Lessons Learned Using YANCEES to Support Impromptu

Roberto Silveira Silva Filho, David F. Redmiles
Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3430 USA

{rsilvafi, Redmiles}@ics.uci.edu

Abstract

This paper describes the lessons learned in the use of an extensible publish/subscribe infrastructure in the supporting for a peer-to-peer collaborating system called Impromptu. We describe our experience using YANCEES to solve different problems faced during this project, and the lessons learned in the process. Some limitations and architectural issues are also discussed. They are mainly related to the role of dependencies in product line architectures.

1 Introduction and motivation

This paper describes the lessons learned in the use of an extensible publish/subscribe infrastructure in the supporting for a peer-to-peer collaborating system called Impromptu. We describe our experience using YANCEES to solve different problems faced during this project, and the lessons learned in the process.

2 Impromptu Scenario

Impromptu (DePaula, Ding et al. 2005) is an ad-hoc file sharing application. It allows users to share files in different access levels. Impromptu's interface gives users a clear representation of "who is around," what files are shared and in what degree, and what actions by other users are being taken at a given moment. In addition, the application allows users to easily configure the sharing levels of their files by directly moving them in and out from the Pie GUI. The "pie" designates the entire ad-hoc file-sharing group in which each slice corresponds to a single participant's shared area of the workspace. A participant's own slice is represented by the darker shaded slice. The interface is separated into multiple concentric regions representing different degrees of sharing. Files outside the circle are not shared at all, but available to the local user; files in the outer region are visible but not readable or writable to others; files in the next region are readable but not writable; in the next, readable and writable; and in the center, readable, writable, and available persistently.

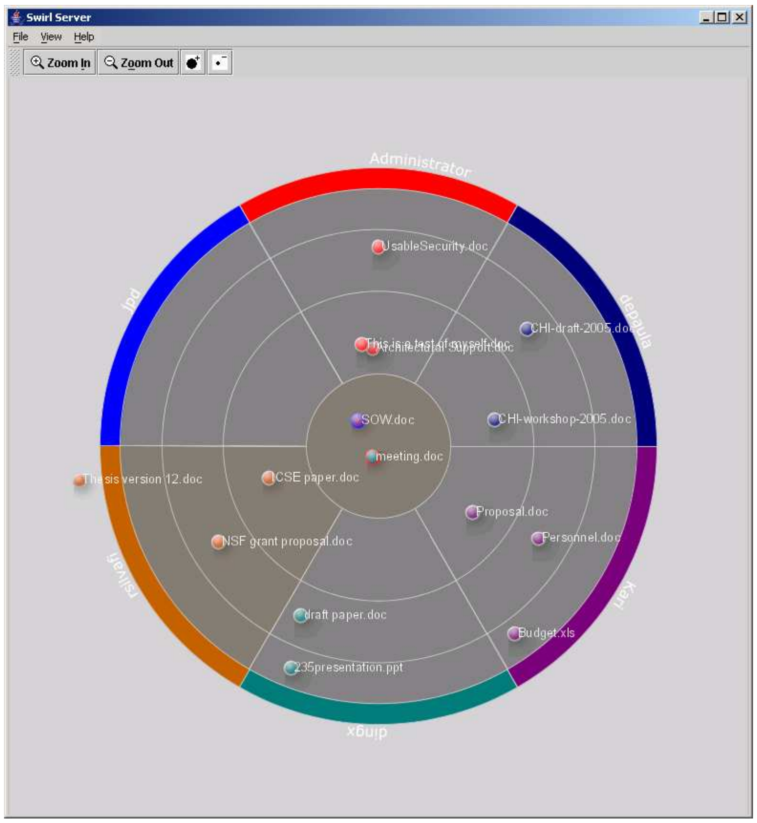

Figure 1 Impromptu interface - six-people collaboration scenario

Persistent access means that, even when someone leaves the session, his or her files remain accessible to others in the group; by default, files are non-persistent, meaning that when the user leaves the session, their files will disappear from others' interfaces. File access is managed by moving the files between the levels. People can only control the accessibility of their own files.

