The Event-Based Paradigm in Collaborative Product Data Management: A Case Study

Mauro Caporuscio and Paola Inverardi
Dipartimento di Informatica
Università dell'Aquila
I-67010 L'Aquila, Italy
{caporusc, inverard}@di.univaq.it

Abstract

In this paper we discuss our ongoing experience in integrating a Product Data Management application, THINKTEAM*, and the* SIENA *Publish/Subscribe Middleware. This integration should support* THINKTEAM *users in managing the* lost-update *problem. This problem arises when a client performs a* check-out/modify/check-in *cycle on a document that may be used as reference copy by other clients. In this context* SIENA *should allow for disseminating alerts about the state of the document. The integration between* THINKTEAM*, and* SIENA *poses new requirements on how to manage events and subscriptions. The paper discusses a set of integration problems and their possible solutions.*

1 Introduction

Since the past few years industry has been facing the necessity to decrease its time-to-market. In fact, a design engineer usually spends up to 25-30% of his time for handling information, looking for it, retrieving it, waiting for copies of drawings, archiving new data, etc...

In order to speed up these activities computer systems have been developed to help improving the flow, quality and use of engineering information throughout a company. These systems, known as *Product Data Management* (PDM) [10], support the engineering processes management through the control of engineering data, of engineering activities, of engineering changes and of product configurations. That is, PDM systems maintain the control of data and distribute it to the people who need it when they need it.

The way PDM systems cope with this central role is by holding data (only once) in a secure repository where data integrity can be assured and all changes to data can be monitored, controlled and recorded. Due to the distributed nature of the system, all these actions need coordination and concurrency control. In fact, a usual problem that arises from this kind of systems is the so called "*lost-update*" problem. The lost-update problem (depicted in Figure 2) may occur when a user inserts, deletes, or changes something in the repository, and the effects of his work is nullified by another user.

While an automatic solution of this problem is not possible (as it is critically related to the type, nature and scope of the changes that will be performed on the document) a distributed notification facility would provide the mean to supply the actors with adequate information. In this context content-based Publish/Subscribe systems [6, 5] seem to be a well suited solution for disseminating information about operations made on documents of interest. Moreover, previous work that has been done in exploiting Publish/Subscribe capabilities within Computer Supported Collaborative Work (CSCW) [7, 3], demonstrates how event-based system can be used to build complex, distributed applications for data management.

In this paper we present ongoing work on using the *SIENA* Publish/Subscribe Middleware [4] within the THINKTEAM PDM [11] application. This work shows the use of an event-notification system in a real business-oriented application, THINKTEAM, and discusses some research issues raised from this experience.

The paper is organized as follows. Section 2 describes the THINKTEAM PDM application and introduces the above mentioned coordination problem. Section 3 outlines the integration of *SIENA* with THINKTEAM and discusses the problems, and solutions, that are posed by this integration. The last Section concludes and presents future work.

2 ThinkTeam Description

THINKTEAM is a Product Data Management (PDM)

application that provides Product Lifecycle Management (PLM) facilities [11]. THINKTEAM provides data (and document) management capabilities to deal with all product documentation: 3D models, 2D drawings, product specifications, analysis results, etc... Moreover, THINKTEAM allows multiple users to access to updated, released and ongoing work of product information.

Structurally, THINKTEAM is a three-tier application, which implements an information management system using an underlying DBMS for persistence and retrieval of the metadata (non-document data). The most typical in-

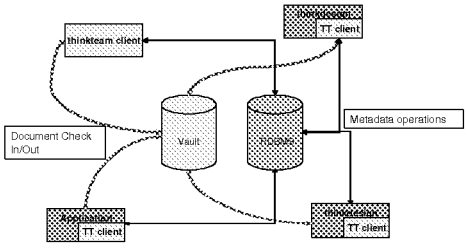

Figure 1. Thinkteam Network

stallation scenario is a network (as shown in Figure 1) where several distributed clients interact with one centralized RDBMS server and one more file server (called *vault*). In this environment, components resident on each client peer supply graphical interface, metadata management and integration services. Persistence services are achieved building on the characteristics of the RDBMS and file servers.

In this context a central role is played by the vault component. The vault is a file system-like repository that controls storage and retrieval of document data in PDM systems. The main functions of the vaulting system are:

1. Providing a single, secure and controlled storage environment where the documents controlled by the PDM system are managed.

