Exploiting an event-based infrastructure to develop complex distributed systems

G. Cugola, E. Di Nitto, A. Fuggetta
CEFRIEL – Politecnico di Milano
Via Fucini, 2
20133 Milano Italy
+39 2 239541
e-mail: {cugola, dinitto, fuggetta}@elet.polimi.it

ABSTRACT

The development of complex distributed systems demands for the creation of suitable architectural styles (or paradigms) and related run-time infrastructures. An emerging style that is receiving increasing attention is based on the notion of event. In an event-based architecture, distributed software components interact by generating and consuming events. The occurrence of an event in a component (called source) is asynchronously notified to any other component (called recipient) that has declared some interest in it. This paradigm holds the promise of supporting a flexible and effective interaction among highly reconfigurable distributed software components.

We have developed an object-oriented infrastructure, called JEDI (Java Event-based Distributed Infrastructure), to support the development and operation of event-based systems. During the past year, JEDI has been used to implement a significant example of distributed system, namely, the OPSS workflow management system.

The paper illustrates JEDI main features and how we have used it to implement the OPSS workflow management system. Moreover, it provides an initial evaluation of our experiences in using an event-based architectural style.

Keywords

Event-based systems, distributed systems, workflow, business processes, object-orientation.

1 INTRODUCTION

Convergence between telecommunication, broadcasting, and computing is opening new opportunities and challenges for a potentially large market of innovative network-wide services. The class of users interested by this revolution is significantly large: families, professionals, large organizations, government agencies, and administrations. The services range from home banking and electronic commerce, to coordination and workflow support for large dispersed teams, within the same company or even across multiple companies.

Many research and industrial activities are currently being carried out to identify feasible strategies to develop and operate these services in an effective and economically viable way. The technical problems that have to be addressed are complex and critical. Services must be able to operate on a wide area network with acceptable performance. The software technology used to implement these services must be "light", i.e., it should be scalable with respect to the capabilities of the platform on which services are running. Moreover, the technology must enable a "plug and play" approach to support dynamic reconfiguration and introduction of new service components. Finally, it is essential to support openness, since services have to be easily extended and integrated with other services being offered on the network.

A very important research topic to be addressed to foster the diffusion of network-wide applications is the identification of proper architectural styles able to cope with the above requirements and challenges. Most architectural styles exploit Remote Procedure Call (RPC) to support communication among distributed components. Middleware infrastructures such as CORBA [10] and Java + RMI [15] are based on this kind of communication model. RPC is based on a tight coupling between the object that requests a service (i.e., the client) and the object that satisfies such request (i.e., the server). Before invoking a service, the client has to know the existence of a server capable of satisfying its request and has to obtain a reference to such server. In many situations, however, a decoupled communication model between objects would be preferable. As an example, let us consider a network management system. In this system, whenever a network node signals a failure, a procedure has to be started to fix the failure. Each node does not necessarily need to know the existence of such recovery procedure. It has simply to notify the "external world" of the detected failure. This kind of scenarios is not easy to develop using the communication model implemented by CORBA and Java+RMI.

An appropriate paradigm to address the above issue is proposed by *event-based architectures*. The components of an event-based architecture cooperate by sending and receiving *events*, a particular form of messages. The sender delivers an event to an *event dispatcher*. The event dispatcher is in charge of distributing the event to all the components that have declared their interest in receiving it. Thus, the event dispatcher allows decoupling between the sources and the recipients of an event.

The relevance and potential impact of the event-based paradigm has been acknowledged by OMG that has recently defined an event service on top of the CORBA framework (see Section Related work). Nonetheless, there are several open issues that need to be addressed to define effective and workable event-based infrastructures. As a contribution to this research work, we have developed an event-based, object-oriented infrastructure called JEDI (Java Event-based Distributed Infrastructure). JEDI has been used to implement a network-wide Process Support System (PSS) called OPSS (ORCHESTRA Process Support System).[1] A PSS [2] is an environment for developing and executing process-based (or also workflow-based) applications. A process-based application is a software system supporting a *coordinated set of activities involving both humans and computerized tools*. Typical examples, are business services such as customer care or interoffice procedures.

The contributions of the paper can be summarized as follows:

- It introduces JEDI, a new event-based infrastructure suitable to develop a wide range of distributed systems.
- It illustrates how we have exploited JEDI to develop OPSS, and discusses the advantages derived from the adoption of an event-based approach.
- It presents our experiences in using the event-based paradigm.

Consistently, the paper has the following structure: "Section A quick tour of JEDI" presents JEDI basic concepts and implementation; Section "OPSS: ORCHESTRA " provides an overview of the architecture of OPSS; Section "Evaluation" provides an evaluation of our experience; Section "Related work" presents the related works; finally, Section "Conclusion" draws the conclusions.

2 A QUICK TOUR OF JEDI

2.1 The architecture of JEDI

Figure 1 describes the architecture of JEDI. The infrastructure is based on the notion of *active object*[2] *(AO).* An AO is an autonomous entity that performs an application-specific task. An AO interacts with other AOs by producing and consuming *events*. Events are a particular type of message. Conventional messages are sent from a source to one or more recipients, as specified by the source itself. Conversely, events do not include any information about their recipients. An event is *generated* by an AO and *notified* to other AOs (*event recipients*) that are dynamically selected by a specific component of the infrastructure called *event dispatcher (ED).* ED waits for the occurrence of an event, and delivers it to those AOs that have explicitly declared their interest in receiving it. An AO declares the classes of events it is interested in by invoking an *event subscription* operation. It can also stop accepting events of a given class by invoking the *unsubscribe* operation. Event subscription and unsubscription can be invoked at any time during the active object lifetime. The notification of events is accomplished *asynchronously* with respect to their generation.

