Issues in Supporting Event-based Architectural Styles

Antonio Carzaniga
Univ. of Colorado at Boulder
Dept. of Computer Science
Campus Box 430
Boulder, CO 80309-0430
carzanig@cs.colorado.edu

Elisabetta Di Nitto
CEFRIEL and UCI
CEFRIEL
Via Fucini, 2
20133 Milano, Italy
dinitto@elet.polimi.it

David S. Rosenblum
Univ. of California, Irvine
Dept. of Inf. and Comp. Science ICS2 209
Irvine, CA 92697-3425
dsr@ics.uci.edu

Alexander L. Wolf
Univ. of Colorado at Boulder
Dept. of Computer Science
Campus Box 430
Boulder, CO 80309-0430
alw@cs.colorado.edu

1. INTRODUCTION

The development of complex software systems is demanding well established approaches that guarantee robustness of products, economy of the development process, and rapid time to market. This need is becoming more and more relevant as the requirements of customers and the potential of computer telecommunication networks grow.

To address this issue, researchers in the field of software architecture are defining a number of languages and tools that support the definition and validation of the architecture of systems. Also, a number of *architectural styles* are being formalized. Each of them defines "a set of design rules that identify the kinds of components and connectors that may be used to compose a system or a subsystem, together with local or global constraints on the way the composition is done" [5]. The formalization of styles helps the understanding and categorization of existing architectures and supports developers in the definition of the structure of new systems.

A style that is very prevalent for large-scale distributed applications is the *event-based style*. In an event-based style, components communicate by generating and receiving *event notifications*. A component usually generates an event notification when it wants to let the "external world" know that some relevant event occurred either in its internal state or in the state of other components with which it interacts. When an event notification is generated, it is propagated to any component that has declared interest in receiving it. The generation of the event notification and its propagation are performed asynchronously. Usually, a connector called an *event service* (or an *event dispatcher* or *bus*) is in charge of managing the propagation of the event notifications. This propagation is completely hidden to the component that generated the event. Thus, the event service implements a multicasting mechanism that fully decouples event generators from event receivers. This provides two important effects:

- A component can operate in the system without being aware of the existence of other components. All it has to know is the structure of the event notifications that are interesting to it.
- It is always possible to plug a component in and out of the architecture without affecting the other components directly.

These two effects guarantee a high compositionality and reconfigurability of a software architecture.

In the last few years, interest in the event-based style among practitioners has resulted in the development of a number of *event-based middleware infrastructures* (see for instance [3], [7], and [6]). These infrastructures implicitly support the event-based style; that is, they provide APIs and frameworks for defining applications structured according to this style.

We started investigating the event-based style two years ago in two separate research efforts where we participated in the definition of a general model for event-based architectures [4] and in the development an event-based infrastructure called JEDI [1]. By using JEDI and by comparing it with other systems and infrastructures, we recognized a number of different variations of the event-based style. These variations have different impact on the structure, the behavior, and the performance (in other words, on the architecture) of applications. Therefore, these variations need to be carefully analyzed and explicitly defined as part or specialization of the event-based style, in order to be exploited whenever the architecture of a system is defined.

In this paper we identify the event-based style variations introduced by a number of event-based middleware infrastructures and point out the advantages and drawbacks of the different approaches as well as the open issues.

2. EVENT-BASED STYLE AND MIDDLEWARE INFRASTRUCTURES

Figure 1 shows an operational and pragmatic description of the event-based style. An architecture that realizes this style is characterized by a connector called an *event service*. It is in charge of dispatching event notifications. The

components can be classified into two categories: *recipients* and *objects of interest*. Recipients declare their interest in receiving event notifications by issuing a *subscribe* operation offered by the event service. Objects of interests notify the occurrence of an event by sending a *publish* request to the event service. Alternatively, the event service itself can poll objects of interests to know if some event has been produced. A component can behave both as an object of interest and a recipient of events. The event service reacts to a publish request by forwarding the corresponding event notification to all the recipients that have subscribed to it. This high-level architectural style is being exploited by most of the event-based infrastructures that have been currently implemented. In the following we mention a few of these that provide significant variations of the style.

CORBA defines the concept of *channel*, which is a simplified version of an event service [3]. All the recipients that are connected to a channel receive all the notifications that are published by object of interests on that channel.

Smartsockets [7] proposes a more powerful approach in which the event service can accept subscriptions for a number of different subjects. Each notification is characterized by its *subject* and a *data part*. A component receives all the event notifications that belong to the subjects to which it has subscribed. Therefore, the subject defines a kind of virtual connector between objects of interest and recipients. The same approach based on the subject has been adopted by TIBCO for the development of TIB/Rendezvous [8].