2. Preventing inconsistent updates or changes to the document base, while still allowing the maximal access compatible with the business rules.

While the first point involves the low-layer implementation of the vaulting system, the second point concerns the following standard set of operations:

- **get**(*doc*): extracts a read-only copy of the document *doc* from the vault.

- **import**(*doc*): inserts a new document *doc* in the vault.

- **checkout**(*doc*): extracts from the vault a copy of the document *doc* for being modified; on a specified *doc* only one check-out at a time is possible.

- **checkin**(*doc*): replaces a modified document *doc* in the vault. Of course, *doc* must have previously been checked-out.

- **uncheckout**(*doc*): cancels the effects of a previous check-out on *doc*.

Due to the high number of possible clients there is a high degree of concurrency in vaulting operations. In order to maximize concurrency, THINKTEAM does not create exclusive locks for check-out operations. It is therefore possible for clients to perform a **get** operation on already checked out documents. This usually causes the so-called *lost-update*

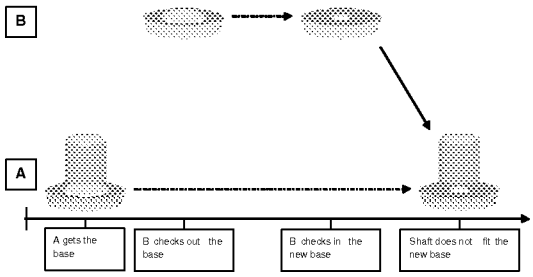

Figure 2. The lost-update problem

problem. The problem (as depicted in Figure 2) arises when one client performs a *check-out/modify/check-in* cycle on a document that may be used as reference copy by other clients.

Since an automatic solution of this conflict is not possible, an alternative is required. In this context dissemination of information about performed actions would allow for alerting interested clients. Of course, each user will be responsible of his own reactions.

In next Section we show the use of the SIENA Publish/Subscribe Middleware for disseminating information of interest.

3 Event-Notification Within ThinkTeam

As pointed out in the previous Section, an automatic solution of the conflict is not possible, because it is critically related to the type, nature and scope of the changes that will be performed on the document. A distributed dissemination of the information would provide the mean to supply the users with notices by:

1. Advising the user who checks out a document that there exist outstanding reference copies.

2. Notifying the copy holders upon check-out and check-in operation of the document.

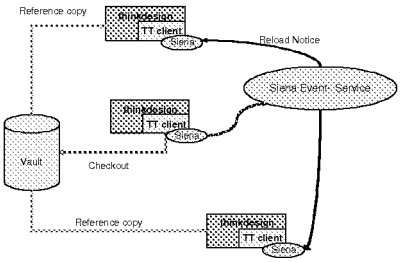

Figure 3. thinkteam and SIENA

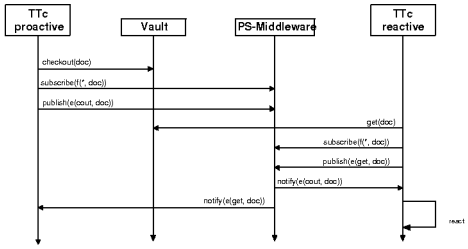

Figure 5. A Scenario concerning get and checkout operations

Figure 3 shows the new integrated software architecture. In this architecture two additional actors can be recognized: the *Siena Client* and the *event-service*. Each *Siena Client* can be both *publisher* and *subscriber*. Publishers emit events into the event-service which in turn delivers them to the interested subscribers.

Starting from the actions described above and the temporal constraints upon them we can draw the High-level Message Sequence Chart [8] (depicted in Figure 4) and, then derive a set of scenarios. Some of them describe critical use cases that require careful management. For example,

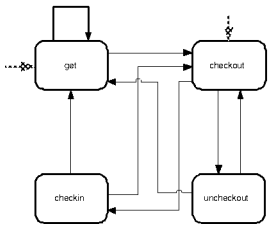

Figure 4. High-level MSC

Figure 5 shows the scenario involving a **get** and a **checkout** operations: *TTc-proactive* checks out a document doc, subscribes for information about it and, then publishes the associated event in order to inform other clients about its intents. After that, the *TTc-reactive* retrieves the same document from the vault, subscribes for actions involving it and, informs other clients about its action. At this time the event-service should inform *TTc-reactive* that there exists a client that is currently modifying such a document, and the *TTc-proactive* that there exists a client that has a reference-copy of such a document.