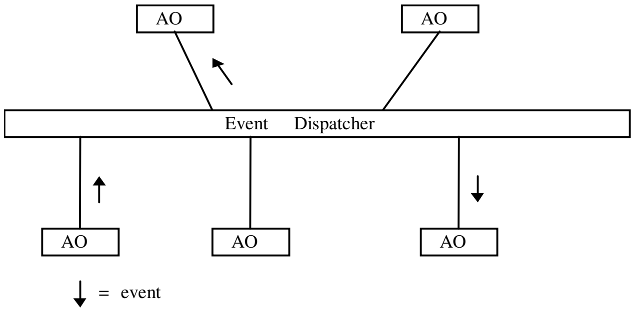

Figure 1: A logical view of JEDI architecture.

In JEDI, an event is an ordered set of strings. The first string is the *event name*. The remaining strings are *event parameters*. In the paper, an event will be represented using a notation similar to function calls in traditional programming languages (e.g., `open(foo.c,read)`, where `open` is the name of the event, and `foo.c` and `read` are its parameters). We have chosen this simple event structure for the sake of flexibility and interoperability. By exploiting the dynamic binding and type checking features offered by Java we could have defined events as Java objects, thus significantly enriching their semantics. However, this choice would have introduced several constraints on the network-wide availability of the system.

AOs can either subscribe to a specific event or to an event pattern. An *event pattern* is an ordered set of strings representing a very simple form of regular expression. The first string of the pattern (i.e., the pattern name) may end with an asterisk, while the other strings are either standard strings or strings composed of the single character '_'. Given a pattern `p`, an event `e` matches the pattern iff the following conditions

---

[1] OPSS has been developed as part of the ORCHESTRA project 9, funded by Telecom Italia.

[2] We have not used the term "component" since it is heavily overloaded and could have induced some confusion.

hold:

- The name of `e` is equal to the name of `p`, if the latter does not contain the asterisk; or the name of `e` starts with the same characters of `p` name, excluding the asterisk.
- `e` and `p` have the same number of parameters.
- Each parameter of pattern `p` that is not equal to '_', is equal to the corresponding parameter of event `e`.

According to our experience, active objects often operate according to a quite standard sequence of operations. Upon activation, the AO subscribes to a set of events and then starts waiting for their occurrence. When an event is notified, the AO performs some operation (possibly generating new events) and then starts waiting again. It therefore executes a standard loop: wait for any event among those it has subscribed to, and then process it. For this reason, we have introduced a particular type of active objects called *reactive objects*. A reactive object exhibits an abstract method (called `processMessage`) that has to be specified by the programmer and that is automatically invoked each time the reactive object receives an event it has subscribed to. JEDI provides classes to implement both generic active objects and reactive objects (see next section).

Reactive objects offer also a mechanism to support mobility. A reactive object can autonomously decide to move to a different site by invoking the `move` operation, which causes the following actions to occur:

1. The state of the reactive object is serialized and saved using standard Java facilities.
2. The reactive object moves to the new location and informs the ED that it is ready to receive events.
3. The ED keeps the events that should be received by the migrating reactive object until it is ready to receive them.

There are two versions of the ED that exploit different implementation strategies: centralized and hierarchical. In the centralized approach, the ED is constituted by a single process. The hierarchical approach has been introduced to address the issue of scalability at a network-wide level. In many critical applications (e.g., network management), the number of AOs is very high and they are typically dispersed on a large number of hosts. Moreover, the number of events to be dispatched becomes extremely large. In this context it is vital to identify means to reduce the event traffic and optimize the performance of the distribution mechanism. To address this issue, the hierarchical ED has been structured as a collection of processes (usually, one for each machine running JEDI) interconnected to form a tree. Each AO connects to anyone of these processes. Events are propagated across the ED process tree on the basis of the subscriptions posted by each AO. Notice that AOs behavior is not influenced by the implementation strategy chosen for the ED. The decision of exploiting the centralized or the hierarchical version only affects the overall performance of the system. We do not provide here further details on this issue since it is not the main focus of the paper.

In summary, the event-based communication style used in JEDI is characterized by the following properties:

- it is asynchronous;
- it is based on multicast;
- the source of a communication cannot specify the destination of the communication;
- the destination of a communication does not necessarily know the identity of the source;
- events are guaranteed to be received in the same sequence in which they are produced;
- a reactive object can move without loosing the occurrences of the events it has subscribed to.

2.2 The implementation of JEDI

JEDI has been implemented as a set of Java classes and supports the development of pure event-based applications (i.e., applications that communicate only by exchanging events). JEDI includes the event dispatcher and the components needed to develop active and reactive objects. These components have to be properly tailored according to the specific requirements of the system to be implemented. JEDI includes two Java packages. Package `polimi.jedi` contains all the classes needed to implement active objects. Package `polimi.jedi.dispatcher`, includes the classes that implement the event dispatcher. Figure 2 and Figure 3 describe the UML logical design of the two packages.

