exposes the whole application, allowing the modification of every aspect of the system. Due to this white box approach, the extension of the system requires extra knowledge of the internals of the application. This approach contrasts with object-oriented frameworks that externalize only specific parts of the software and allows the extension of specific points of the application. Hence AOP and reflection strategies must rely on additional tools and methods to regulate the definition and combination of the different application concerns (or aspects) in a target application. Architectural constraints and dimensions must be observed in order to restrict and guide their use. For using a fine-grained approach (to the level of specific method invocations), performance degradation is an issue that must always be managed in such approaches. Despite their limitations, we find these two strategies promising for handling of non-functional (horizontal) application concerns and are currently investigating the use of such strategies in extending publish/subscribe infrastructures.

ARCHITECTURAL APPROACHES BEYOND MIDDLEWARE

Instead of relying on middleware services alone, compositional approaches are being developed that achieve some of the versatility qualities through the combination of different components in a distributed system. They rely on additional services and tools and are far from standardized at this moment.

Model-driven architectures

OMG promotes the use of Model Driven Architectures (MDA) as a way to decouple the application specification from its particular implementation on different middleware platforms. The approach maps platform independent models defined in UML to middleware-specific implementations. The idea is to better isolate the application specification from the specifics of different middleware, improving portability. The mapping from independent specification to middleware is automated, and performed with the help of platform-specific models.

Service-oriented architectures

Service-oriented architectures (SOA) are used to implement complex applications by the integration of distributed services. A service is an application externalized through standardized programmatic interfaces, a façade in the design patterns jargon, or a component in a software architecture point of view. Services hide the implementation of more complex systems behind well defined interfaces, which should be operated according to a richer semantic protocol. Hence, service-oriented approaches are not middleware extensibility approaches, but a composition and integration strategy that combines distributed applications. For such, they use standardized protocols, such as XML-RPC, SOAP, ODBC, and JMS.

PROVIDING AND MAINTAINING VERSATILITY

The approaches discussed above provide mechanisms that ease the selection and implementation of different aspects of versatility. However, simply the use of such approaches is still not sufficient to provide and maintain versatility. As with any software approach, developers must use the approaches in the context of each application domain and provide a usable adaptation interface. Specifically, the design of a versatile software system must follow those generalized steps:

1. **Initial Design / Domain analysis and understanding.** The basic model underlying the application must be first understood. The generalized support provided by publish/subscribe middleware consists of event creation, publication, routing (subscriptions), notification and delivery. Additionally, application-specific protocols need to be identified to handle different interactions or services.

2. **Refinement of the design.** Once the basic model is understood, developers need to identify the many functional and non-functional requirements that the infrastructure must provide. Those concerns may be implemented in one or more of these design dimensions (some use the terms, "vertical" and "horizontal" with dimensions). These define the general middleware process. For example, subscription-related functions such as event sequence detection, content-based filtering and routing are functional (or vertical) requirements associated to the subscription dimension alone; whereas concerns such as security, fault tolerance and reliability are "cross-cutting" (or horizontal) concerns that spam different dimensions of this model.

3. **The selection of the extensibility mechanism based on the problem domain characteristic**. Once the problem is framed, developers must select the proper approach and technology. For example, if the middleware must support extensive variability of crosscutting concerns, approaches as AOP or

computational reflection may be more appropriate. On the other hand, applications that require extensive variability in the middleware functional requirements, may adopt an object-oriented framework approach. Intermediate solutions may require the combination of two or more approaches.

4. **Building the software along the identified extensibility dimensions.** Finally, once identified the extensibility dimensions and their variability, developers should define principled rules, constraints, and extension and configuration mechanisms in order to allow software developers to adapt the infrastructure to the different applications.

YANCEES, A CASE STUDY

Based on the main steps described above, we implemented YANCEES (Yet Another Configurable and Extensible Event Service) notification service [15]. YANCEES is a versatile publish/subscribe infrastructure based on the idea of plug-ins and extensible languages. As such, we developed an object-oriented framework, described below.

