Event based systems as adaptive middleware platforms

Andreas Frei, Andrei Popovici, and Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology Zürich
CH-8092 Zürich, Switzerland
{frei, popovici, alonso}@inf.ethz.ch

ABSTRACT

Adaptive middleware is increasingly being used to provide applications with the ability to adapt to changes such as software evolution, fault tolerance, autonomic behavior, or mobility. It is only by supporting adaptation to such changes that these applications will become truly dependable. In this paper we discuss the use of event based systems as a platform for developing adaptive middleware. Events have the advantage of supporting loosely coupled architectures, which raises the possibility of orthogonally extending applications with the ability to communicate through events. We then use this ability as a way to change the behavior of applications at runtime in order to implement the required adaptations. In the paper we briefly describe the mechanisms underlying our approach and show how the resulting system provides a very flexible and powerful platform in a wide range of adaptation scenarios.

1. INTRODUCTION

Advances in hardware architectures and the widespread availability of wireless networks have radically changed the computing environments a software application must face during its active life time. A way to cope with the increasingly dynamic nature of the computing environments (e.g., mobile or pervasive computing) is to use adaptive software architectures. Of the several possibilities for implementing such architectures [9], event-based systems offer a wide range of advantages [11]. The main one is the high degree of decoupling between components. By using events as the way to communicate, components are independent of each other and can therefore change and evolve independently of other components. This property fits rather well with the need to cope with unexpected changes in the computing environment. One can treat changes as new types of events being thrown into the system and adaptation as the ability to dynamically react to or generate new events.

The type of changes and the range of adaptations we have in mind can be illustrated with a few examples. As a first example, consider changes in network technology that result in an increase in available bandwidth. Applications that today exchange very and very compressed events to avoid cluttering the network may have to cope later with a larger number of more complex events that are generated taking advantage of the additional bandwidth. Ideally, we would like order applications to be able to deal with the new stream of events without having to redesign or change them in a significant manner (which might be practically impossible if the applications are embedded in real life appliances such as cars or refrigerators). One way to do this is to dynamically add a software layer that deals with the new events and acts as a translator for the older, less capable application.

A second example are changes in policy that may force applications to cope with new types of events not foreseen at the time they were designed. An automobile control system, for instance, may be required to generate an event when the combustion in the engine is less than perfect. The event can then be used to monitor the pollution level caused by the automobile. The ability to monitor the combustion is present in most modern cars but the software that turns the information into an event is probably not. Ideally, it should be possible to dynamically add the ability to generate such an event without having to change the software already installed in the automobile.

These examples are similar in that coping with such changes requires to dynamically extend already deployed and possibly already running applications. In this paper we outline a system that supports exactly this type of adaptation. The system is based on PROSE [14], a platform for dynamic aspect-oriented programming (AOP) [7, 5, 12]. Dynamic AOP allows points in the execution of a component to be extended, at runtime, with additional functionality. We use this functionality to dynamically alter or as needed the way applications react to events, the types of events they react to, the events that need to be filtered, the information contained in each event, and even the events generated by each application. This adaptive event based system is then used as the basis for implementing application level adaptation.

The rest of this paper is structured as follows. Section 2 discusses the system’s architecture, followed in section 3 by an explanation on how eventization of applications can be achieved with aspect-oriented programming. Section 4 describes the event-system design using JXTA. Section 5 concludes with examples where the proposed architecture can be used.

*Effort sponsored in part by the Swiss National Science Foundation NCCR MICS (Mobile Information and Communication Systems).
2. SYSTEM OVERVIEW

Figure 1 depicts the basic architecture of the system. We assume each node runs a Java Virtual Machine (JVM). On top of this JVM, a first layer implements the basic resources of the system. These basic resources constitute the adaptive middleware layer and include a platform for dynamic AOP and the event system. A second layer is devoted to the dynamic resources acquired at run time and that will modify the behavior of the application and/or the event system. This second layer contains application extensions, i.e., additional code that will be executed at the appropriate moment (e.g., upon arrival of a particular type of event, when the application reaches a particular state, when an event is raised, etc.). In addition to these two layers, we assume there is an application running on the same JVM and that this is the application that will be adapted as needed.

