Eleggua: An Event Infrastructure for Application Cooperation

Rubby CASALLAS - Nicolás LOPEZ - Darío CORREAL
e-mail: {rcasalla,ni-lopez,dcorreal}@uniandes.edu.co

University of Los Andes
Department of Systems and Computing Engineering
Bogotá - Colombia

Abstract. Global Software Development (GSD) is a reality but still faces a lot of challenges to give solutions to problems related to software processes themselves, and problems raised by geographical dispersion. Focusing on the latter, we are interested in building a supporting software development environment to integrate processes and tools. This paper introduces Eleggua, an infrastructure for application cooperation with a central focus on low coupling and loose integration of applications. Eleggua is based on distributed events and Web Services for communication. It also uses Event Condition Action (ECA) rules to describe application cooperation models. The goal of application cooperation is to ensure consistency and integrity of overlapping domain concepts embedded in applications that interact throughout the enactment of business processes. The main restrictions on the design are imposed by the existence of heterogeneous, distributed, legacy systems which are too expensive to modify.
Eleggua is being validated in the context of a GSD project[1] focused on improving processes of a software house that faces the challenges of geographical dispersion.

1 Introduction

Global Software Development (GSD) processes present a series of challenges and difficulties due to the geographical dispersion of the people involved [6]. In GSD processes coordination is an area that needs more support. Coordination can be considered on various levels; at a low level support applications need to cooperate for processes to be enacted. Business processes are described at higher levels. Our main motivation in this paper is at the low level.

At this level it is critical that the integration mechanisms guarantee consistency and integrity of information. The main problem at this level of coordination

[1] The project "Testing in a global environment" is directed by the Department of Systems and Computer Engineering at University of Los Andes and Heinsohn Software House S.A. The project is supported and partially financed by the "Instituto Colombiano para el Desarrollo de la Ciencia y la Tecnología Francisco José de Caldas" - COLCIENCIAS. Colombia.

is the existence of distributed, legacy systems in software companies; applications are heterogeneous, and can have common domain concepts. In order to give support to long running processes, these applications need to interact and keep consistency and integrity of overlapping domain concepts in an Internet scale. The cost of modification of these tools to interact and integrate in a distributed context is high [10].

This paper introduces Eleggua, an infrastructure for application cooperation based on distributed events and Web Services as communication mechanisms. It is also based on Event Condition Action rules and rules for external observation of applications to describe application cooperation.

We are currently validating the infrastructure in a pilot project in a real-world GSD context to support a testing and defect handling software process.

This document is organized in the following way: Section II relates our work with ideas proposed by several other authors with similar requirements and approaches. Section III introduces our approach. Section IV describes in detail the architecture of Eleggua and its components. Section V describes the implementation details of the infrastructure. Finally, in Section VI we present our conclusions and future work.

2 Related Works

Our infrastructure brings together ideas from various domains. In particular, Event Notification Systems (ENS) and ECA rule based systems meet to provide control mechanism to the enactment of GSD processes. We integrate the approach with a request/reply mechanism as well; Web Services is now a known, industry-wide standard used to integrate applications developed on heterogeneous platforms [17]. We now present various works related to the mentioned domains

2.1 Event Notification Systems

Systems based on event generation, observation and notification are a widely used architectural style for distributed, loosely-coupled systems in a variety of domains like application and usability monitoring, application awareness and mobility [12]; each of these domains introduces different functional and non functional requirements. Rosemblum et Al. [11] propose a design framework that can be used as a base to categorize and compare various systems available. Our focus is primarily in the observation and notification models which describe the interaction between the ENS and applications. The observation model defines the observation of events and event pattern occurrences. This observation is achieved through observer objects. The notification model characterizes communication between observers and interested receptors.

Hilbert et al. [7] introduce 4 different activities in their event monitoring framework: Observation, Processing, Notification and Actions. Observation is the process of collecting basic information from applications. Processing involves

computations of basic information in order to produce events. Notification involves communicating interested parties of the occurrence of events. Finally, actions are the processes performed as a reaction to a notification or to an observation. These activities, we believe, are strongly related with the ENS design framework; we use these activities to describe our observation and notification models.

Several groups have developed different alternatives for event notification systems related to our work. EDEM [7] is an event-based application and usability monitoring infrastructure; it focuses on observation of applications through probes and generation on events based on these observations. Like our work, this infrastructure is based on an ECA rule model to describe the interaction between applications and external agents that observe and react to events.