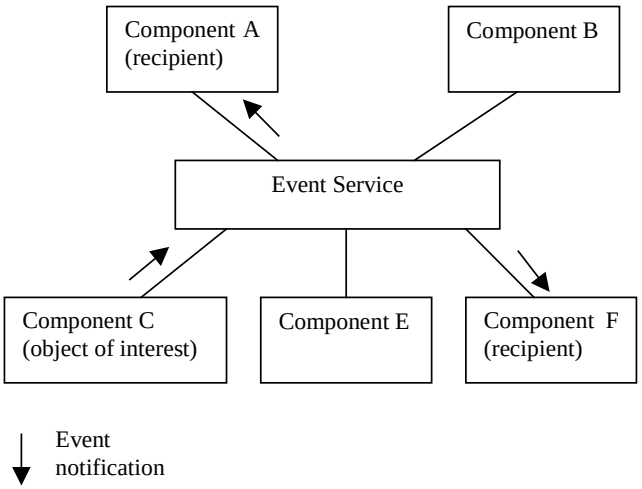

Figure 1. Architecture of an event-based system.

Event channels and subjects are simple mechanisms for connecting event receivers and objects of interest. However, they are not very flexible. If, for instance, an object of interest is interested in producing an event notification on a number of subjects or channels, it has to explicitly publish the notification on all of them.

An alternative approach is the one adopted by Elvin [[6]]. In Elvin, notifications are sets of named and typed data elements. A subscription is a declarative boolean expression over the components of event notifications. By issuing a subscription, a component can declare its interest in a number of notifications characterized by some common property. Notice that this property is established by the subscriber, and it is not hard-coded in any element of the notification (as in Smatsockets) or in a channel (as in the CORBA event service).

JEDI [1] provides a mechanism for event subscription having a similar expressive power. In JEDI a notification is defined by a name and by a number of parameters. For instance, `Alarm(PC1, HALTED)` is a notification whose name is `Alarm` and has two parameters whose values are `PC1` and `HALTED`. In JEDI, event receivers subscribe for *event patterns*, which are expressions over the name and parameters of a notification. So, for example, `Alarm*(_, _)` would match all the notifications whose name starts with `Alarm` and that have two parameters.

Yeast [2] is an *event-action system*. It observes event sequences and reacts to their occurrence according to some action specification. It is not an event-based infrastructure *per se*, since its event service triggers actions relevant to human beings rather than delivering notifications to other software components. However, it encapsulates interesting mechanisms for observing events. Differently from JEDI, in Yeast an event pattern can be composed of a number of *event descriptors* combined together using some logical and temporal operators. For instance, the event pattern "`file foo mtime changed then in 10 minutes`" is matched 10 minutes after a change to file foo.

3. THE IMPACT OF MIDDLEWARE INFRASTRUCTURES ON APPLICATION ARCHITECTURES

We argue that the assumptions introduced at the implementation level by event-based infrastructures have an impact on the structure of the architectures implemented on top of them, and therefore define new event-based architectural sub-styles. In this section we briefly discuss about the architectural implications of three important aspects of middleware infrastructures:

- The mechanism that supports the selection of the recipients to be notified of the occurrence of an event (the *subscription mechanism*).
- The structure of notifications managed by the middleware infrastructure.
- The scalability properties of the middleware infrastructure.

3.1 Subscription mechanisms

The subscription mechanism influences the configuration of architectures and the interaction among components. As an example, suppose that we are building an application composed of three classes of components, `A`, `B`, and `C`.

Also, suppose that they interact through an event-based style and, in particular, that components of type A send two types of events, one that is supposed to be received by components of type *B* and the other that is received by components of type C.

In this case, if we use the CORBA event-based approach, the architecture of the system will reasonably contain two event channels, one connecting components of type A with components of type B, and the other connecting components of type A with components of type C. Since components of type B and components of type C will be connected to different connectors, they will receive separate sets of events. Components of type A will be in charge of selecting the proper event channel depending on the type of event it generates.

Conversely, by using Smartsockets, all the components will be connected to the same connector, the RT Server, and the dispatching of the events will not depend on the configuration of the architecture, but on the content of subscriptions.

3.2 Structure of notifications

The structure of notifications that are produced and consumed by components has an impact on the communication protocol established among them. For instance, if notifications contain minimal information related to an event, a recipient of an event would need to engage in a complex interaction with the object of interest in order to get additional information about the circumstances in which the event occurred. As an example, let us consider the case of a system for software deployment in which one component, *A*, is in charge of notifying the release of a new software product. Other components can subscribe to this event notification and, upon receiving it, can deploy the new software on the nodes where they are running.

