What are my students doing? Introducing ELENA, the E-Learning Event Notification Architecture

Manuel Caeiro, Jorge Fontenla, Martin Llamas
Department of Telematic Engineering
E.E. Telecomunication, Universidade Vigo
Vigo, Spain
Manuel.Caeiro@det.uvigo.es

ABSTRACT

In this article we introduce ELENA, a new architecture for notifying events to a Learning Management System (LMS) by an external tool. At the core of this work is the idea of integrating external tools exposed as Web Services. Although some progress is being done in this field by the research community, current solutions achieve at best a very "soft" integration, it is, the LMS can link an external tool but cannot monitor or alter its workflow. The ELENA system tries to alleviate this problem by introducing an architecture that allows an LMS to track the use that teachers and students make of the tool. Therefore, the LMS can have a greater control over the integrated tools. The ELENA system is based on Web Services and specific APIs and protocols, resulting on a simple and scalable architecture.

1. INTRODUCTION

In recent years we have witnessed a massive deployment of the so-called Learning Management Systems (LMSs, [17]). These platforms provide their users (namely, students and teachers) with a holistic environment involving a broad variety of tools such as wikis, chats, podcasts, blogs, media players or simulators to carry out their tasks [3]. These features, along with the possibility to carry out learning units avoiding spatial and temporal barriers, and their low cost of deployment and maintenance, are the cause of their rapid dissemination during last years. Well-known examples of LMSs are Moodle [11] or Blackboard [4].

Unfortunately LMSs cannot cover by themselves all the educational needs, as they fail to adapt to specific contexts. This problem is currently addressed by means of extensions [10], small plugins providing the LMS with new functionalities. This mechanism allows, for example, to include a new wave simulator in an *Hydrodynamics* course, therefore expanding the original possibilities of the platform.

Nevertheless, extensions have reusability concerns as long as they must be programmed LMS by LMS. This limitation has led several working groups to work on new ways to extend LMSs. Nowadays, the most promising alternative is following the Software as a Service (SaaS) deployment model [13][6][7]. This approach relies on hosting and running the tool as a Web Service that can be invoked from the LMS: when a user wants to operate the tool the LMS grants him/her seamless access, in such a way that he/she is not aware to be operating an external system. Among the many advantages of this approach we can mention its scalability and the reusability of a single tool in many LMSs.

However, despite the advantages of the SaaS model a problem remains: how can a teacher (or an automated monitoring system) be aware of what students are doing with the tools, provided they are external systems that are out of the scope of the LMS? The tracking of users in e-Learning environments is a very important issue as it allows to identify difficulties or conceptual problems of the users as soon as they appear and assist them. This tracking is quite straightforward if they are operating an extension of the LMS, as extensions are designed to work in coordination with the core of the LMS. Nonetheless, external third-party tools deployed following the SaaS model are standalone applications which are exploited by the LMS, and in general they do not support the tracking of their users by external systems.

The goal of this article is to describe ELENA (standing for E-Learning Event Notification Architecture), an event-notification system designed and developed at the University of Vigo to support the tracking of students in distributed e-Learning environments. ELENA supports the subscription of the LMS to events in external tools and the corresponding notification of such events.

2. INTEGRATION OF TOOLS FROM THIRD-PARTY PROVIDERS

The integration of external functionalities has recently boosted due to the increase of popularity of Cloud Computing [15] and side efforts such as mashups [8]. However, in this paper we are more interested in a tighter integration of learning tools in e-Learning platforms. This section is devoted to describe what we try to accomplish by this integration, and how tight it must be.

2.1 Business Model

Figure 1 depicts the new business model where we can see three kinds of stakeholders involved, represented into clouds: LMSs, educational tools and the final users. On the one

hand, LMSs provide the core functionality of the learning platform. These include authentication modules, databases to store students' personal data (e.g. name, email, grading, preferences, portfolio), tasks sequencing, etc. Along with this functionality, LMS developers also have to embrace some specifications to support the interaction with third-party tools, involving issues such as how to control the tools and supervise them. This interaction is represented linking the LMSs' cloud with the tools' cloud as it makes possible the integration between every pair LMS–Tool. On the other hand, tools' developers also have to embrace these interaction specifications and develop their products. It is important to notice that in this scenario educational tools are standalone software products that do not need an LMS to operate. Instead, LMSs use them to complement their functionalities. Finally we have the end users that access the LMS and the tools. It is important to notice that when a user is accessing a tool the later has to communicate with the LMS to inform it about what the user is doing and to enable its control by the LMS.