Initial Design / Domain analysis and understanding. Rosemblum and Wolf described the main design dimensions of a publish/subscribe system in the form of a design framework [16].. In their 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.

Even though very complete, this framework does not address the new services a publish/subscribe system must support. In order to capture these new services, we introduce a new design model, the protocol. The protocol model captures all the different interactions with the publish/subscribe system that are not the common publishing and subscription of events. For example, the protocol model may be used to support mobility, providing hoarding primitives (move-in, move-out), can complement the push delivery mechanism allowing the collection of events stored in the server (collect events), can be used to express synchronization and communication messages used to federate different notification servers, and so on.

Design refinement. In a publish/subscribe system, the functional requirements (vertical concerns) are mainly related to the subscription and notification models. They specify different ways of correlating and delivering events. One key characteristic in the publish/subscribe paradigm, however, is the asymmetry of publishers and subscribers of information. Both publishers and subscribers may provide events and subscribe to this information in different ways. For such characteristic, the interaction with the system is asymmetric: whereas clients publish events using the service programmatic API in its programming language, subscriptions are expressed as logical expressions in the service specific subscription language. This important characteristic represents a challenging point in providing versatility to publish/subscribe systems: the addition of a new functionality in the notification, subscription and protocol models need to be defined by both: language extensions and plug-ins that implement those extensions.

Horizontal concerns such as security, scalability, fault tolerance and other “ilities” are usually handled by applying policies and constraints in the way the notification servers are composed, events are represented and subscriptions are preformed, hence they spam different models in our design.

Selection of the extensibility mechanism. For the implementation of YANCEES, we chose to provide an object-oriented framework implementation where new functionality can be provided in each one of the publish/subscribe models.

Building the software along the identified extensibility dimensions. An extension is implemented by the combined use of plug-ins and languages, installed along the basic publish/subscribe activities. In order to illustrate the role of plug-ins, languages and the design models, we briefly sketch the main YANCEES concerns in Figure 1 as follows. A detailed description of the system is provided in [15].