In the case where we adopt a flat structure for notifications, the event notified by *A* can have the following appearance:

"Product A released on April 4th"

The components that receive this notification must parse it, extract the information about the software being released, and then engage some kind of communication with component A to know how to download and install the software.

The exploitation of an object-oriented notification structure provides more support to event recipients in the interpretation of the event semantics. In fact, if we adopt an object-oriented model, *A* can generate a notification containing the information on the released product plus a method to download and install the product. Upon receiving the notification, recipients can invoke this method to get the product installed without being aware of the location of the code or of the downloading and installation procedure.

3.3 Scalability properties

The internal architecture of the event service significantly influences the performance of the architectures built on top of it. Intuitively, it has to *scale* to accommodate a growing number and distribution of components. If we assume that the event service is implemented as a centralized element, it can rapidly become a critical bottleneck as the number of components it has to serve grows. This scenario becomes even worse when components are distributed over a wide-area network.

To solve this problem, in JEDI the event service itself is built as a set of distributed components, the *event servers*, organized in a hierarchy. Each event server manages the communication among a set of components geographically located in the same "neighborhood" and also connects them to other remote components by forwarding messages to and from other servers. In particular, subscriptions are stored and forwarded upward in the hierarchy until they reach the root server while notifications are sent to all the local subscribers, to all the lower-level servers that have forwarded a corresponding subscription, and then upward in the hierarchy.

In JEDI the distribution of the event service is hidden to components and does not have an impact on the functional behavior of the system. In particular, the notification delivery functionality of the event service is guaranteed regardless of whether the objects of interest and recipients are located closely to each other. This requires a consistent amount of information to be exchanged among event servers.

In the case in which recipients and objects of interest of the same event are confined to the neighborhood managed by a single event server, the performance of the whole system could be even worse than in the centralized approach, since messages would be unnecessarily propagated to the top of the hierarchy. Thus, other propagation mechanisms for subscriptions and publishing of notifications need to be identified and evaluated. Also, the assumption that the event servers are organized hierarchically should be assessed against other alternative topologies.

4. CONCLUSION

We argue that the event-based style presents interesting characteristics for the development of distributed, highly-decoupled systems. Several infrastructures that support the development of event-based applications have been developed. Each of them makes specific assumptions on the structure of notifications, on the mechanism that allows components to declare their interest in some event, and on the way scalability of architectures is supported. All these assumptions have an impact on the final architecture of the applications that are developed on top of these infrastructures. Conversely, considerations concerning the architectural structure of applications influence the choice of the underlying event-based infrastructure. In this paper

we have briefly discussed these issues, pointing out the advantages and drawbacks of each approach.

The evaluation we are currently carrying out results in a more general consideration that we have begun to analyze in more detail: *middleware infrastructures (not necessarily event-based) implicitly define architectural (sub)styles.* The knowledge of these styles can be profitably used when the architecture of an application is defined. It is therefore desirable to explicitly define them in terms of architectural elements, so that they can provide guidelines to application developers and can support the transition of an architecture into an implementation on top of a selected infrastructure.

5. ACKNOWLEDGEMENTS

We thank Gianpaolo Cugola and Alfonso Fuggetta from Politecnico di Milano for their important contribution to the accomplishment of the work described in this paper.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9701973. This effort was also sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number F49620-98-1-0061. This work was also supported in part by the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, the Air Force Office of Scientific Research or the U.S. Government.

REFERENCES

[1] G. Cugola, E. Di Nitto, and A. Fuggetta, “Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems”, In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[2] B. Krishnamurthy and D. S. Rosenblum, “Yeast: A General Purpose Event-Action System”, *IEEE Transactions on Software Engineering*, Vol. 21, No. 10, October 1995.

[3] Object Management Group, “CORBAservices: Common Object Services Specification”, December 1997.

[4] D.S. Rosenblum and A.L. Wolf, “A Design Framework for Internet-Scale Event Observation and Notification”, In *Proceedings of the 6th European Software Engineering Conference* (Joint with SIGSOFT '97, Foundations of Software Engineering), Zurich, Switzerland, September 1997.

[5] M. Shaw and P. Clements, “Toward Boxology: Preliminary Classification of Architectural Styles”, In *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco (CA), USA, October 1996.

[6] B. Segall and D. Arnold, “Elvin has left the building: A publish/subscribe notification service with quencing”, In *Proceedings of AUUG97*, September 1997.

[7] Talarian Corporation, “Mission Critical Interprocess Communications - an Introduction to Smartsockets”, white paper.

[8] TIBCO Corporation, “TIB/Rendezvous”, white paper. http://www.rv.tibco.com/rvwhitepaper.html.