Each active object communicates with the event dispatcher through the methods offered by the interface `ConnectionToED` shown in Figure 2. This interface includes all the operations needed to produce events, receive event notifications, subscribe to and unsubscribe from events. The infrastructure provides two implementations for this interface, through classes `RMIConnectionToED` and `SocketConnectionToED`. The former uses RMI to connect to the event dispatcher (i.e., to implement the relationship `connectedTo`), while the latter uses standard TCP/IP sockets.

JEDI provides an abstract class `ReactiveObject` to implement reactive objects. Users may easily implement new reactive objects by creating subclasses of `ReactiveObject`. These subclasses have to provide a suitable implementation for the abstract method `processMessage`.

Figure 3 illustrates the Java classes used to implement the event dispatcher (package `polimi.jedi.dispatcher`). The event dispatcher supports connections based both on RMI and on standard TCP/IP sockets. TCP/IP connections allow non-Java active objects to exploit the features of the JEDI event dispatcher. Classes `EventQueue` and `Register` store the queue of events that have been received and not yet dispatched, and the received event subscriptions respectively.

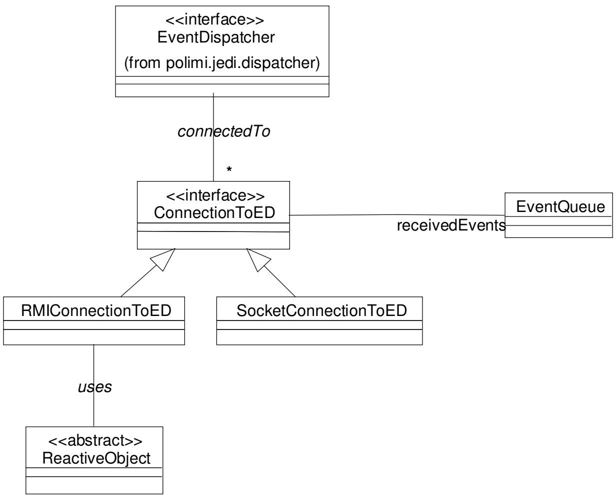

Figure 2: Packagem polimi.jedi.

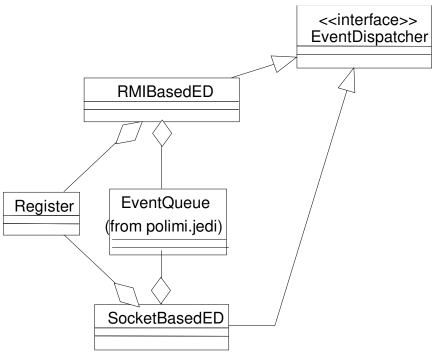

Figure 3: Dispatcher (package polimi.jedi.dispatcher).

3 OPSS: ORCHESTRA PSS

ORCHESTRA is a multimedia, distributed infrastructure offering a range of advanced telecommunication features [9]. In particular, it allows users to transparently access services from several types of terminals. It also supports nomadism: users can access the ORCHESTRA environment without being constrained by their physical location. Moreover, services can be distributed/replicated across the network, depending on load balancing needs. OPSS has been conceived to support the design and operation of business services on top of the ORCHESTRA infrastructure. To address these requirements we decided to exploit the JEDI event-based approach. In this section we present the main characteristics of OPSS and how it has been implemented on top of JEDI.

3.1 The Architecture of OPSS

OPSS main components are a set of *agents* and a *State Server* (see Figure 4).

3.1.1 Agents

Agents are autonomous entities able to receive an *activity description* (i.e., a process model fragment) and execute it. Activities are specified in any language that can be understood by the agents that execute them. Agents can be dynamically instantiated during the execution of the process. We use event distribution as the key mechanism to support agent interoperation. Events can be used to notify a variety of situations, e.g., the start up and the termination of an activity or the creation of a new artifact. The exploitation of the event mechanism makes it possible to achieve two important results. First, agents can be dynamically and seamlessly plugged in and out of OPSS. In particular, the creation or removal of agents does not affect (at least directly) other agents. Second, event notification defines a standard interoperation mechanism that is independent of the language interpreted by the agents.

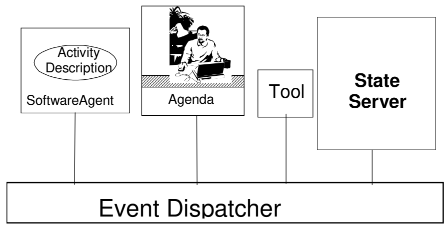

Figure 4: The ORCHESTRA Process Support System.

OPSS offers three kinds of agents: external tools, software agents, and human agents. *External tools* are (possibly off-the-shelf) components that execute business-specific activities (e.g., a configuration management tool). The activity description for an external tool is just the set of information needed to launch the tool (e.g., the initial parameters). External tools can be either OPSS-dedicated or off-the-shelf tools. The latter have to be interfaced with OPSS through a gateway. JEDI class `ConnectionToED` supports the programmer in the implementation of tools and gateways. *Software agents* are general-purpose interpreters of automated activities. In the current implementation of OPSS, activity descriptions for software agents are coded in Java. They are defined as sub-classes of `ReactiveObject`. *Human agents* are people executing creative, human-specific activities (e.g., a customer service operator). Human agents are supported by an Agenda that show their assignments and responsibilities in the process. Agenda has been explicitly developed for OPSS and uses `RMIConnectionToED` services to send and receive event notifications.