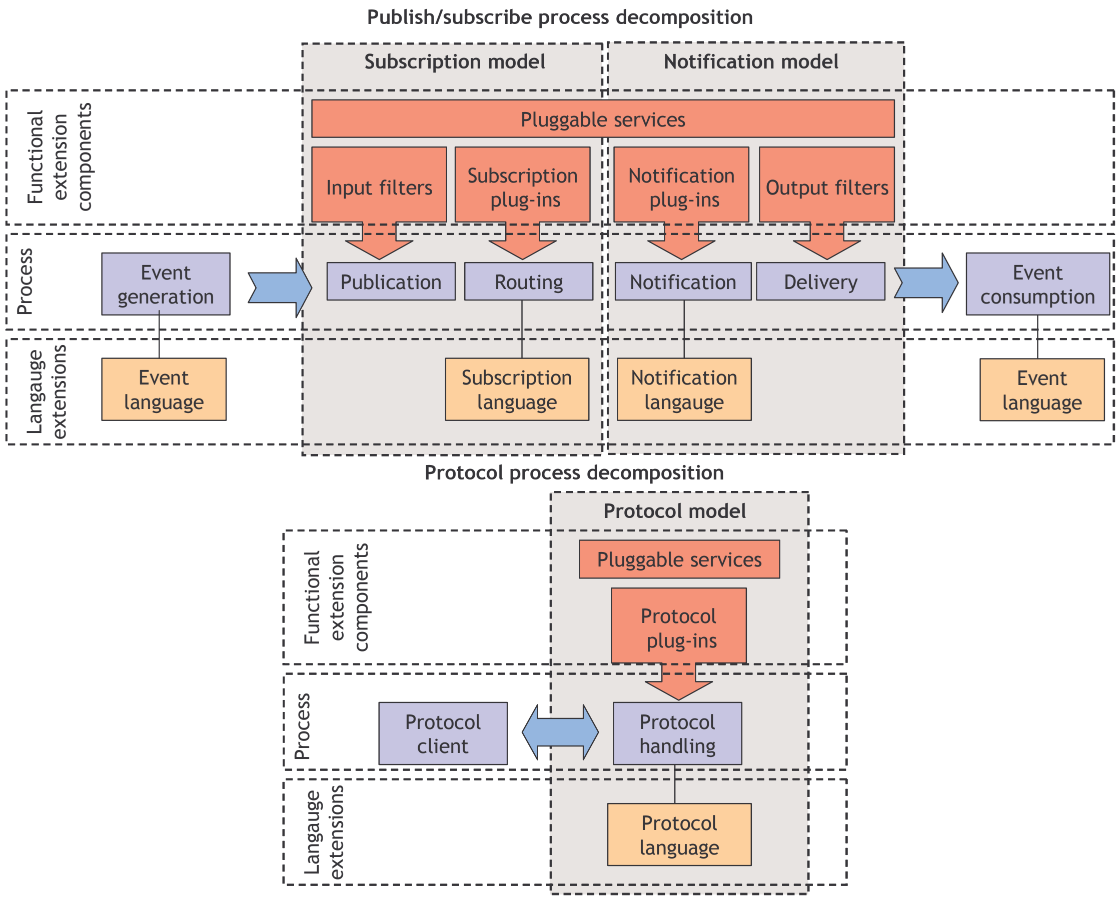

Figure 1 The generalized publish/subscribe process, the main YANCEES extensibility dimensions and their approach

Internally, to a service developer, the framework provides different extension points along the event, subscription, notification and protocol models where plug-ins, defined according to standard interfaces can be installed. It also allows the specialization and implementation of custom event dispatchers, services and filters. These internal components are combined in configurations, defined in XML files, used to bootstrap the framework. YANCEES also allows the dynamic configuration/installation of plug-ins, which facilitates the upgrade and evolution of the whole service.

With YANCEES, developers can specify configurations (sets of interdependent plug-ins), that together compose the functionality required by different applications. Plug-ins can be added to both client and server sides, which increases the model flexibility. Plug-ins are also dynamically allocated, on a per-subscription basis. Since subscriptions are recurring expressions on the content, order, timing and so on, plug-ins are usually combined to accommodate these complex expressions, thus promoting reuse and separation of concerns. Moreover, the publish/subscribe core can be replaced by existing event-based services such as Siena, Elvin, CORBA-NS or JMS, which copes with interoperability.

We summarize the main features of YANCEES and how it provides the versatility qualities previously presented, in Table 1 as follows.

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

Property	Approach
Extensibility	Achieved by the combination of extensible subscription, notification and protocol languages defined in XML and the implementation of those extensions by means of plug-ins Use of input/output filters and internal services;
Functional configurability	Supported by the ability to define different configurations: sets of plug-ins, filters, services and dispatcher components; Together with the ability to dynamically update or install new components.
Distribution configurability	Ability to specify client-side (publisher and subscriber) or server-side plug-ins, filters and services, distributing the event processing;
Reuse	Extensions are defined in an incremental way with plug-ins implementing specific language extensions; Existing plug-ins are dynamically composed, as they get used in logical subscription expressions Existing notification services such as CORBA and Siena can be wrapped and used as the core publish/subscribe service
Usability	Use of object-oriented frameworks which specify specific extension points, hiding internal program details, and provide standardized interfaces Extensible languages based on XML and extended by XML-Schema which can be parsed and integrated in GUI tools;
Interoperability	Allows the use of different simultaneous publish/subscribe cores, bridging heterogeneous notification services; YANCEES can be used as an abstraction layer on top of existing notification servers, allowing standard interaction with heterogeneous services, hiding different middleware idiosyncrasies from end users.
Scalability	The ability to use existing solutions such as Elvin and Siena, which are designed for scale, allows the reuse and incorporation of scalability in our model. Protocols can be implemented to distribute and integrate different YANCEES instances in a federated way.