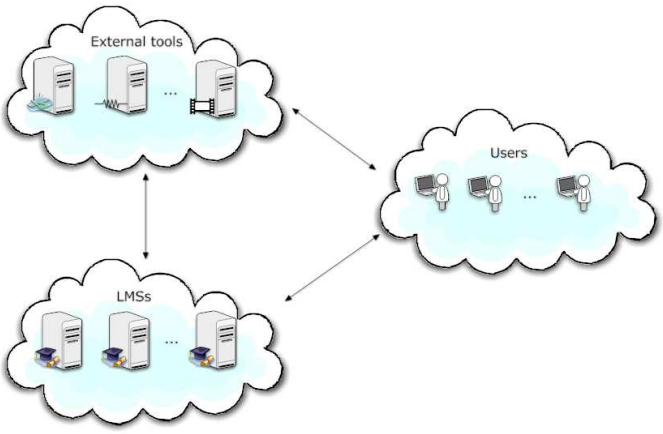

Figure 1: Business model.

This business model supposes a Copernican change with regards to the traditional model in the e-Learning domain. Up to date, the LMS has been at the centre of the business model, as it included all the provided functionality. With the Web Services approach we turn things upside-down removing the LMS from the centre, and granting the tools the same importance in the schema. Now the LMS can use many different tools, but also each tool can be used by many LMSs. This solution implies that the development of the LMSs and educational tools can follow separate paths. This business model is related to two increasingly popular methodologies:

1. Cloud Computing [15]. This approach is showing us that many applications that previously had to be installed locally can be made available through the Internet providing a satisfactory user experience.

2. Service Oriented Architectures (SOA, [12]). SOA is a concept of software architecture that allows to build up highly scalable systems, based on the invocation of stateless functions called services. The most habitual way to accomplish this is by the use of Web Services. Due to this loose coupling it is more feasible to add new services to the system.

These same approaches can be taken in order to fulfil the need of tools in LMSs, where the required simulators, text editors or evaluation tools have not to be executed locally at the server of the LMS Core, but can be provided remotely.

2.2 Soft and Hard Integration

At this point we consider two different alternatives to integrate third-party functionalities in a software system, which are also considered in [2]:

- Soft integration. The software system functionality can be extended through a hyperlink to an (external) third-party component. Once the user clicks on it, the graphical user interface of the new component is displayed. From this point, the user is operating a system that the original one cannot control by any means. Therefore, a new functionality is included, but with very low integration with the former.

- Hard integration. It includes soft integration, but providing a more comprehensible control over the integrated functionalities by means of some integration mechanisms. These integration mechanisms vary with the kind of software system. The term "hard" comes from the fact that the integration is such that the new functionality seems to be a part of the original system.

2.3 Need for Event Notification in e-Learning Systems

In this article we are only interested in studying the issues concerning the subscription and notification of events. As pointed out in [5], events in the context of a distributed e-Learning system are the result of recording and retrieving information, typically as a result of undertaking learning activities or interacting with materials. These events may contain important information for teachers, system administrators, or students themselves. Nevertheless, it is up to the LMS to subscribe to those relevant events. For example, in the case of a schematics simulator the LMS may be interested in knowing when the student launches a simulation and obtains a result, but not when the student places a resistor or a capacitor, and therefore it has to subscribe just to the corresponding event.

In traditional, centralized LMSs the notification of events is a pretty straightforward process, provided that the tools and the LMS core run at the same machine. The case of distributed LMSs is slightly different. Two peculiarities justify an in-depth study of it:

1. **Tools are generally in different network domains from the LMS core**. This fact has deep programmatic implications from the point of view of which technologies to use to make an efficient implementation of the Publisher–Subscriber design pattern [9]. In centralized systems the most common choices for notifying events to the LMS core are low-level solutions

such as signals or pipes, which usually achieve reliable and low-latency notifications. In distributed systems, however, such low-level solutions are not feasible, and therefore the choice of a suitable network-based event-notification technology deserves careful consideration.

2. **The LMS core is not the centre of the business model anymore**. Centralized systems feature a rather simple Publisher–Subscriber model, as the LMS core is the only subscriber and all the events are notified to it. Distributed systems have the peculiarity that tools are completely detached from specific LMS cores. In other words, one LMS core can use many tools, but also one tool can be used by many LMS cores. This implies that an identification mechanism is required at the tool to discern the several subscribers.