JEDI [3] is an ENS used to implement a distributed workflow management system; the key issues for this system are the distribution and the migration of applications. They argue that a mixed approach of event based and request/reply communication mechanisms can be useful for a workflow management system. We agree that these communication mechanisms can be used together and take this approach in our proposal.

SIENA [2] is an Internet-Scale ENS implemented by the authors that define the ENS design framework [11]. Its central consideration is the impact of design on performance over wide area networks. The considerations taken, due to the dispersion of wide area networks, relate to our work, where the main characteristic of the process is the geographical dispersion of applications.

2.2 Event Condition Action (ECA) Rules

ECA rules have been used on various domains over the years since their beginning in active database systems [4,18]; one of these domains is support for process enactment. Ideas presented by various authors [1,5,8,9,16], support the use of ECA rules for process control and workflow execution at various levels.

Tomboros and Geppert present EVE in [5,16], an extensible workflow system that uses an event based infrastructure. The proposed system provides an architectural component-based framework; this approach allows reuse and composing of the entities in the process. The system is built extending the functionality of an event based integration platform. Among other basic functionalities of an ENS, this system provides administration and monitoring of processes through event history. Like our infrastructure, the platform is based on the use of reactive components whose interaction is characterized in ECA rules; also, we take the approach of event histories for process administration and monitoring.

YEAST [9] is specialized in complex event processing through the use of event-action rules for local network applications. We apply in our infrastructure this event processing approach through event-action rules.

Bae et Al [1] and Kappel et Al. [8] present similar approaches for the development of workflow management systems. They include a rule based model to characterize the coordination of activities and resources. These two proposals relate to our work in the use of ECA rules for application coordination,

nevertheless, our approach covers only the coordination involved in application cooperation.

3 Our approach

Figure 1 presents the intergration model of Eleggua; its main elements are the ENS and the application representatives. The ENS is a distributed component responsible for administration of event types and subscriptions; it receives events and dispatches them to application representatives in an Internet scale. An application representative encapsulates the rules that define the integration of an application with Eleggua through ECA and observation rules. Observation rules define which events are observed and generated; ECA rules describe how events are processed amd dispatchrd to the ENS and reactions to events notified by the ENS. The main concepts of our distributed infrastructure are Event Types and Logical Events.

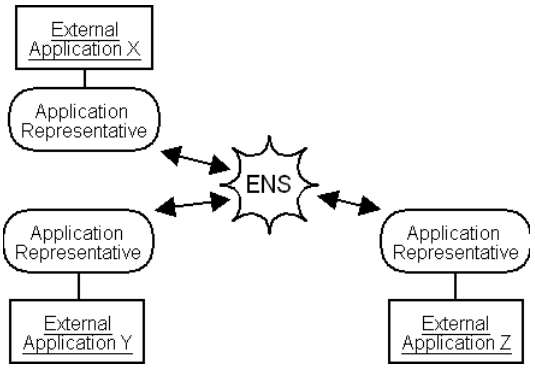

Fig. 1. Integration Model

Event Types An event type is a duple of the form <*topic, context*>; a topic represents a common domain concept for a group of applications using the ENS. These concepts are based on agreements for naming and identification of overlapping domain concepts in applications. A context is a logical event domain; for example context '*process.testing*' is the domain of all events involved in the enactment of a testing process; context '*error*' includes all event types related to errors in the process.

Logical Events Logical events are the main concept for exchange of information between applications. A logical event consists of an event type, the producer

identification, a timestamp established at event production, and a set of named parameters.

Our approach focuses mainly on the design of two models from the ENS design framework [11]: the observation and notification models. We use the activities for application monitoring [7]: the observation model is described by the observation and processing activities; the notification model is described by the notification and action activities.

Observation Model

Our observation model is based on describing how information is observed and collected from applications and processed to produce logical events. This processing is necessary in order to reduce overload of events notified; events generated are only those that have been previously defined as interesting.

Observation and processing is described in Eleggua by means of observation rules. In Eleggua we can intercept the execution of application services. An observation rule specifies the interception of a service of interest and describes how it should be processed to generate a logical event.

Notification Model

Our notification model is based on an ENS that offers publish/subscribe services. Subscriptions are declared for event types; the ENS notifies logical events to interested applications. ECA rules describe the reaction to logical events at notification time; notification mechanisms are then external to the application code. Filters on parameters are generated specifying the event type and conditions on the parameters described in the rule; logical events processed are only the ones that pass these filters.