EXPERIENCES AND LESSONS LEARNED

In the design of YANCEES, we opted to use a framework-based approach. Although this approach requires some extra effort in order to model a generic publish/subscribe system with its many extension points; to the developers' point of view, it represents a more natural and easy to learn approach. Foremost, plug-ins make it easy for developers to customize and understand the software. They leverage separation of concerns, providing reuse and customization. Moreover, the representation of the model in the form of a framework provides a principled (as opposed to ad-hoc) approach to manage the extensibility, configurability and evolution of the infrastructure, which is a key point in maintaining the versatility properties of the overall system. Finally, frameworks also hide internal application details, allowing their extenders to concentrate in extending the system to their particular needs with no need to understand the implementation details of the whole infrastructure.

A publish/subscribe system needs to cope with two complementary concerns: a programmatic model, which deals with the extension of functionality based on source code (plug-ins in YANCEES); and a language model, which deals with the event, subscription and notification languages representation. These two models need to be consistent with one another. In other words, the extension of the subscription language with a new function such as sequence detection needs to be matched by the implementation of that feature in the framework by the addition of a new plug-in. The use of functional plug-ins in this context not only provides dynamism but also improves reuse. Plug-ins can depend on one another and are allocated according to the functionality required by the each subscription provided to the service. This approach also allows the dynamic installation and upgrade of plug-ins, at runtime.

Usability is another key issue in maintaining versatility. Even though many approaches exist to extend and adapt software infrastructures, the choice of more usable and understandable approaches such as framework-based may considerably improve the overall maintainability of the system by the simple fact that it is closer to the everyday vocabulary of the programmers. The easier a code is to understand and extend, the easier it can preserve the versatility qualities provided by the approach.

No approach, however, can prevent developers from misusing it, e.g., by providing inefficient or incompatible extensions. This danger is more critical in less principled approaches such as AOP and Reflection. A common way to tackle this problem is the use of plug-in dependency checking, software benchmarking and automated tools that hide these issues from end users.

An ever present concern is performance. However, experiments with our prototype shows that the versatility achieved with the use of extensible languages and plug-ins provides only a slight degradation in performance compared to some monolithic approaches. Depending on overall application requirements, however, the performance drop is compensated by the versatility obtained. More details including performance are published elsewhere [15].

On the one hand, middleware needs to be transparent and hide the networking and communication details from the application developers. On the other hand, the increasing diversity of applications and their fast evolution has created a need for ways of adapting, extending and configuring middleware. As a consequence, it is essential to provide versatile middleware implementations that can accommodate those new requirements, allowing the evolution of middleware without loosing its properties.

REFERENCES

[1] W. Emmerich, "Software Engineering and Middleware: A Roadmap," in *The Future of Software Engineering*, A. Finkelstein, Ed.: ACM Press 2000, 2000.

[2] 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.

[3] C. R. B. d. Souza, S. D. Basaveswara, and D. F. Redmiles, "Using Event Notification Servers to Support Application Awareness," presented at IASTED International Conference on Software Engineering and Applications, Cambridge, MA, 2002.

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

[5] 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.

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

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

[8] 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.

[9] R. E. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object Oriented Programming - JOOP*, vol. 1, pp. 22-35, 1988.

[10] D. Roberts and R. Johnson, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks," in *Pattern Languages of Program Design 3*, A. Wesley, Ed., 1996.

[11] 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.

[12] F. M. Costa, G. S. Blair, and G. Coulson, "Experiments with an architecture for reflective middleware," *Integrated Computer-Aided Engineering Journal*, vol. 7, pp. 313-325, 2000.

[13] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden, "Inserting ilities by controlling communications," in *Communications of the ACM*, vol. 45, 2001, pp. 116-122.

[14] C. Zhang and H.-A. Jacobsen, "Refactoring middleware with aspects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 1058-1073, 2003.

[15] R. S. Silva-Filho, C. R. B. d. Souza, and D. F. Redmiles, "Design and Experiments with YANCEES, a Versatile Publish-Subscirbe Service," Institute for Software Research, Irvine, CA UCI-ISR-04-1, April 2004 2004.

[16] 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.