3.1.2 State Server

The State Server is in charge of coordinating agents by offering a logically centralized view of the *state of the process*. The state of the process is defined by the entities shown in Figure 5. Each entity has associated a set of possible states that define its behavior:

- `AgentInfo`. This class is used to store information on process agents. The modeled agents' states are `Available` and `NotAvailable`. In the first state the agent can be requested to execute an activity.
- `ActivityInfo`. This class is used to maintain information on the activities of the process. An activity can be in one of the following states: `Defined`, `Assigned`, `OnGoing`, `Suspended`, `Terminated`, `Aborted`. These states will be presented more in detail later on.
- `ArtifactInfo`. This class defines the information concerning the outcomes of the process. The possible states are `Created`, `OnEdit`, `Edited`, and `Destroyed`.
- `ResourceInfo`. This class contains data on the tools that can be invoked or used by OPSS (e.g., the executable code of the Java interpreter or of an external tool, devices such as a printer or an audio device). The possible states are `Available` and `NotAvailable`.

These entities are subclasses of `ProcessElement` (see Figure 5). In turn, `ProcessElement` is a subclass of `ReactiveObject`. As a consequence, each instance of these subclasses has an autonomous thread of execution that reacts to JEDI events.

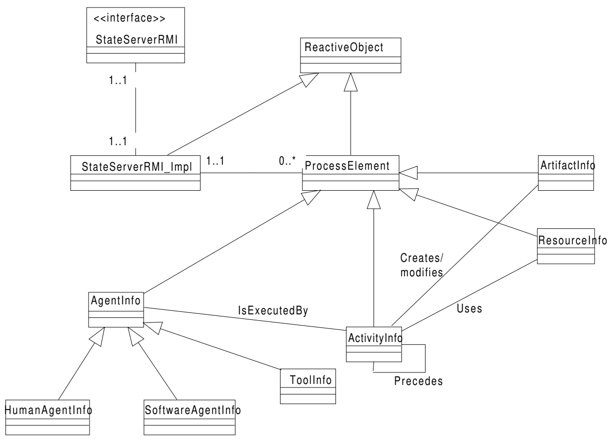

Figure 5: StateServer structure.

Each entity reacts to events according to a finite state machine, defined at the class level, called *life cycle*. It defines the set of admissible transitions between states. A transition is defined by a triple: triggering event, condition, and action. With this respect, transitions are similar to ECA rules in active databases [6]. When an entity receives an event notification E_i in state S_j, all the transitions having S_j as initial state and E_i as triggering event are evaluated for firing. One of the transitions whose condition evaluates to true is nondeterministically fired. The firing of the transition causes the execution of the action part and moves the instance to the final state. The execution of the action part of a state transition can produce new events that may influence the execution of activity descriptions and the state of other objects in the State Server.

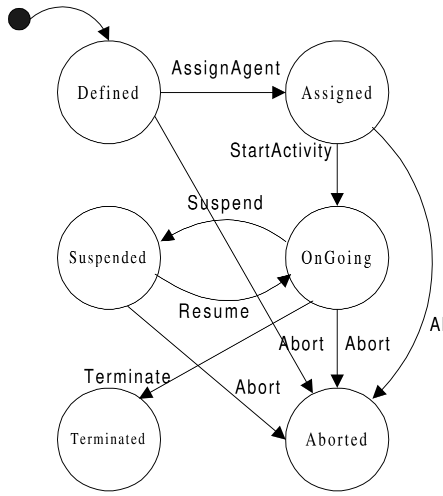

Figure 6: The Activity life cycle.

As an example, Figure 6 shows the life cycle associated with class `ActivityInfo`. When an object of this class is created, it is in state `Defined`. In this state the object is characterized by a unique identifier and by an activity description. From state `Defined` the object can move to state `Assigned` when the corresponding activity description has been assigned to an agent for execution (i.e., event `AssignAgent(activityID, agentID)` is received). The transition can only be executed if the instance of class `AgentInfo` that corresponds to the selected agent (`agentID`) is in state `Available`. Upon transition execution, the `ActivityInfo` instance moves into state `Assigned`, the `AgentInfo` instance moves into state `NotAvailable`, and the event `AgentAssigned(activityID, agentID)` is produced. Agendas usually subscribe to these types of events to provide human agents with information about their assignments. When the `ActivityInfo` instance receives event `StartActivity(activityID)`, it moves from state `Assigned` to state `OnGoing`, provided that all the activities preceding activity `activityID` have been terminated. When executing the action part of this transition, the `ActivityInfo` instance produces the event `ActivityStarted(activityID, AD-URL)`. This event is subscribed by the agent assigned to activity `activityID` or, if she is a human agent, by her Agenda, and triggers the execution of the activity. Parameter `AD-URL` contains the location of the activity description to be executed.

The State Server main class is `StateServerRMI_Impl`. It defines the inherited method `processMessage` to react to events like: login of users and creation of new activities, artifacts, or resources. The dynamic behavior of the State Server is very simple: it subscribes to and waits for the above events. When one of such events occurs (e.g., a new activity needs to be started), the State Server creates an object able to describe the state of the corresponding entity in the process (i.e., the new activity) and to keep track of its evolution. Therefore, at any time, the information stored in the State Server mirror the state of the process being executed.