Actions combine the event based approach with Web Services as a request/reply mechanism. Logical events contain only the minimum possible information to reduce load on the infrastructure. Further context information, required to complete reactions to notified logical events, is requested through Web Services. Actions usually include the invocation of an application's API services, the invocation of other application's APIs through Web Services, and the generation of new logical events.

4 Infrastructure Architecture

4.1 Example Scenario

We now present an example scenario of the cooperation of three applications before introducing the details of the infrastructure. The scenario is based on the enactment of a simple testing process. In parallel with the testing process, planning and tracking activities are always performed to schedule every task required during the development (including testing activities), and to record time spent by people on tasks.

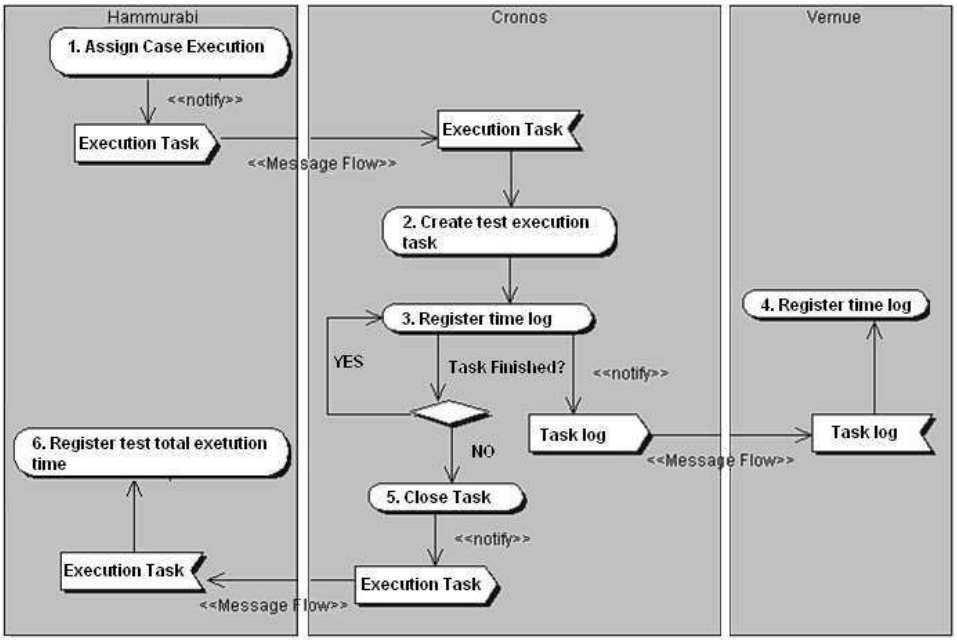

Fig. 2. Example Scenario

The example scenario is presented in Fig. 2 using an activity diagram. Various legacy applications are available to support this process. *Hammurabi* is used in the process to create test plans and to record defect reports. *Cronos* is an application for scheduling and tracking; it provides services to record planned tasks and to record time logs for these tasks. *Vernue*, a payroll application is used to calculate employee payments based on time worked on the various project activities.

Vernue, Hammurabi and *Cronos* are distributed applications that were developed independently; *Vernue is* a legacy application developed on an AS400 platform. *Cronos* and *Hammurabi* are J2EE applications. They do not share the same databases, and even, if from an abstract point of view, they deal with the same domain concepts, these concepts usually do not have the same name or the same properties in each application. These applications offer services to access their functionalities by means of simple APIs and also, by means of Web Services.

Eleggua has to provide the following services:

– When a test case execution is assigned to a user in *Hammurabi* (Fig 2. activity 1), automatically, in *Cronos* a test case execution task has to be created and assigned to the developer responsible (Fig 2. activity 2). With this information, the test case execution is planned.

– Afterwards, the developer in charge can record the time spent during the execution activities (Fig 2. activity 3). The time spent on this task has to be automatically sent to *Vernue* each time a log is entered to calculate various reports (Fig 2. activity 4).
– Once the task is finished the developer has to close the test case execution task (Fig 2. activity 5), this has to automatically send the total time spent on the task to *Hammurabi* to be able to generate other reports (Fig 2. activity 6).