The dots that represent files do more than simply convey the position of an object in the Pie; they also represent activities over those files. Remote file accesses to local files cause a ring around the icons for the files to blink in colors that indicate the identity of the user accessing them. In so doing, it implements the concept of integration of action and configuration and of dynamic visualization of activity.

2.1.1 System Implementation

Internally, the Impromptu application consists of the following components: the graphical user interface, the Jetty web server, the Impromptu WebDAV filter, and the Slide WebDAV servlet that provides file read/write access to HTTP. The implementation also includes a secure WebDAV connector and the YANCEES event notification connector that implements a common event bus. The architecture is depicted in Figure 2. Jetty and Slide are external open source software components. The user interface component, the proxy component, the secure WebDAV connector, and the YANCEES connector (see above) are developed by us.

Jetty serves as a dynamic application server that allows an add-on component to decide what a response will be when Jetty receives a request. Slide is such an add-on component that provides WebDAV repository support (write access to web urls). WebDAV (Goland et al., 1999) is an HTTP extension that provides Internet-scale resource storage, retrieval, and modification capability. It is an open standard, easily available in different platforms, and is thus chosen as the foundation storage for the ad-hoc file sharing application.

Each participant stores his/her files in his/her own Slide server. However, this local storage is not directly seen by the participant. A participant only interacts with the Impromptu proxy server, using the Pie GUI depicted in Figure 1. The proxy provides an illusion of a unified, shared file storage workspace. When an Impromptu proxy receives a file operation request, it determines whether the request is directed at a local file or a remote file belonging to another participant. In the former case, it retrieves the file from the local Slide server, using a standard WebDAV request. In the latter case, it performs the operation against the remote Impromptu proxy, which will accomplish the operation using its own local Slide server. All the WebDAV access commands are logged and published as events to YANCEES. Impromptu uses then YANCEES, configured for a peer-to-peer setting, to maintain the client Pie views in sync by informing each client of events taking place on the others and to convey important access events to the GUI that informs end-users about the access of other peers in their files.