Beside this event-based interface, the State Server exports a set of services through which any Java component can query the state of the running process (i.e., of the instances of these subclasses). These services constitute a synchronous interaction mechanism that is not directly supported by JEDI. The motivation of this choice is discussed later on in Section "Evaluation".

4 EVALUATION

The development of OPSS has demonstrated that the main advantage of the event-based paradigm supported by JEDI is the easy re-configurability of the system. For instance, we have recently integrated a process monitor in OPSS without affecting the behavior of the other parts of the system. The process monitor simply subscribes to the events that represent a change of the process state and visualizes it accordingly. However, our experience has also identified some problems and open issues, as we will briefly discuss hereafter.

4.1 Synchronous vs. asynchronous communication

In JEDI, active objects communicate using a pure event-based style. Namely, the only mean for an active object to send (receive) an information is to generate (receive) an event. Events are sent and received in an asynchronous way. We have noticed that in many situations an active object, after generating an event, needs some response from the recipient(s) of the event in order to perform the next operation. For instance, consider the case in which an agent needs to notify the State Server that a new activity has to be created and that this activity has to be assigned to a certain agent. The agent executes the following code fragment:

```
sendEvent("DefineActivity(ActID,ActType)");
sendEvent("AssignAgent(ActID,AgentID)");
```

The execution of this code might be erroneous because of possible race conditions. For instance, the State Server, that reacts to event `DefineActivity`, might be unable to create the corresponding `ActivityInfo` object before the event `AssignAgent` has been produced. As a result, this last event would be lost since the `ActivityInfo` object would be late in subscribing to it. In this case the State Server would not be able to properly keep track of the agent assignment.

To avoid this situation, it is convenient that the agent receives the confirmation of the creation of the `ActivityInfo` object before generating the next event. In JEDI, this behavior can be obtained by programming the event recipient to produce an event that acts as a "response" to the initial event. This way, the source of the initial event can explicitly subscribe to this event and wait for the event occurrence before producing the `AssignAgent` event. This solution is quite cumbersome and expensive, since it requires the exchange of a high number of messages between the event source, the recipient(s), and the event dispatcher.

An alternative solution would be to explicitly define in JEDI the concept of "return value", from the event recipient(s) back to the agent that has generated the event, and to provide the programmers with mechanisms to easily manage these values. In particular, we are introducing an additional synchronous operation for event generation that requires a "return value" from the recipient(s) of the event. The execution of this operation allows an active object to send an event to the dispatcher and wait until some information is returned from the event recipient(s) or, if no object is interested in the event, from the event dispatcher. When the event has multiple recipients, several policies can be envisaged to manage the return values. For instance, the source can wait for the first return value, or it can wait until all the recipients have provided a response. In this latter case the event dispatcher should inform the source of the number of return values that it should receive.

Notice that this additional synchronous mechanism still preserves the anonymity of the recipient(s) of the event, since the exchange of return value can be still managed by the event dispatcher. More in general, it preserves the basic semantics of events (multicast dispatching, and anonymity of both source and recipients), still introducing a significant amount of flexibility and optimization in the management of complex agent interaction patterns.

4.2 Event granularity

We have experienced a significant problem in identifying the events to be exchanged among agents. If the granularity of events is very low, many events have to be generated, since each of them has a poor or limited meaning. This choice might significantly complicate the programming activity, reduce the performance of the system, and make it difficult to test and monitor the system. On the other side, a too course-grained definition of events might hide inside agents significant operations that must be made visible to the rest of the system. For instance, consider the example presented in the previous section. In that case, the events `CreateActivity` and `AssignAgent` (that gave us several synchronization troubles) could have been replaced by a unique event carrying the information about both the creation of the activity and its assignment to the specified agent. This design choice reduces the number of exchanged events but modifies the semantics of activities: any activity can be created only if a proper executing agent has been already selected.

There is no universal solution to this problem. It is the designer's responsibility to evaluate the trade-off and select the most suitable solution, based on the constraints and requirements of the problem being addressed.

4.3 Client server vs. event-based design paradigms

The main problem a programmer encounters using a pure event-based approach is that the programming philosophy differs from the traditional client-server approach that she is used to. In a client-server approach interaction between components occurs when one component is not able to perform some operation and asks the other one to do it on its behalf. In an event-based approach, components are autonomous entities that inform the "external world" of the main changes occurred in their internal state or in the state of the components and devices they can observe. The notification of an event is seen by a component as an external stimulus that can determine a change in its internal state. Thus, collaboration among components is indirect.

Based on this consideration, a main step in understanding both architectural paradigms should be the identification of the classes of systems that better suit each approach. Since they address different requirements, we might discover that event-based and client-server approaches are not alternative. Instead, they can be profitably integrated even in the same system. In OPSS we have tried to use the event-based approach to guarantee autonomy of process agents and reconfiguration of the system. Moreover, we exploited the client-server approach to query the global state of the process maintained by the State Server. We are aware, however, that a more systematic study is needed.

4.4 Open issues: network-wide event distribution and mobility

The development of OPSS has emphasized the need for powerful and efficient mechanisms to support the notification and distribution of events on a network-wide scale (e.g., on the Internet). The event-based infrastructure must guarantee that the services implemented on top of it are made available to users dispersed over the Internet. The hierarchical ED we implemented may represent an initial solution to the problem. However, there are still a number of issues to be addressed. In particular, a distributed ED provides an overall performance improvement only if the number of messages exchanged for each delivered event across the ED components is "reasonable". According to our current experience, several aspects have an impact on this issue, such as the topology of the connections of ED components, and the expressive power provided by the subscription mechanism. Colleagues at the University of Colorado at Boulder and UC Irvine are addressing this issue by defining and assessing new architectures for distributed EDs.