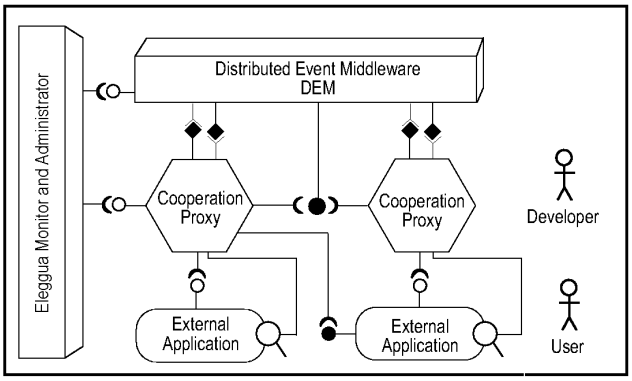

Fig. 3. Eleggua Infrastructure

Cooperation for this scenario must be described for all events crossing the boundaries of a system: execution task and task log message flows. The main concepts used to express this in our approach are logical events, observations and ECA rules. We use this scenario to explain the components of Eleggua.

4.2 General Description

Fig. 3 depicts the main components of Eleggua. The Distributed Event Middleware (DEM) provides the event notification service. The component notifies logical events to the Cooperation Proxy (CP) through a push/pull mechanism. External applications interact by means of the Cooperation Proxy (CP). ECA rules and observation rules describe this cooperation. The components are explained in further detail in the following sections.

Fig. 4 presents the graphical notation used in the infrastructure. Fig. 4 (a) represents the exchange of logical events notified to applications. Fig. 4 (b) represents a Web Service interface offered by a component (black circle) and consumed

by another component (half circumference). Finally, Fig. 4 (c) represents interception of a service execution on the component that contains the magnifying glass.

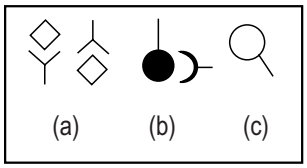

Fig. 4. Graphical Notation

4.3 Distributed Event Middleware

The DEM offers the basic functionality of a publish/subscribe ENS. Its main responsibilities are: 1) application registration, 2) application subscription, 3) event dispatching and notification to CPs, 4) event persistence, and 5) historical event queries. This component is made up of three kind of distributed sub-components: dispatching component, consumer components and producer components.

The dispatching component is the main node of communication between consumer and producer components. The consumer component provides an API for interaction with the DEM to consume events. The producer component provides the API for interaction with the DEM to produce events.

We describe in detail how these responsibilities are materialized by the components of the DEM.

Application registration This is the first service invoked by applications using the DEM; it registers the basic information of an application to uniquely identify it. The registration of a consumer application may include the specification of an event handler for push notification purposes. Applications register using a consumer or producer component; registration information is propagated and stored at the dispatching component.

Application Subscription An application must register to an event type before publishing or receiving notifications of this type of events. Subscription to event types is offered by the consumer and producer components.

Event dispatching Event dispatching is the distribution of events to consumers subscribed to an event type. Events produced are pushed to a producer component and forwarded to the dispatching component, which in turn propagates to all consumer components. Consumer components store information on

interested applications; events are stored and pushed to applications if an event handler class was declared.

Event persistence Events are persisted at the dispatching component and at consumer components for each interested application. Events are marked as consumed if event handling was successful when pushed or when pulled directly by the application. Events are also persisted at the producer component for failure recovery purposes

Other Services The DEM also offers extended services through Web Services; these services include queries of historical events, event types and subscriptions from producers and consumers. These services are used as part of actions in ECA rules.

4.4 Cooperation Proxy

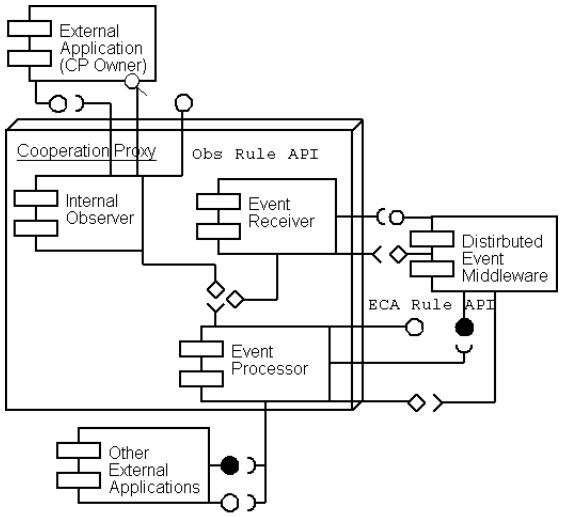

Fig. 5. Cooperation Proxy Architecture

The CP Intermediates the communication between an application and the DEM as shown in Fig. 5. Applications do not need to be modified before compilation; any application exposing services through a known API and Web Services can be integrated to the infrastructure. The responsibilities of the CP are: 1) event observation and generation, 2) event notification reception and 3) event processing. These responsibilities are grouped in three sub-components: Internal Observer (IO), Event Receiver (ER) and Event Processor (EP).