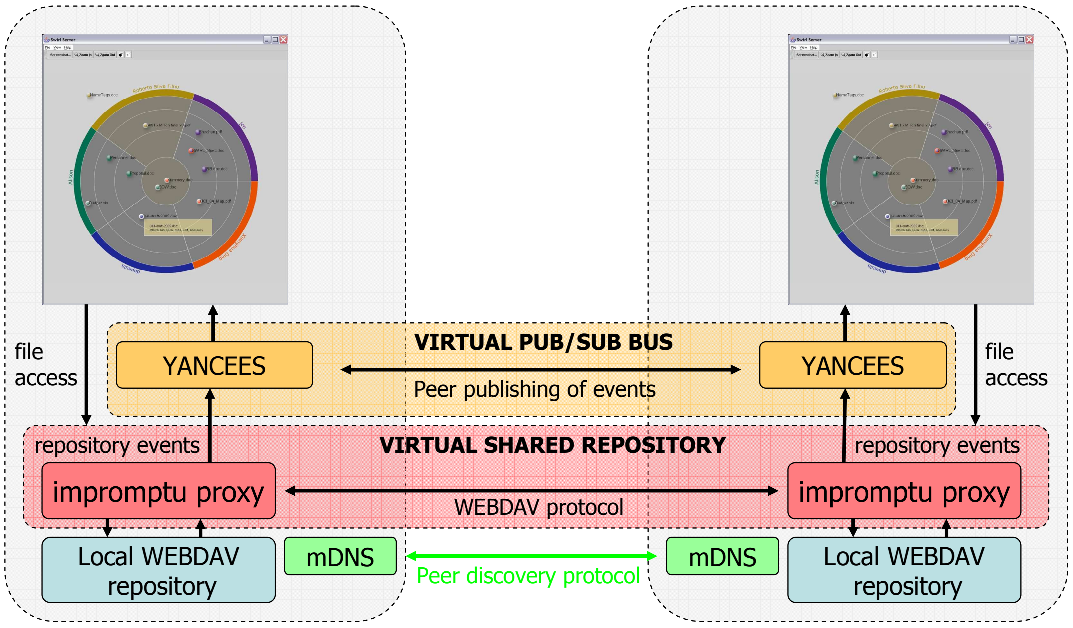

Figure 2 Impromptu Architecture

3 YANCEES framework

YANCEES (Silva Filho, de Souza et al. 2003; Silva Filho, De Souza et al. 2004) is a versatile notification service designed to be programmable, configurable and dynamic. The main motivation in the development of YANCEES is the observation that middleware infrastructures, in special publish-subscribe systems, have many commonalities. In spite of the proliferation of different infrastructures that implement this communication style, they address the same common problems. For example, all route events, all provide subscription (a way to express interest on the events) and notification (a way to be notified or receive those events). In spite of this commonality, pub/sub systems are implemented over and over each time a similar problem is found. The idea is then to achieve a higher degree of reuse, configurability and extensibility by implement a generic framework with which different publish/subscribe infrastructures can be devised.

Another observation is that specific applications demand different functionality from the pub/sub infrastructure. For example, mobile applications require roaming. Event-based monitors require filtering and sequence detection. Event-driven architectures as C2 require a simple event bus; Awareness applications require advertisement; Security, guaranteed delivery, total order and other requirements may also arrive. In other words, besides the basic functionality, additional features are usually necessary.

YANCEES aims at providing a common baseline with which different services can be implemented. It was initially designed as a way to allow the customization; extension and implementation of new functionality on top of existing publish/subscribe infrastructures, especially content-based notification servers such as Siena and Elvin, with special focus on collaborative software engineering applications. In fact, the system can be used as an event processing layer.

Internally, YANCEES uses a combination of plug-ins, extensible languages, open implementation and composition filters techniques to provide configurability, extensibility and programmability over the main design dimensions of a publish/subscribe system, including support for protocols. As depicted in Figure 3, YANCEES provides a bare-bones implementation on top of which plug-ins can be added. Plug-ins implement extensions in the subscription, notification and protocol languages. Besides plug-ins, filters can be used to intercept the publication and notification queues of events, performing event transformations, type checking, persistency or other actions. Static services can also be installed in order to support the implementation of plug-ins and filters. Finally, the system uses open implementation to allow the replacement of the event dispatcher with different event routing strategies, for example content-based, topic-based or record-based routers. Those components are managed and combined together with the help of a configuration language.

While input and output filters can be used to implement policies that apply to all the events, subscription filters can be defined to specific subscriber needs. For example, in Impromptu, input filters are used to eliminate unnecessary event repetitions, and to collect the events that must be routed to other peers in that application. As the same time, subscription filters are used to subscribe to specific events in the GUI.

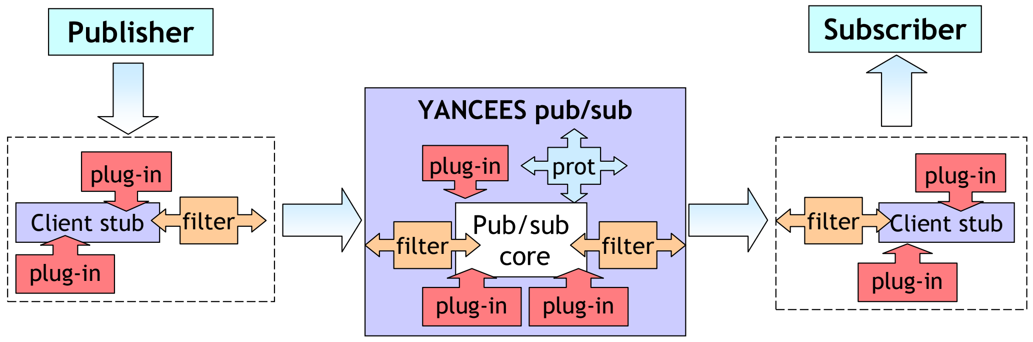

Figure 3 General Overview of YANCEES and its extensibility points

4 Supporting impromptu evolution with YANCEES

This section describes some of the problems and issues faced in the implementation of the peer-to-peer event bus that supports the impromptu application. It discusses the solutions provided to those problems using the extensibility and configurability of YANCEES. Those problems illustrate the evolution of features that middleware is required support on the support of an application. They show how application requirements evolve throughout the development of a scientific project, and how this translates in new features that the middleware need to provide in response to those requirements. If design for flexibility was not present in the infrastructure beforehand, the adaptation and change of the infrastructure would be much more costly that it was in our experiments. In fact, if traditional publish/subscribe infrastructures as Siena and Elvin were originally, used the implementation of those new features would be harder, and consequently more costly, than they actually were by the use of YANCEES.