We argue that mobility of reactive objects as it is supported by JEDI represents a powerful mechanism for implementing sophisticated applications. However, it may introduce several problems when combined with ED distribution. The ED has to provide specific mechanisms to guarantee that moving objects do not receive duplicated events and that the original ordering of events is respected. We provided a specific solution for our hierarchical ED, but the impact of this issue on alternative ED architectures is still to be understood. Finally, we still lack an extensive experimentation of this mechanism since it was not exploited in the OPSS implementation.

5 RELATED WORK

This section surveys event-based infrastructures and compares them with JEDI. Also, it shows the impact that the adoption of an event-based approach had on OPSS, by comparing our system with similar state-of-the-art PSSs.

5.1 Event-based infrastructures and frameworks

In the past years there has been a growing interest in distributed software architectures capable of easily supporting dynamic system reconfiguration. The event-based paradigm provides a very promising solution to the problem. It breaks the tight connection between clients and servers, eliminating the need for clients to know the identity of servers. Several examples of event-based systems may be found in literature. They differ in the structure of the events that can be dispatched, the way events are observed, the mechanisms for event subscription, and their overall run-time architecture (see [13] for a detailed characterization of these aspects). In general, the products and approaches we mention in this section do not support the mobility of the software components exchanging events.

Multicast RPC [3, 18, 19] (also known as group RPC) allows a client to invoke a service on a group of servers which exports the same interface. Servers "register" to a class of messages (service requests) by joining a group and by exporting the common interface defined for the group. This is quite different from the approach taken by JEDI. In JEDI event consumers use a more powerful declarative approach to "register" to a class of messages and they do not need to export any common interface. Moreover, multicast RPC is a synchronous communication mechanism in which an answer is required, while JEDI implements an asynchronous communication mechanism without answer. From this viewpoint, multicast RPC is complementary to the JEDI approach, and could be similar to the synchronous mechanism we advocated in Section Evaluation.

Linda [5] is the precursor of a generation of languages aiming at describing and supporting cooperative computations. The basic idea is that different autonomous computations can cooperate by reading and writing information through a shared repository (or *space*) of information *tuples*. Each Linda program can *read* a tuple from the repository on the basis of its contents, using a pattern matching mechanism. A *read* operation does not remove the tuple from the repository. Linda offers also a *consume* operation that reads the tuple and remove it from repository. There are several differencies between Linda and JEDI (and, in general, the event-based paradigm). First, JEDI makes it possible to "declare", through the *subscribe* operation, the class of events which an application is interested in. As a consequence, the application will receive all the events that conform with the *subscribe* declaration. It does not need to explicitly request them further. Events are distributed by the ED to the application as they are produced and asynchronously with respect to the main control flow of the application. Conversely, in Linda each *read/consume* operation is independent of each other and is synchronously executed by the Linda program. Second, JEDI (as any other true event-based approach) guarantees that all the parties that have declared their interest in an event will eventually receive it. This is enforced by the JEDI run-time support based on subscription requests. In Linda the only way to achieve a similar effect is to work at the application level. For instance, before removing the tuple, a Linda program might check for some global information to be sure that all the other interested parties have already read it. Another possibility is that each event producer writes multiple copies of a tuple, one for each interested party. This means that the producer must know the number of interested parties. In both cases, the correctness of the event distribution semantics is left to the programmer's responsibility.

Event-based systems can be considered as an evolution of a well-established class of products often called MOMs (Message-Oriented Middleware) [11]. In MOMs, explicit message queues are used to distribute messages. They guarantee delivery of messages and location transparency. In several MOMs, there can be multiple consumers for the same message queue. A queue is therefore similar to a Linda tuple space. We argue that MOMs exhibit the same problem of Linda. In fact, even if a MOM made it possible to just "read" a message from the queue without removing it, this would be a decision left to the consumer. It can't be guaranteed that the event is delivered to all the interested parties.

Tooltalk [14] is a product derived from FIELD [12] that was originally conceived to support tool integration in software engineering environment through a message exchange facility. Tools can subscribe to events, send events, and receive the events they have subscribed to. Events in Tooltalk can either be asynchronous or synchronous (they are called notifications and requests respectively). In the latter case, the recipients are supposed to provide the source with a return value. This approach is similar to the one we are developing for JEDI (see Section Synchronous vs. asynchronous communication). The publish/subscribe semantics implemented by ToolTalk is typically oriented to support tool integration in a CASE environment and is insufficient in other application domain. In particular, Tooltalk offers two event visibility levels: session and file. A session is defined as the set of all tools served by the same Tooltalk server. Usually, each user launches one or more Tooltalk servers, each of them controlling a separate group of tools. A program can subscribe to all the messages belonging to a session and/or related to a file. This mechanism makes it impossible a wide application of the approach. For instance, it is not possible to develop a monitor tool that subscribes to the events related to all files.