Event observation and generation The Internal Observer (IO) offers services to observe and generate events; its services are: 1) registration of observation rules, and 2) observation of applications and generation of logical events from observations.

Registration of observations includes: an API service to be intercepted, the method parameters and the event type. Parameters of the generated event are set using method parameters. Each event parameter has a name, a method parameter, and optionally a getter method to be invoked on the method parameter; in this case the event parameter is the returned object of the getter method.

Observation is achieved through method interception by means of aspect technology. Eleggua generates an aspect interception class, weaves the intercepted class, and recompiles and redeploys the application. During execution, the invocation of the intercepted method triggers the observation rule. The component generates a logical event using input parameters from the observed method, and forwards events to the Event Processor (EP).

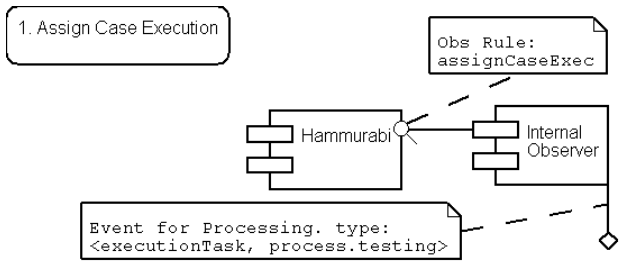

Fig. 6. Scenario Applied: Event observation and generation

Fig. 6 retakes the example scenario presented in section 4.1 to illustrate the role of the IO component. The IO at the *Hammurabi* CP has an observation rule that intercepts the execution of a test case execution assignment service. A getter method is invoked, on a value object received as parameter, in order to query the test case execution id. A logical event is generated, with type *<executionTask, process.testing>* and a parameter with name *executionId,* and is passed to the Event Processor (EP).

Event Notification Reception The Event Receiver (ER) has as main services: 1) registration of applications and subscription to event types, and 2) managing notifications of logical events.

The component interacts directly with the DEM using the API provided by a local consumer component for registration of applications and subscription to event types. The ER interacts with the EP by receiving event types. With this information, the ER subscribes to each required event type to the DEM.

Notified events are pushed by the DEM to this component. The component generates a handler class in charge of receiving events and forwarding them to the EP for event processing.

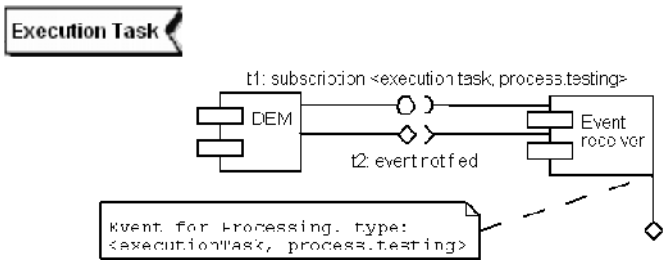

Fig. 7. Scenario Applied: Event Notification Reception

Fig. 7 returns to the example scenario to illustrate the role of the ER component. At the *Cronos* CP an ECA rule was previously registered for events with type *<executionTask, process.testing>*; the ER subscribed to the event type at the DEM. Events received with this type are passed to the EP.

Event Processing The Event Processor (EP) has services for 1) declaration and 2) execution of Event Condition Action (ECA) Rules.

Each ECA rule specifies a logical event type, a condition and an action to be executed.

- The event part of the rule declares event types that will be matched; more than one rule can be defined for a certain event type.
- Conditions specify named parameters that need to be present in the logical event.
- Actions are arbitrary code to be executed; these methods must receive as parameter the logical event instance.

When a new rule is registered for a new event type the component communicates with the Event Receiver (ER) to subscribe to it.

The execution of an ECA rules is as follows:

- The EP offers an API that processes logical event instances from the Event Receiver (ER) and the Internal Observer (IO).
- Events are matched with rules registered by event type and are then filtered by checking the included parameters according to the condition part of the rule. For each rule that matches the event type the condition is evaluated, and if passed, the action is executed.