4.1 Problem 1: Implementing a dynamic peer-to-peer bus

In the impromptu scenario, YANCEES was configured to implement a common event bus that connects all the peers of this application. Due to the peer-to-peer and ad-hoc characteristic of impromptu, the event bus has to include and or exclude new peers as they appear and leave the application.

A P2P bus is the one that grows and shrinks as new peers arrive and depart from the network. The main challenges involved are the implementation of protocols that allow the discovery of peers and management of the connections between peers. Another challenge is the fast routing of events required in this configuration. The solution to those two sub-problems are discussed as follows.

4.1.1 Sub-Problem (a): Peer discovery and ad-hoc networking

The challenge here is the discovery of peers in the network and the interconnection of those peers by means of an event bus, where events published in one peer are visible to all other peers in the application setting. Traditional content-based pub/sub servers as Siena (Carzaniga, Rosenblum et al. 2001) and Elvin (Fitzpatrick, Mansfield et al. 1999) allow the federation of servers. This federation, however, is static and needs to be performed at startup time and the modification of the federation protocol to support such dynamism. In YANCEES, however, the implementation of federation protocols can be implemented by the strategic addition of plug-ins and filters as follows.

4.1.2 Solution: protocol plug-ins

Through the use of a protocol plug-in to inter-connect servers and a service plug-in that implements the JmDNS protocol, we could configure YANCEES to dynamically discover and connect to other YANCEES peers. The PeerDiscovery YANCEES protocol plug-in uses existing ad-hoc JmDNS protocol to manage the arrival and departure of YANCEES peers in the network, interconnecting those peers as they arrive in the network. The federation strategy used in the case of Impromptu was the publication of events to all the peers. Other strategy would be the propagation of subscriptions but, since in the impromptu scenario, we were interested in the implementation of a distributed event bus, the latter was preferred.

With the peer discovery problem resolved, a publication filter installed in the input queue of YANCEES, forwards all the events published locally to the other peers through the federation protocol. Conversely, this protocol plug-in also receives events from other peers and publish them locally allowing them to be subscribed.

4.1.3 Sub-Problem (b): Fast routing of events

Due to the fast-forwarding requirement of the impromptu scenario, our initial though was to use Siena or Elvin as our pub/sub engine in the extensible YANCEES framework. YANCEES was them configured and instrumented to run in those two configurations. Our scalability tests, however, showed the inability of those two systems to route a large amount of events. Typical numbers were around 3000 events per second in both architectures. Our expectation that Elvin, being implemented in C, would perform better, were not true. Due to the relative general-purpose character of those systems, their use in such heavy traffic was not well supported

4.1.4 Solution: client-side event buffering (pipe and filter)

In order to outperform Siena and Elvin in those scenarios, two main extensions were implemented in YANCEES: the first was the use of client-side publication buffers; and the second was the implementation of a fast-switch core. The input buffers gather the events published at every 50 seconds and deliver then in a single RMI call to the notification service. The fast switcher performs topic-based routing only, allowing a faster delivery of those events to the subscribers. As a results, the throughput of the service increased reaching peaks of 10000 events per second.

4.2 Problem 2: Security policy: Preventing local events from being propagated to all the peers

In the impromptu scenario, YANCEES is used as an event bus that multicast interface events to all the peers in order to achieve interface synchronization, as well as a local notification service that monitors the events happening in the local Webdav repository. Part of this information should be published to all the peers in order to indicate, for example, that a user read a file in my repository. Part of this information, should not be multicasted, for example, the inclusion of a private file in the repository. In a traditional event bus, it is up to the subscribers to define which events they are interested on, or to the publishers to decide what events to publish. In our case, since the medium is shared, another solution needed to be implemented

4.2.1 Solution: client-side security filter (pipe and filter)

Filtering events in the BUS. In order to discriminate those events and prevent them to being sent to the other peers, a publish filter was defined in YANCEES. This filter decides whether to publish or not the events to the peers based on a special meta-tag provided in the event. If the event has a tag PUBLISH_TO_PEERS= true or no tag at all, it is automatically published to all the peers, otherwise, it is only published locally.