The rest of this paper is devoted to provide an in-depth study on several issues concerning the design and implementation of an event-notification system for distributed e-Learning environments.

3. USE CASE

In this section we introduce a complete example where the notification of events plays a key role in a distributed e-Learning system. This example will let us derive, in Section 4, a set of formal requirements that must be fulfilled by any e-Learning-oriented event-notification solution.

The example is based on a Web fluid simulator with the capability of notifying events, which is being operated by several users. These users are students of two different LMSs, which subscribe to the events triggered by the simulator. The LMSs are hosted and managed by University 1 and University 2 respectively.

The corporative networks of the two universities have two outbound routers. The router of University 1 is a router with NAT capabilities and does not allow inbound connections, whereas the router of University 2 is a normal router whose only open port for inbound connections with the LMS is the TCP:8080. Figure 2 depicts the scenario.

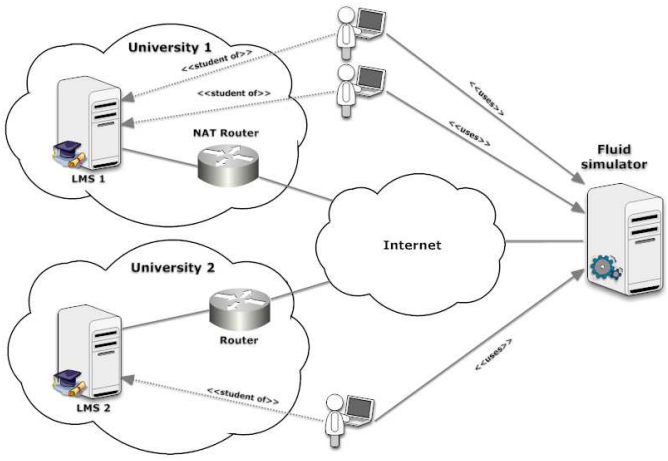

Figure 2: Summary of the use case.

The LMSs. The process starts when LMS1 grants two of its students access to an instance of the simulator. LMS1 wants to know what happens at the instance to know whether or not students can progress in their tasks or, on the contrary, they have any troubles. Therefore, LMS1 is interested in receiving information about the events `event.actions.users.execution.success` and `event.actions.users.execution.failure`, which are generated every time a simulation ends up successfully or unsuccessfully, respectively.

LMS1 cannot tell the simulator to notify it every time one of these events is triggered due to the restrictions in the configuration of its corporative network. Therefore it chooses to poll periodically the simulator asking whether or not any events of the type `event.actions.users.execution.success` or `event.actions.users.execution.failure` have been triggered since the last polling.

In a certain moment the students of LMS1 launch a successful simulation. Next, LMS1 requests information concerning all the events `event.actions. users.execution.success` and `event.actions. users.execution.failure` that took place in the last hour, and the tool replies with information about the last simulation.

Now it is LMS2 which grants its student access to another instance of the simulator. LMS2 also wants to know about the success or failure of the simulation of its student, and therefore it is also interested on receiving notifications about the events `event.actions.users.execution.success` and `event.actions.users.execution.failure`.

Unlike what happens with University 1, the configuration of the corporative network of University 2 does allow inbound connections from Internet. This fact is exploited by LMS2, which allows the simulator to actively notify it when a relevant event takes place. Therefore, LMS2 subscribes to the abovementioned event types and closes the connection with the tool.

Now it is the student from LMS2 who carries out a successful simulation, and so the simulator opens a new connection with LMS2 on TCP port 8080, sends the notification and closes the connection again. The student carries out another simulation, this time unsuccessful, which causes the simulator to open another connection with the on the same port, sends the notification, and closes the connection.

Meanwhile, LMS1 keeps on polling the simulator periodically in case that new events have been triggered by its students. Despite the students of LMS2 has also launched simulations, the tool only reports LMS1 about events regarding the instance of its own students.

The tasks of the students of LMS1 go on until the didactic unit reaches a point when LMS1 wants to collect information about the individual work of each student separately, and wishes to know how many simulations have been run by each of them. Therefore, in its periodical polling, instead of requesting information about the events that take place at the instance LMS1 also specifies the student that triggered the event.