This scenario represents a case that may not be satisfied. In fact, due to the native semantics of SIENA (and of the event-based systems in general), it is not possible to deliver events to those clients that are not subscribed at the time of publications. This problem (that we refer as "Missed-events Problem"), already pointed out in previous work [1], can be solved in different ways that we discuss in Section 3.1.

Moreover, due to the relevance of the events sent, the notification system should be reliable. In fact, if an event of interest gets lost, and then it is not delivered to the clients, the "lost-update" problem occurs again. Currently, SIENA does not solve this problem and then a reliability mechanism must be added. We discuss this aspect in Section 3.2.

3.1 Missed Events

The "Missed Events" problem occurs when a client emits an event *before* another client submitted his subscription matching such event. There are different possible solutions for this problem. (1) To provide an event-persistence mechanism inside the system. (2) To insert an additional component, a *Repeater*, that provides caching services, in the system architecture. (3) To use a pair <subscribe, notify>.

While solution (1) is invasive and requires to change the system, solutions (2) and (3) both operate at *application-level*. That is, they do not change the publish/subscribe architecture, but instead combine its features to reconcile publisher and subscriber actions. We will describe all of them in the following.

3.1.1 Stable Events

A possible solution would be to provide event persistence directly inside the event-service. That is, the event-service itself stores all the events and it will deliver them when somebody will submit a matching filter.

At first we should distinguish two kind of events: *stable-* and *vanish-events*. While *vanish-events* are events that do not need to be stored, *stable-events* represent those events that require persistence. This solution, similar to the one adopted by the Java Message Service (JMS) [9], is particularly invasive, it requires a protocol re-implementation and it introduces many questions that need an answer such as "How long a notification should be stored?" or "May a notification be erased from the repository?" and "Who can delete it?", etc...

3.1.2 Repeater

On the other hand, the *Repeater* acts as a network client and receives and stores all notifications sent through the network (see Figure 6). New users that are looking for a specific event can ask the *Repeater* for old notifications regarding that event. Although easy to develop, this does not look like

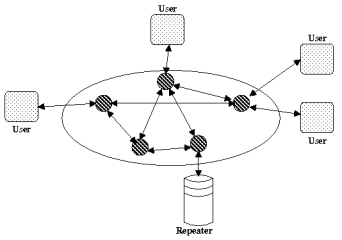

Figure 6. A Subscribe/Notification Architecture with a Repeater.

a good solution. In fact it poses the same questions mentioned in Section 3.1.1 and one more: "Does the presence of a *Repeater* vanish the (asynchronous and loosely-coupled) Publish/Subscribe Architecture philosophy?".

3.1.3 A subscription-notification combination

The basic idea in this solution is to use a combination of a subscription and a notification. As explained before, the main problem is when a client subscribes for an event after the provider published an announcement for that event.

The following actions explain how a pair <subscribe,notify> works (see also Figure 7):

- provider:

  step p1: subscribe for "I need service *S*"

  step p2: publish "Service *S*"

- client:

  step c1: subscribe for "Service *S*"

  step c2: publish "I need service *S*"

Note that these are not *atomic actions* but there is a little time between the subscription and the notification. Moreover, we assume that the Publish/Subscribe service is unreliable and messages could be delayed through the network. So, different cases are possible. For space reason we show only one of them, the interested reader can refer to [1] where it is shown how the pair <notify,subscribe> works in all cases. Figure 7 depicts how this solution avoids the missed-event problem. We suppose that the client subscribes for an event after the provider sends its notification. Since we are

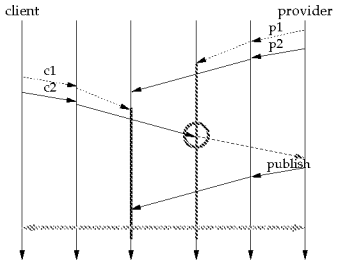

Figure 7. Both users send a pair <subscribe,notify>.

using the pair <subscribe,notify>, the provider has been subscribing for interested parties (action p1). This subscription will catch client's notification (action c2); then the provider knows that a user needs his service. Therefore the provider can publish again his notification.