The adaptations that we consider are of three generic types. The first type of adaptation consists of endowing an application with the ability to generate events (either because it did not generate any event at all or because it did not generate the necessary events). The second type of adaptation transforms applications into consumers of events (again, either because the application did not accept any events to start with or because it did not accept the necessary events). The third type of adaptation affects the event system itself and allows to change the way it works, e.g., the way events are generated and published, the information the events contain, the format of the events, etc.

The way the system works is as follows. The dynamic AOP platform intercepts the execution of both the application and the event system and monitors their progress. Whenever they reach selected points in the execution, the dynamic AOP platform redirects the execution to the appropriate extension. Once the extension is executed, the control flow returns to either the application or the event system that then resumes its work (exactly where and how work is resumed depends on the nature of the extension being executed).

In what follows we explain all these mechanisms in more detail.

3. ADDING EVENTS

3.1 Dynamic AOP support

The basics for adaptation in our event based system is Dynamic Aspect Oriented Programming [7, 5, 14, 12]. Aspect-Oriented Programming (AOP) [8] is a software design technique that allows the separation of orthogonal concerns within an application. These orthogonal concerns are then programmed as separate aspects rather than locating them in many different places in the code. The main advantage of AOP is precisely the possibility of abstracting out concerns that crosscut through the application (i.e., appear in many different places throughout the code). These aspects can then be treated as separate software modules thereby increasing the modularity of the design. For instance, Zhang and Jacobsen [17] argue that aspect-oriented re-factorization can enhance the modularity of middleware and reduce structural complexity.

An aspect defines what to do (e.g., invoke an additional method) when a particular point is reached in the code (e.g., when invoking methods with certain signatures, when modifying a variable, etc.). Conventional AOP uses a weaver to add the aspect code to the base code of a program at compile time, e.g., AspectJ [16]. In dynamic AOP, the aspect code is added (woven) at run time by executing it whenever the specified point in the execution is reached. Aspects can also be dynamically withdrawn (unwoven) leaving the application in its original state.

In our system, we use PROSE [15, 14] as the platform for dynamic AOP. PROSE hooks into the JVM and intercepts byte-code operations to identify the points in the execution where aspects are to be executed. In PROSE, these aspects are known as extensions. A local Extension-Manager inside PROSE allows to load and unload extensions at run-time and also provides support for securely adding, removing and executing these extensions.

Figure 2 shows how PROSE is used to introduce extensions that deal with events. The figure assumes an application that is not aware of the events and depicts how extensions can be used to turn the application in a consumer and/or producer of events.

3.2 Producers of Events

Transforming an application into an event producer requires two things. The first is to detect the application state
that should lead to the generation of an event. The second is to actually publish the event using whatever event system is available. We deal with these two problems using a single extension.

Once the designer has identified when events are to be generated (e.g., after a variable has been updated, when a method is invoked, etc.), an extension is created that traps that particular situation. The code for the extension contains the logic necessary to publish the corresponding event. In Figure 2, step 1 indicates the points in the execution that are relevant for the generation of the events. When these points are reached, the extension is invoked and the event is passed on (step 2) to the underlying event system which will then publish it (step 3).

As an example, consider a calendar application running on a PDA. The following PROSE extension can be used to automatically generate an event for every meeting entry the user makes in the calendar. In this way, other calendars can be synchronized and notifications being sent to other persons or applications (e.g., to reserve a meeting room).

```java
aspect AddMeetingEventsAsProducer {
    // mechanism for publishing events
    EventSystem eventSystem;
    // definition of the points were events should be fired
    eventizablePoints(): field sets(MeetingDates *)
    at eventizablePoints() do {
        Event e = new Event(MeetingDate);
        eventSystem.publish(e);
    }
}
```

This aspect defines, using a pseudo-aspect language, the operation that has to be intercepted (line 6). As explained, it corresponds to state changes in the MeetingDates class. Line 8 defines the extension to be executed. This extension creates an event object with the information about the meeting and then calls the eventSystem component to publish the event.