This strategy has the advantage of enforcing security policies ion the source, keeping the subscribers agnostic of their willingness to subscribe or not to an event. In other words, public events are always subscribable an it is up to the publishers to decide what to disclaim. It is also good to the publishers since the security can be enforced in the bus level.

This ability of filtering events in the source is not present in pub/sub infrastructures as Siena and Elvin

4.3 Problem 3 Eliminating repeated events in a specific time interval

It is usually the case that many repetitive events get published in the impromptu scenario when a file is successively read by a peer. This is a consequence of the HTTP protocol implementation that we are using, that issues many successive get calls to read a single file (multiple-block read). This kind of implementation represents an extra burden to the awareness visualization users that need to cope with this duplication of events in a short period of time.

4.3.1 Solution: 'within interval' filter (pipe and filter)

By defining an input filter in the server-side of YANCEES, repetitive events can be removed from the system, guaranteeing that only the first event of the desired kind, within the specified interval, should be published, whereas the other events are discarded.

5 Lessons learned

5.1 YANCEES implementation

Some design decisions in the implementation of YANCEES, in special, the selection of some design patterns, have the goal of managing some of the common problems in the of software product lines domain. Those problems include the management of dependencies between the components of the system, the insulation of concerns related to feature implementation from those components; configuration management and runtime activation. This section discusses some of the solutions employed in YANCEES in the handling of those issues.

5.1.1 The use of patterns in YANCEES

In the design of YANCEES, patterns such as filter, adapter, abstract factory, singleton and others have been used to provide important characteristics to the architecture such as extensibility and dynamism, and to address specific problems of software product lines such as activation dependencies and coupling. This section discusses the main use of design patterns in addressing those issues.

5.1.1.1 Decoupling runtime plug-in selection with factories

In YANCEES, factories are used to allocate plug-ins in the subscription language. Plug-ins are allocated to handle specific keywords in the subscription language, being assembled together in event processing trees.

5.1.1.2 Resolving dependencies with configuration manager

5.1.1.3 Building architectures with builder

5.1.1.4 Extending subscription languages with listener plug-ins

5.1.1.5 Processing events with input and output filters (pipe-and-filter)

5.1.1.6 Using existing pub/sub infrastructures with adapters

WE MUST HAVE EXAMPLES WHERE YANCEES WAS NOT SO SUCCESSFUL: IN THE SUBSCRIPTIONLANGUAGE!!!

5.1.2 Extensible languages and plug-ins in YANCEES

Extensible languages and plug-ins (Wilson 2004; Birsan 2005) provide flexibility to YANCEES, allowing the implementation of different

5.2 Using YANCEES: the role of dependencies between features

Dependencies play an important role in the design of product line architectures(Lee and Kang 2004), they espress the relations that exist between the main features of a product line. In a product line, features are implemented by different components that usually need to communicate to implement the desired functionality. Features are composed in order to implement the final product, in our case, the YANCEES final configuration, that provides the desirable set of services for the application. This approach, however, has some issues such as feature interaction (Cameron and Velthuijsen 1993), and hidden dependencies. This section discusses some of those problems found in the implementation of impromptu prototype.

5.2.1 Filters hidden inter-dependencies

One interesting limitation of the approach using patters is the management of complex inter-dependencies between the components that implement the system. Different dependencies can be found in product-line architectures such as YANCEEE (Lee and Kang 2004). In special, sequence dependencies exist in the use of input and output filters in our architecture. As an example, in our implementation, the combined use of a sequence detector input filter and a publish-to-peer input filter has some interesting dependency problems and implications.

The publish-to-peer filter redirects events from the input stream to all the peers in the infrastructure. It works together with the peer discovery protocol and peer publication plug-ins that implements peer discovery and the distributed event bus, respectively. This input filter collects all the events that must be published to the other peers in the application.