3.2 Reliability Mechanism

Although *reliability* is one of the most important non-functional requirements needed by a distributed application, it is difficult to be guaranteed in a loosely-coupled asynchronous environment such as Publish/Subscribe systems. SIENA does not provide any explicit mechanism for managing reliable communication.

In distributed event-based systems, two different level of reliability can be recognized: *inter-service* and *service-to-client*. While *inter-service* reliability concerns the communication between two nodes in the event-service, *service-to-client* involves a client and its access-point. Since the particular implementation of SIENA does not distinguish between these two kind of communication, it is possible to develop the solution once and apply it in both cases.

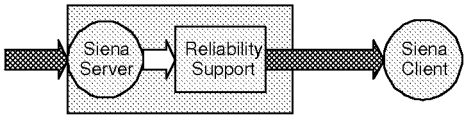

Figure 8. A Support Service for Reliable Communication.

Also in this case we have investigated different approaches: (i) an invasive one that requires to modify the protocol. (ii) a non invasive implementation developed as support-service. Previous work done in designing support-services for the SIENA Middleware [2], suggest how to operate. The idea (see Figure 8) is to build a wrap around the SIENA server that intercepts the outgoing communication,

extracts information of interest and implements the new service transparently to the under-layer protocol. This new service should store the notification whenever the communication is not available and keep it until the communication will be restored. Of course some kind of policy is needed in order to avoid infinitely-stored events.

4 Conclusion and Future Work

In this work we have presented and discussed our ongoing experience in integrating the THINKTEAM PDM application and the SIENA Publish/Subscribe Middleware. The use of an event-based system should avoid the so called *lost-update* problem. This problem, usual in such kind of applications, arises when a client performs a *check-out/modify/check-in* cycle on a document that may be used as reference copy by other clients. In this context SIENA should allow for disseminating alerts about the state of the document.

This integration poses new requirements on how to manage events and subscriptions: (i) since alerts must reach every interested client, the system should avoid lost-events. (ii) Moreover, since the native semantics of SIENA does not prevent the *missed-events* problem, a specific solution is required.

This paper has pointed out these aspects and proposed some possible solutions. Currently, ongoing work is in two directions: (i) to validate these solutions with respect to the mentioned problems. They will be implemented as additional services for SIENA and will be tested to measure their impact on the system performances. (ii) Since THINKTEAM is implemented by using MICROSOFT® Technology, such as COM/DCOM, not compatible with SIENA distributed implementations, we have been porting SIENA-client code to this platform.

Acknowledgment

The authors would like to thank Alessandro Forghieri and Maurizio Sebastianis from THINK3 INC. for their contribution to this work.

The authors would also like to acknowledge the Italian national project C.N.R. SP4 that partly supported this work.

References

[1] M. Caporuscio. Co.M.E.T.A. - Mobility support in the Siena Publish/Subscribe Middleware. Master's thesis, Università degli Studi dell'Aquila - Dipartimento di Informatica, L'Aquila - Italy, March 2002.

[2] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071, Dec. 2003.

[3] M. Caporuscio and P. Inverardi. Yet Another Framework for Supporting Mobile and Collaborative Work. In *International Workshop on Distributed and Mobile Collaboration*, Linz, Austria, June 2003.

[4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving Scalability and Expressiveness in an Internet-Scale Event Notification Service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, Portland, OR, July 2000.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[6] A. Carzaniga and A. L. Wolf. Content-based Networking: A New Communication Infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, pages 59–68, Scottsdale, Arizona, Oct. 2001. Springer-Verlag.

[7] P. Fenkam, E. Kirda, S. Dustdar, H. Gall, and G. Reif. Evaluation of a publish/subscribe system for collaborative and mobile working. In *Proceedings of the Eleventh IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 02)*, 2002.

[8] ITU-T Recommendation Z.120. Message Sequence Charts. ITU Telecommunication Standardisation Sector.

[9] Sun Microsystems Inc., Mountain View, California. Java message service.

[10] The Product Data Management Information Center. Understanding PDM. http://www.pdmic.com.

[11] Think3 Inc. ThinkTeam: a Product Data Management Application. http://www.think3.com.