3.3 Consumers of Events

Transforming applications into consumers is done via an event consumer extension. The extension does two things. First, it subscribes to events of interest. Then, upon arrival of an event, it calls the corresponding method(s) of the underlying application. The procedure is the reverse of the one used for event producers. In Figure 2, when an event arrives (step 3), the event system notifies the extension (step 4) which will then invoke the methods in the application (step 5).

Following the calendar example, this extension can be used to automatically generate entries in the calendars of users as a result of an entry being made in a particular calendar. In PROSE, such an extension looks as follows:

```java
class CalendarBecomesEventConsumer {
    EventSystem eventSystem;
    { eventSystem.subscribe(MeetingDate); }
    // action when events are received
    public void processEvent(MeetingDate received) {
        // add the date to the calendar
    }
}
```

When the extension is inserted it subscribes in the initialization phase 5 for MeetingDate events. When such an event arrives, the processEvent method is executed. This method inserts the date in the calendar and may notify the user of the new appointment or of potential conflicts.

4. EVENT SYSTEM

The event system we use is based on a publish and subscribe model [4, 13, 6, 7]: a producer node publishes an event and a consumer node may subscribe for this event and be notified of the occurrence of the event. TPS [2] is a type based publish and subscribe model built on top of JXTA [10, 3]. In our implementation we use a similar approach to TPS. By using the P2P protocols if JXTA it provides for nodes to discover each other, self-organize into peer groups, advertise and discover network services, communicate, and monitor each other.

4.1 JXTA Overview

Since we are mainly interested in pervasive and mobile computing applications, we have chosen JXTA as the underlying event system. Unlike Jini [1], for instance, where a centralized lookup service is responsible of distributing services to interested nodes, JXTA does not rely on a centralized server. This is a significant advantage in ad-hoc application scenarios where it is not feasible to use a centralized server. Similarly, JXTA uses peer groups to combine peers with similar services or behavior. As such, a peer may reside in different peer groups at the same time. We use this functionality to apply extensions to entire groups rather than to individual nodes. Being able to do so is a significant advantage when dealing with realistic scenarios where several dozen devices are involved in the interaction.

JXTA defines three layers: a core layer, a service layer and an application layer. The core layer is built up of the protocols defined in JXTA. As such the core functionality supports creating peer groups, finding services, establishing communication channels, or using a secure communication. On the service layer new services can be created to extend the functionality of JXTA. The application layer is used to create applications using the services created in the lower layer.

There are six protocols [10, 3] defined in the core layer. At the top: the Peer Discovery Protocol, the Peer Information Protocol, and the Pipe Binding Protocol. These protocols use a layered protocol stack (from top down): Peer Resolver Protocol, Rendezvous Protocol, and the Peer Endpoint Protocol.

4.2 Event System Implementation

The event system is built on the JXTA service layer and supports peers in a peer group with the GroupEventService as shown in the code below.

```java
interface Event {
    net.jxta.Document toDocument();
}

interface EventListener {
    processEvent(Event e);
}
```
The `GroupEventService` represents an implementation for the event system. It extends the `Extension` class. As such, the `GroupEventService` can be dynamically downloaded like any other service from another peer in the peer group. Interaction with the event system can proceed as soon as the event service is instantiated and started locally. Subscription to events is done through event-templates following the type- and attribute-based subscription model introduced in [13, 2].

To maintain flexibility in the system, we use an implementation that can be switched from a centralized to a distributed treatment of events. When that happens, the extensions that receive and generate events are changed accordingly to adapt each node to the new distribution model.

## Centralized Event System

In an infrastructure where a more powerful peer is available, this peer can become the master over other peers, the slaves. The master may also take over the task of colecting events and redistributing them.