The sequence detection filter described in 4.3.1, removes repeated and unnecessary events, preventing them from being published to the subscribers. The combined use of those two input filters have an implicit sequence dependency. The sequence detector plug-in must come before the publish-to-peers filter in order to prevent repeated events from being sent to the other peers in the system. An out of order installation of such filters result in undesired behavior as the publication of repeated events to all the pees and the elimination of repetitions in the local host only. Those dependencies are not usually obvious for software developers, and may lead to unexpected behavior of the system.

This example also demonstrates the not always obvious impact that filters may have on one another. Since the output of a filter is the input of another filter, the order matters and their functionality may be incompatible with one another, hindering their reuse in certain conditions.

6 Discussion

The YANCEES strategy is to provide extension points representing the main design dimensions of Rosemblum and Wolf. Those dimensions are translated into plug-ins, filters and extensible languages that allow end-users to modify and extend the main aspects of a publish-subscribe infrastructure. Anticipation of changes by the externalization of those hooks is the goal.

In each of the examples provided in this report, the extension to the middleware was implemented by the insertion of one or more components that together provided the desired functionality. In some cases, we could modularize the feature in a single filter, as the case of the within interval filter. In other cases, we needed to introduce different components that work together in providing the required functionality, as the case of the dynamic peer-to-peer bus. In one case, however, the bufferization of the publish command required the change of the publish API (I could have done it by the insertion of a filter in the publish API and this would guarantee the modularity).

Some changes, however, cannot be easily modularized as described in this document. It is the case of the changing of the event representation. The changing of the event requires updating the whole components that handle with it. The only alternative in the OO world in this case is to keep the event representation encapsulated in an object that keeps the attribute/value pair interface. This may not be feasible with documents, but can be feasible with record-based approaches, or with object-based approaches if a reflective wrapper is used. Even though, those approaches address some of the problems, still, one cannot modularize some of the changes in a single object.

The solutions discussed here use pipe and filter, a design pattern way o implementing composition filters. Plug-ins inserted in the system work as interceptors and adapters, talking to each other through direct communication. Hence, the set of plug-ins working together implement a certain aspect. Those plug-ins, so far, can be componentized and ‘weaved’ in the architecture through the use of the architecture manager, that employs traditional design patterns to implement it. More advanced weaving mechanisms such as those provided by AOP could be used in this process, instead of more traditional hooks.

Plug-ins define a simple component model where each plug-in publishes its extension points keeping a fixed API that can be extended by others (by listeners and hooks). Plug-ins provide reusable code following explicit conventions to separate API from internal code. Each plug-in exposes an API that allow its use while protects its secret. This prevents, for example, extensions, and modification of the plug-ins internals.

Flexibility has its costs: planning ahead, development costs, and in our case, some delays associated to building the framework logic.

Acknowledgements

This research was supported by the U.S. National Science Foundation under grant numbers 0205724 and 0326105, and by the Intel Corporation

References

Birsan, D. (2005). On Plug-ins and Extensible Architectures. ACM Queue. **3:** 40-46.

Cameron, E. J. and H. Velthuijsen (1993). "Feature interactions in telecommunications systems." IEEE Communications Magazine **31**(8): 18-23.

Carzaniga, A., D. S. Rosenblum, et al. (2001). "Design and Evaluation of a Wide-Area Event Notification Service." ACM Transactions on Computer Systems **19**(3): 332-383.

DePaula, R., X. Ding, et al. (2005). "In the Eye of the Beholder: A Visualization-based Approach to Information System Security." International Journal of Human-Computer Studies - Special Issue on HCI Research in Privacy and Security **63**(1-2): 5-24.

Fitzpatrick, G., T. Mansfield, et al. (1999). Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin. European Conference on Computer Supported Cooperative Work (ECSCW '99), Copenhagen, Denmark, Kluwer.

Lee, K. and K. C. Kang (2004). "Feature Dependency Analysis for Product Line Component Design." Lecture Notes in Computer Science - 8th International Conference on Software Reuse, ICSR'04 **3107**: 69-85.

Silva Filho, R. S., C. R. B. de Souza, et al. (2003). The Design of a Configurable, Extensible and Dynamic Notification Service. International Workshop on Distributed Event Systems (DEBS'03), San Diego, CA.

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

Wilson, G. V. (2004). Extensible programming for the 21st century. ACM Queue. **2:** 48-57.