At the end of the learning unit the students of both LMSs exit the tool, their instances are deleted, and the information retrieved as events is used to elaborate statistics and carry out a close study of the performance of the students.

The Tool. The fluid simulator receives two requests to create instances, one from LMS1 and another from LMS2. Two users are assigned to the instance of LMS1, who start running simulations using the graphical user interface of the tool.

During their interaction with the simulator users carry out several actions, including simulations, that the tool keeps registry about by storing relevant information (action performed, time, etc.). Next, LMS1 opens a connection with the tool requesting information regarding when successful simulations have been run, and keeps it open waiting for the reply of the tool. The latter access the requested information, serializes it in a format that LMS1 can understand, sends it, and closes the connection.

On the other hand, LMS2 grants access to the second instance of tool to its user, and next the simulator receives an incoming request from LMS2 requesting a notification every time a simulation is launched, be it successful or not. LMS2 does not wait for a response, but tells the tool to notify it on the TCP port 8080 every time a new simulation is launched, and closes the connection.

The student of LMS2 starts running simulations, and the tool operates as described before. The learning units of LMS1 and LMS2 reach the end and the two instances are deleted, which implicitly cancels the subscription to events of LMS2.

4. REQUIREMENTS

The previous use case allows us to derive a list of formal requirements that must be fulfilled by any event notification technologies for e-Learning.

Requirement 1: Interoperability. The LMS must be able to interoperate with a Web tool even if they are in different network domains. This is necessary because in general the Web tool is not managed by the same entity of the LMS, and hence it is located at a different network domain. This requirement is implicit in the use case above.

Requirement 2: Ad-hoc events. The need for ad-hoc events arises from the fact that distributed e-Learning environments try to provide the same functionalities of centralized ones, but in a more scalable way. Given that in centralized environments the LMS core can perform an in-depth tracking of the operation of the users with a tool, the only way to perform such tracking in distributed systems is to notify events whose nature depends of the kind of tool being operated. As a counterpart, we have generic navigation events, which only provide information concerning the navigation of the user through the different parts of the tool. The use of ad-hoc events instead of generic navigation events is a requirement which is addressed in the use case of Section 3.

Requirement 3: Primitive events. At the time of choosing what kind of events (primitive or composite) will be triggered by the tool, it is important to discern the events triggered by it from the composition of such events. Primitive events provide objective information about the operation of the user with a tool (in use case of Section 3 the simulator reports the primitive events `event.actions.users.execution.success` and `event.actions.users.execution.failure`). A different issue is their composition, which has the purpose of interpreting them according to the course planning or the rules established by the teacher (e.g. if many `event.actions.users.execution.failure` events are generated by the same user, it can be interpreted as the composite event "The user has conceptual problems"). Therefore, provided that this is something that concerns just the LMS, the tool does not have to worry about what kinds of composition will be done, if any. In this case, primitive events are the preferable choice.

Requirement 4: Vocabulary independence. The solution should be agnostic in terms of the vocabulary used to name events. This would allow, for example, to use a different (but similar) tool while keeping the same vocabulary and, on the contrary, to use the same tool changing the vocabulary if required.

Requirement 5: Publisher–Subscriber architecture. The scenario depicted in Section 2.3 describes a single publisher (the tool) and a single subscriber (the LMS). Therefore, a Publisher–Subscriber architecture is more suitable than a P2P one.

Requirement 6: Push-pull notifications. Push notifications are ideally more preferable than pull notifications for a couple of reasons. Nevertheless in several scenarios they may not be feasible due to several network restrictions. Therefore, the LMS should be able to choose the kind of notifications that matches best the characteristics of the network. This need is addressed in the use case of Section 3.

Requirement 7: Brokerage. The work load of a tool can be alleviated by delegating the notification process itself to a dedicated broker. This is specially in cases like the one depicted in Section 3. Notice how many notifications and accesses to databases are required in a scenario involving just two LMSs, two instances, two event types and three users. In high performance external tools, with lots of users and subscribers, the work load can be unfeasible unless there is a dedicated agent responsible for the notifications (i.e. a broker).

Requirement 8: Plain events submission. The fact that the use case of Section 3 involves different LMSs, prob-

ably developed in different programming languages, stresses the importance of choosing a neutral format to serialize and submit notifications by the simulator.