When an event is generated by a slave, the corresponding extension uses the `GroupEventService` to propagate the event to its master running the `MasterGroupEventService`. Similarly, when a consumer extension is instantiated at a node, it will subscribe to events using the `subscribe(EventTemplate, ExtensionListener)` call that send the corresponding event template to the master.

The centralized event system extension uses the `Pipe Binding Protocol`. A pipe provides a mechanism to send a message over a virtual communication channel. Different types of pipes are supported; an `unidirectional pipe` sends a message from peer `A` to peer `B`, a `propagation pipe` sends a message from a master peer `A` to slave peers `B1, B2, ...` whereas slave peers may send messages back on this propagation pipe. For both the consumer and producer extensions, their instantiation triggers a connection of the extension to the propagation pipe of the master. The master in return opens an unidirectional pipe to the new slave.

A slave sends an event or event-template to the master over a `propagation pipe`. The master in return is able to send an event-notification, back to all slaves over the propagation pipe or to a subset of slaves through the `unidirectional pipe`.

A message sent over a pipe by calling `publish(event)` needs to be of the type `net.jeta.endpoint.Message` which can be generated from the event by calling `toMessage`. Vice versa, a message received on the other end of the pipe can be recreated with `initMessage(msg)`.

## Decentralized Event System

The decentralized implementation uses the `Peer Resolver Protocol`. Instead of establishing pipes between peers the resolver protocol allows peers to send and process generic requests. The requests will be sent to any peer where the initiating peer has a connection to over the endpoint protocol. The `GroupEventService`, which uses the resolver layer, sends requests to all peers using this service. Any peer participating in a peer group may acquire the resolver for the `GroupEventService` to be able to send XML Documents.

This allows the `GroupEventService` to send a XML document to all remote `GroupEventService`. The XML document sent can be obtained from the Event by calling `toDocument()` which turns the event into `net.jeta.Document`. The document received on the remote peer is turned into an event with `initDocument(document)`.

When publishing an event, the event is sent by the `GroupEventService` to all remote `GroupEventServices` in the same peer group. In every remote `GroupEventService` the received event will be matched with the subscribed event templates. If the received event matches any event template the appropriate ExtensionListener are called.

## 5. CONCLUSION AND FUTURE WORK

In this paper we have outlined an adaptive middleware platform based on events. The platform treats events and the management of events as aspects that can be changed at run time in response to new requirements or necessary adaptations. We are currently in the process of completing the implementation of the platform and exploring more advanced forms of adaptation, including dynamic changes to the event management system itself.

We are also already using the first prototype of the platform in a variety of applications. One of them is a cooperating robots scenario where autonomous robots with wireless communication capabilities coordinate their movements through the exchange of events. Through the adaptive platform described in the paper, we can separate the software that controls the movement of the robots from the software that deals with events and the coordinated behavior. This is a significant advantage over existing designs as it greatly simplifies development, maintenance, and offers much more flexibility in terms of adaptation. The management of events and the coordinated behavior are treated as dynamic extensions and can be changed at any time. For instance, a set of extensions implement a `train formation` movement where all robots follow a leading robot. The leading robot is remotely controlled with a joystick and its movements are communicated to all other robots via events. These events are interpreted by extensions on each robot that then control the movement of the robot as a function of its position in the formation and the movement of the leader. The advantage of using this software architecture is that we can change at any time the extensions and completely modify the behavior of the system. As an example, the train formation extensions can be exchanged for a `line formation` extension where all robots move parallel to each other. The extensions can also be exchanged for new ones that implement more sophisticated behavior by generating new types of events such as robots randomly leaving the formation, speed control events, movement filters that prevent the formation from entering certain areas, etc. We are also exploring using the adaptive event system to implement data propagation schemes suitable for sensor networks. Another area which we are currently investigating is dynamic changes to the event based system itself. The changes that we are currently pursuing involve changing the communication medium (from wireless card to Bluetooth port), changing the format of the events for enhanced security (transparent encryption of the event traffic through extensions to the event system), and seamless transition from centralized (using a base station) to peer-to-peer mode.

## 6. REFERENCES