– Actions receive the logical event instance as parameter. If an error occurs during action execution an error event is generated and sent to the DEM. Actions may change the state of an application, query for additional information, and generate events.

Additional services are provided by the component to ease the task of coding actions. These include execution of Web Services, and creation and notification of logical event instances to the DEM.

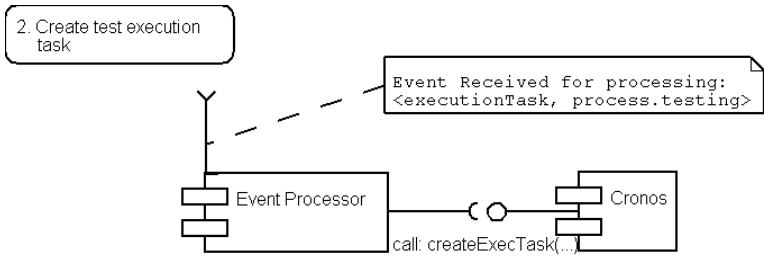

Fig. 8. Scenario Applied: Event Processing

Fig.8 illustrates the role of the EP component in the example scenario. An ECA rule has been registered at the *Cronos* CP that processes events with type *<executionTask, process.testing>* and parameters with names *projectId*, *executionId*, *caseId*, and *userId*. The action part for this rule is executed when a logical event with these characteristics is received at the EP. The action calls Web Services provided by *Hammurabi,* to complete information necessary for the test case execution task, including the requirements associated and a detailed description of the test case. Finally, the action creates a task in *Cronos* for the given user to enter time logs.

Another ECA rule is declared at the *Hammurabi* CP for logical events locally generated (as opposed to notified), with type *<executionTask, process.testing>* and with one parameter named *executionId.* The action inquires the project id, case id and user id responsible of the test case execution using a Web Service provided by *Hammurabi*. The action includes *projectId*, *caseId* and *userId* as parameters of the logical event, and notifies it to the DEM.

4.5 Eleggua Monitor and Administrator

A monitor and administration component has been developed as part of Eleggua. This is a JMX component that provides the administration services for:

– Applications: manages external applications connected to Eleggua. Event notification to an application can be stopped or restarted during execution;

events received for the application while notification is stopped are sent to the application once notification has restarted.
- Event types: offers service for listing, creating or deleting event types.
- Events: provides services for event tracing and event queries with various filters.
- ECA rules: offers services to activate or deactivate ECA rules during execution.
- Subscriptions: offers services for creation and deletion of subscriptions.
- Component performance: manages component performance notifications. These are periodically sent from each component to the Eleggua Monitor, alarms can be set to be generated if performance is decreasing.
- Configuration of component properties: provides services to query and modify component properties

Additionally, this component generates alarms for the process administrator to process and compensate errors. If an error occurs during the example scenario, i.e., the developer is not working on the project and on the requirements of the test case, the process administrator receives an alarm from the monitoring application and informs the project leader to take corresponding actions.

5 Implementation

Eleggua is implemented on a J2EE platform, specifically on the JBoss Container [14].

The dispatching component of the DEM is implemented as a central component with various consumer and producer components communicating via JMS technology. Point-to-point connections are established between the components for application registration and subscription. A publish/subscribe connection is established for event notification. All sent events are wrapped in JMS messages; these include logical events and administration events. JMS technology is provided by JBossMQ [15]. The components of the DEM are implemented using Session, Message and Entity EJBs with container and bean managed persistence.

Cooperation Proxies are implemented using Session and Entity EJBs. A CP intercepts methods using AspectJ [13] which imposes a series of constraints. Mainly, direct access to Java Classes is required to properly weave and recompile an application; all Java Classes that execute intercepted methods must be available.

The monitor application is implemented using Managed Beans that interact directly with the Session and Entity EJBs of the other components.

External applications that want to communicate and exchange information with the infrastructure are not explicitly part of the infrastructure. The infrastructure has been tested with some restrictions on the applications used. The J2EE applications offer services via a Session Bean API; observation of applications is achieved using these Session Beans. The AS400 application is accessed through Web Services; events from this application are pushed to the infrastructure.

6 Conclusions and Future Work

In this paper we have presented Eleggua, an event based infrastructure for application cooperation. The main concerns in its design are cooperation and integration of applications, and consistency of information. Restrictions are imposed by legacy systems; our goal is to achieve loose integration and low coupling of applications. Our approach focuses on:

- the interaction between external applications and the event based infrastructure;
- the communication based on events and the use of Web Services as a request/reply mechanism;
- ECA rules and observation of applications to describe the cooperation between applications.