Requirement 9: Negotiable transport protocol. Given the potential diversity of tools that may be integrated by the LMS it is not sensible to pre-establish whether or not notifications must take place in real time or elastic time. This fact, together with the potential packet filtering carried out by intermediate routers, suggest that a more convenient approach is to negotiate the transport protocol and the port number that will be used to send notifications.

Requirement 10: Instance orientation. All events triggered by a tool are generated by (or due to) actions that take place at the core of one of its instances. Recall that an instance of a tool is a standalone working environment, together with a set of data elements and a group of users allowed to access it, and one instance is disjoint from other instances. Therefore, event notifications must refer to the instance they were triggered.

5. A SOLUTION FOR THE NOTIFICATION OF EVENTS IN E-LEARNING

With ELENA (E-Learning Event Notification Architecture) we support the subscription and notification of events between LMSs and exernal tools. Its operation is based on rssCloud, but featuring some characteristics from other technologies.

In this section we provide an in-depth description of ELENA. In the next Section we give a high-level description of the system, giving an overview of the static architecture and defining the actors involved. Section 5.2 describes the system from the point of view of the sequence of messages exchanged between the several actors. Section 5.3 describes the fields and format of the messages exchanged.

5.1 General Architecture

The LMS core and the tool are glued up by means of the Tool Binding Adapter and the Primitive Events Vocabulary. This section is devoted to describe these four elements. The result is depicted in the UML component diagram of Figure 3.

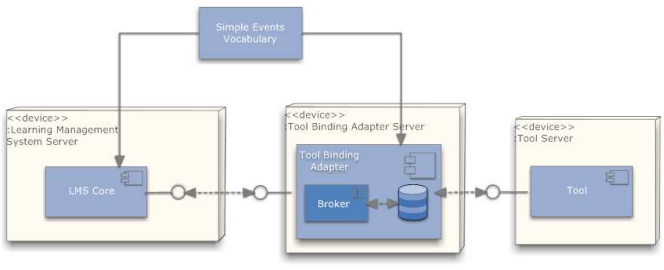

Figure 3: General architecture of ELENA.

The LMS Core. The LMS Core is the central element of the e-Learning system. It stores information related to educational scenarios, such as participants, learning goals, temporal constraints and so on [16]. Additionally, the LMS Core makes the state of educational scenario instances evolve depending on the events received (both from the Tool and internal to the LMS Core). As a consequence, users may be assigned new tasks to attempt after the completion (or attempt) of the previous ones.

The LMS Core may optionally feature an Event Composer. The Event Composer processes and interprets the events received from the Tool, in order to provide the LMS Core with higher-level information. A typical example of the Event Composer in action would take place if the Tool, an Electromagnetism simulator, triggers three consecutive "The user tried to run a simulation without setting the boundary conditions". The first two notifications would bypass the Event Composer and be reported to the LMS Core. However, at the third notification the Event Composer may interpret that the user has problems in understanding the underlying theory, and report to the LMS Core the "The user needs assistance on differential equations".

The Tool. The Tool is the component that provides some functionality that complements those of the LMS Core. We can consider, for example, tools such as calendars, wikis, media players, simulators, virtual world or forums, and therefore the specific characteristics of the tool may differ from one case to another. However, for the purposes of this paper we assume that the tool is such that it is in its nature to notify events. At this point two possibilities arise:

1. The tool has been developed in such a way that it notifies events by default.
2. The tool did not notify events by default, but is has been slightly modified to make it possible.

The Tool Binding Adapter. The Tool Binding Adapter (hosted at the Tool Binding Adapter Server) has many functions in the system. Firstly, it provides the LMS Core with a unified set of methods called the Generic Tool Interface. These methods allow the LMS to control issues of the several tools such as their instances, the assignment of permissions, the transfer of data, the authentication of their users, functions which are specific from each tool, and the issue covered in this article, the subscription and notification of events. Table 1 provides a summary of the methods of the Generic Tool Interface related to the management of events.

Secondly, it adapts the calls to the methods of the Generic Tool Interface into methods of the API of the Tool itself. The reason of the existence of the Tool Binding Adapter is that, while the Generic Tool Interface has been designed for general-purpose tools (featuring generic methods such as `createInstance()`) the API of the tool features a specific syntax (e.g. `newCalendar()` in the case of a calendar tool), and therefore a conversion must be carried out. In many cases there is a one-to-one correspondence between a method of the Generic Tool Interface and a method of the tool, because the former has been created keeping in mind

Table 1: Summary of the methods of the Tool Binding Adapter to manage events.