The CORBA event service [10] defines two roles for system components: *event supplier* and *event consumer*. They are described by two different IDL interfaces that provide methods to exchange events between event suppliers and consumers. The structure of a CORBA event is hidden to the event service. Events are distributed from suppliers to consumers through *event channels*. An event channel allows multiple suppliers to communicate with multiple consumers asynchronously. An event-based system may include several event channels. A component of the system (either supplier or consumer) may be connected to several event channels.

The CORBA event service differs from JEDI significantly. A CORBA event is distributed on the basis of just one (implicit) attribute: the name of the event channel where the event was originally posted. The event will be dispatched to all the consumers attached to that channel. The contents of the event is "not visible" to the event channel, and is not used to manage the distribution of the event. Conversely, a JEDI event is composed of a set of attributes. Producers do not see different channels. They simple post these structured events to the ED. Consumers can flexibly subscribe with a single "declarative" operation to a class of events that is dynamically defined using event patterns. Consequently, the expressive power of JEDI is higher than CORBA. CORBA event channels can be easily simulated using JEDI event names, while it is quite cumbersome and inefficient to simulate JEDI patterns in CORBA. It is indeed necessary to write a specific code that in general will need to poll different CORBA event channels. In general, if the JEDI pattern includes a selection criterion that involves event attributes other than the event name, the equivalent CORBA consumer must be "programmed" to perform the selection of desired events based on the analysis of the event contents. This means that while the JEDI ED can avoid dispatching events that do not match the selection criterion, the equivalent CORBA consumer receives and discards a number of undesired events, with an increase of the event traffic.

TIBCO is an infrastructure for creating and maintaining large distributed and event-based applications [17]. It has been used over the past years to integrate financial and banking applications (especially, trading services for financial markets). It offers several interesting features including reliable and scalable distribution of events. It exploits a three-level hierarchical event dispatcher. From the available documentation it seems that TIBCO offers an event structure that is similar to the one offered by CORBA, i.e., a labeling mechanism to assign names to events. Therefore it seems it lacks the ability of defining event patterns as in JEDI.

C2 is an event-based architectural style that has been designed to support the development of GUI software [16]. In C2 multiple software components can communicate through connectors that manage the routing and broadcasting of events. Components and connectors form a DAG (Direct Acyclic Graph). In this DAG, each component can communicate only with the two connectors "below" or "above" it. Events are classified as notifications and requests, depending on the fact that they travel down or up in the DAG, respectively. There are several differences between C2 and JEDI. In C2 the component developer does not have any event definition and generation primitive. Actually, C2 notifications are messages automatically sent out by the C2 run-time support to notify the execution of a component method invocation. It is not possible for the component developer to define and generate events with a different semantics. Moreover, C2 requests (i.e., synchronous communications) are not anonymous and are not multicasted. In JEDI, we do propose the introduction of a synchronous mechanism (the return receipt), but we preserve the anonymity of senders and receivers and the possibility of multicasting the event.

Yeast main component is a centralized server that observes event sequences and reacts to their occurrence according to some action specification [8]. Users can add new event-action specifications while Yeast is running. Events can be either operating system events (e.g., file changes) or messages produced by the components of the system. Events can be combined in a sequence using some logical and temporal operators. Actions can include any command that can be executed by the computer command interpreter. Yeast and JEDI are quite different and complementary. The former does not offer any event dispatching functionality, but provides sophisticated mechanisms for defining, observing event sequences, and reacting to their occurrence. Thus, Yeast functionality can be easily implemented on top of JEDI as a proper active object.

5.2 PSSs

It is worthwhile to compare OPSS with the state-of the-art in PSSs, to better appreciate the impact that the adoption of JEDI has had on its development and on its range of features and functionalities.

A first relevant system is ProcessWall [7]. It is a process state server providing storage for process state, and operations for defining and manipulating the structure of the state. The applications that actually execute the process operate as ProcessWall clients. They execute the process activities and invoke the ProcessWall operations to modify the state of the process in order to reflect the result of their processing. An event dispatching system is used to notify the interested clients of changes occurred in the state of the process. ProcessWall is similar to the OPSS State Server. The main difference is that ProcessWall uses the event-based communication model only to notify state changes to its clients. The clients communicate with ProcessWall via RPC. Conversely, the OPSS State Server supports both RPC and event-based interaction.

Another PSS that presents characteristics similar to OPSS is Endevours [4]. It has been developed to support distribution of process execution, lightweight installation and reconfiguration, and easy integration of components executing process fragments with tools and hyperwebs of artifacts. Its architecture is composed of three main levels: the *user level*, that is in charge of managing the interaction with users, the *system level* that defines the main process abstractions (e.g., activities, artifacts, …), and the *foundation level* that manages object persistency and distribution. Both Endevours and OPSS provide a decentralized execution of processes, i.e., they exploit multiple process engines. The main difference is that Endevours does not rely on the event-based approach to coordinate the interaction of different engines: they interact by sharing the artifacts and information stored in a passive repository.

The definition of the information stored in the OPSS State Server has been inspired by the work presented in [1]. In that paper a CORBA-based PSS is described. It is connected to other tools through the CORBA ORB. The PSS manages activities, artifacts, resources, and agents. They are associated with a life cycle. A state transition defined in the life cycle of an object is executed if the corresponding event occurs. From the available publications, we have been unable to understand the mechanisms used at run-time to manage event creation and notification. Therefore, it has been impossible to carry out a detailed comparison of the architectures of the two approaches.