Eleggua is being validated in a pilot project in a real-world GSD context to support a testing and defect handling software process. For this pilot project we have implemented the cooperation rules for the three applications in the presented scenario. People involved in the enactment of the process are located in different cities; tools are distributed on the various sites.

The pilot has enabled us to identify problems in the definition of cooperation rules, bugs that are difficult to identify during unit and system testing, and more complex and richer administration and monitoring requirements. Once this pilot testing phase is finished more processes will be implemented and institutionalized using the infrastructure.

The pilot project has also shown some aspects of the infrastructure that are still to be improved and are left as future work, some of these are:

- Implementing an efficient communication mechanism, for dispatching components, that fulfills the requirements imposed by a wide area network like the Internet.
- Defining a higher level language, for definition of ECA and observation rules, to help developers further in defining, testing implementing and evolving rules.
- Designing and implementing a Pattern Observer (PO) component in the Cooperation Proxy (CP). The PO has as responsibility the detection of event patterns and composition of events.
- Overcoming the restrictions imposed by AspectJ that make the deployment of new observation rules during execution a difficult process; a language independent aspect interception mechanism is desirable.
- Measuring the performance of the infrastructure; this is an area that needs further evaluation due to the problems that distributed systems present.

7 Acknowledgments

We are very grateful to every people involved in the project, particularly, to the engineers, developers and other people from Heinsohn Software House S.A. that participated in the pilot project.

References

1. Joonsoo Bae, Hyerim Bae, Suk-Ho Kang, and Yeongho Kim. Automatic control of workflow processes using eca rules. *IEEE Transactions on knowledge and data engineering 16 (8)*, 2004.
2. Antonio Carzaniga, David Rosenblum, and Alexander Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems, 19(3):332– 383*, 2001.
3. Gianpaolo Cugola, Elisabetta di Nitto, and Alfonso Fuggeta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering, 27(9), 2001*, 2001.
4. Andreas Geppert, Mikael Berndtsson, Daniel Lieuwen, and Claudia Roncancio. Performance evaluation of object-oriented active dbms using the beast benchmark. *TAPOS Intl. journal, ed K. Lieberherr and R.Zicari, Vol 4(8), pub. John Wiley and Sons, Inc, 1998*, 1998.
5. Andreas Geppert and Dimitrios Tomboros. Event-based distributed workflow execution with eve. *In Proc. IFIP Int'l Conf. on Distributed Systems Platforms and Open Distributed Processing (Mideware'98), Lake District, England*, 1998.
6. James Herbsleb and Deependra Moitra. Global software development. *IEEE Software*, March/April 2001.
7. David Hilbert and David Redmiles. An approach to large-scale collection of application usage data over the internet. *In Proceedings of the 20th International Conference on Software Engineering*, 1998.
8. Gerti Kappel, Stefan Rausch-Schott, and Werner Retschitzegger. A framework for workflow management systems based on objects, rules and roles. *In Bulletin of the Technical Committee on Data Engineering, March 1995,18(1), pp. 11-18.*, 1995.
9. Balachander Krishnamurthy and David Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering, vol. 21, pp. 845-857*, 1995.
10. Audris Mockus and David Weiss. Globalization by chunking: A quantitative approach. *IEEE Software*, 2001.
11. David Rosenblum and Alexander Wolf. A design framework for internet-scale event observation and notification. *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, 1997.
12. Roberto Silva, Cleidson de Souza, and David Redmiles. The design of a configurable, extensible and dynamic notification service. In *In Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, June 2003.
13. AspectJ team. http://eclipse.org/aspectj/. last visited on 2005-01-23.
14. JBOSS team. http://www.jboss.org. last visited on 2005-01-15.
15. JBOSSMQ team. http://www.jboss.org/wiki/Wiki.jsp?page=JBossMQ. last visited on 2005-01-15.
16. Dimitrios Tomboros and Andreas Geppert. Building extensible workflow systems using an event-based infrastructure. *In Proc. 12th Int'l Conf. on Advanced Information Systems Engineering, Stockholm, Sweden*, 2000.
17. Steve Vinoski. Integration with web services. *IEEE Internet Computing, November o December 2003*, 2003.
18. Jennifer Widom and Stefano Ceri. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.