6 CONCLUSION

In this paper we have illustrated the main features of JEDI, an event-based infrastructure for the development of complex distributed systems. JEDI exploits the notion of event and standard Internet technology to provide the software developer with a programming framework where multiple active objects cooperate by generating and consuming events. JEDI has been used to implement a significant example of distributed system, namely the OPSS Process Support System. JEDI offers a simple set of mechanisms to create multiple active objects that interoperate by exchanging events. The entire architecture is based on very simple and orthogonal concepts. Events are asynchronously distributed to subscribers. All the operations related to event subscription and event notification are managed in a highly dynamic and flexible way. OPSS is a significant example of distributed system whose development has greatly benefited from the availability of an event-based infrastructure. By exploiting JEDI features, OPSS can offer an extremely flexible and dynamically changeable support for workflow management.

The main lessons we have learned from the work described in this paper indicate that the event-based approach certainly offers significant advantages over traditional RPC and conventional message-based communication techniques. These advantages are also demonstrated by the growing interest in this technology that has been demonstrated by both academia and industry. Nevertheless, a number of technological issues concerning event-based architectures have to be explored. In this respect, we argue that the most critical issue to be addressed is the identification of appropriate design and implementation strategies that make it possible to integrate different (and sometime conflicting) features such as Internet-wide scalability, enhanced event model (e.g., object-oriented), synchronous and asynchronous event handling mechanisms, event filtering. Moreover, we still miss effective methodological guidelines to guide and support the design of event-based systems. We plan to further investigate these issues since they are critical impediments to the effective exploitation of the event-based architectural style.

ACKNOWLEDGEMENTS

Authors wishes to thank Antonio Carzaniga, Carlo Ghezzi, Dennis Heimbigner, David Rosemblum, and Alex Wolf for their important contribution to the accomplishment of the work described in this paper. They wish also to thank S. Beretta, C. Colombo, S. Montaruli, S. Sargenti, and F. Vadalà who provided an essential support in the development and implementation of JEDI and OPSS.

OPSS development has been funded by Telecom Italia under a contract managed by Armando Limongiello. The views and the conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Telecom Italia.

REFERENCES

1. K. Alho, C. Lassenius, and R.Sulonen, "Process Enactment Support in a Distributed Environment", WET ICE '95, IEEE Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, Berkeley Springs, West Virginia, April 20-22, 1995.

2. V. Ambriola, R. Conradi, and A. Fuggetta. "Assessing Process-Centered Environments", *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, July 1997.

3. K. P. Birman and T. A. Joseph, "Reliable Communication in Presence of Failures*", ACM Transactions on Computer Systems*, 5(1), February 1987.

4. G.A. Bolcer and R.N. Taylor, "Endevours: A Process System Integration Infrastructure", IRUS Conference on Software Process Improvement, Practice and Experience, January 24, 1997, Irvine, CA.

5. N. Carriero and D. Gelernter, "Linda in Context", Communication of ACM, 32, 4, April 1989.

6. P. Fraternali and L. Tanca, "A structured approach for the definition of the semantics of the active databases, *ACM Transactions on Database Systems*, 1995.

7. D. Heimbigner, "The ProcessWall: A Process Server Approach to Process Programming", Fifth ACM/SIGSOFT Conference on Software Development Environments, 9-11 December 1992, Washington, D.C.

8. B. Krishnamurthy and D.S. Rosemblum, "Yeast: A General Purpose Event-Action System", *IEEE Transactions on Software Engineering*, vol. 21, no. 10, October 1995.

9. A. Limongiello, R. Melen, M. Roccuzzo, V. Trecordi, J. Wojtowicz, "An Experimental Open Architecture to Support Multimedia Services Based on CORBA, Java and WWW Technologies", IS&N '97, Cernobbio (Como), Italy, 27-29 May 1997.

10. Object Management Group, "CORBAservices: Common Object Services Specification", July 1997, ftp://ftp.omg.org/pub/docs/formal/97-07-04.pdf

11. OVUM, "*OVUM Evaluates: Middleware*", OVUM Ltd, 1996.

12. S.P. Reiss, "Connecting Tools Using Message Passing in the Field Environment", *IEEE Software*, July 1990.

13. D.S. Rosenblum and A.L. Wolf, "A Design Framework for Internet-Scale Event Observation and Notification", 6th European Software Engineering Conference (Joint with SIGSOFT '98, Foundations of Software Engineering), Zurich, Switzerland, September 1997, to appear.

14. Sun Microsystems, "Integrating applications with the SPARCworks 3.0.1 toolset. http://www.sun.com/software/Products/Developer-products/literature/int_tool/preface.html

15. Sun Microsystems, "Java Remote Method Invocation Specification", February 10, 1997, ftp://ftp.javasoft.com/docs/jdk1.1/rmi-spec.pdf

16. R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J.Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component-based architectural style for GUI software, *IEEE Transactions on Software Engineering*, vol. 22, no. 6, June 1996.

17. TIBCO Enterprise Toolkit White Paper. http://www.tibco.com/products/etkwhite.hml

18. K. S. Yap, P. Tripathi, and S. Tripathi, "Fault Tolerant Remote Procedure Call", *Proceedings of 8th International Conference on Distributed Computing System*, June 1988.

19. X. Wang, H. Zhao, and J. Zhu, "GRPC: A Communication Cooperation Mechanism in Distributed Systems", ACM Operating System Review, 27(3), 1993.