Method	Input parameters	Output parameters	Description
`subscribe`	`eventType`, `instanceURI`, `username*`, `protocol`, `port`	`result`, `subscriptionId`	Subscribe to all the events of the type `eventType` that are triggered within the instance with URI `instanceURI`. If the parameter `user` is present, the only events notified are those which were triggered by that user. The parameters `protocol` and `port` indicate the protocol and port number used to send notifications to the subscriber. Returns an error code, if any, and an identifier of the subscription for future references.
`unsubscribe`	`subscriptionId`	`result`	Cancels a previous subscription given by the identifier `subscriptionId`. Returns an error code, if any.
`getEventsSince`	`eventType`, `time`, `instanceURI`, `username*`	`result`	Requests a list of all the events of the type `eventType` that took place since the time `time` in the instance whose URI is `instanceURI`. If the parameter `username` is present, the only events notified are those which were triggered by that user. Returns an error code, if any.

Those parameters marked with the * symbol are optional.

that it should fit the characteristics of (ideally) any kind of tools. The output of the Tool Binding Adapter is a request that can be appropriately processed by the Tool API.

Finally, the Tool Binding Adapter also features broker functionalities. It is the only agent directly notified by the Tool, which delegates on it to carry out the notifications to all the interested subscribers. To do so, it keeps a registry (represented in Figure 3 as a database) of which subscribers showed interest on which topics, and notifies them in case that some event meeting these characteristics are triggered. This registry also allows the Tool Binding Adapter to cach all the events and do pull notifications without even polling the Tool.

The Primitive Events Vocabulary. The Primitive Events Vocabulary categorizes all the possible events that can be triggered by the Tool. Optionally, and depending on the format used to carry out this categorization (e.g. if an ontology is used), the Primitive Events Vocabulary may include relationships among these event types. In this paper we assume that the particular vocabulary has been previously negotiated between the LMS Core and the Tool Binding Adapter using some protocol for that purpose.

It is important to mention that an automated classification of all the events triggered by the Tool in the categories defined by the Primitive Events Vocabulary exceeds by far the scope of this article. Therefore, the developers or administrators of the Tool Binding Adapter must carry out a previous manual classification for each of the vocabularies they wish to support.

5.2 Dynamic Behaviour

After the static description of the system, in this section we describe the dynamic issues of the system.

Starting with Figure 4, we describe the actions of the several roles involved to make notifications following the pull approach. The process starts when the tool triggers some events. The Tool Binding Adapter, due to its broker capabilities, is the only recipient of the notifications. From this point, the Tool Binding Adapter stores the notifications using some storage solution (e.g. a database) and waits for incoming requests from interested users.

Later, the LMS Core invokes the `getEventsSince()` method of the API of the Tool Binding Adapter with the parameters specified in Table 1.

After the appropriate queries to the appropriate databases, the Tool Binding Adapter gathers all the events with such characteristics, serializes them, and packs them in the payload of a HTTP message (see Section 5.3 for more details).

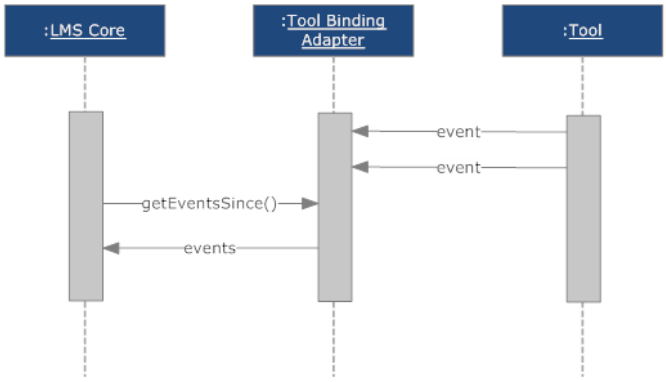

Figure 4: UML sequence diagram of the process to support pull notifications.

As for push notifications, the process is summarized in Figure 5. In this case, the choreography is initiated by the LMS Core, which shows its interest in knowing about some kind of events by invoking the `subscribe()` method of the Tool Binding Adapter. The input parameters of this method have been detailed in Table 1.

The Tool Binding Adapter keeps a registry of all the interested subscribers and, when the appropriate event has been triggered, notifies them encoding the notification in the payload of a HTTP message following the same format of pull notifications.

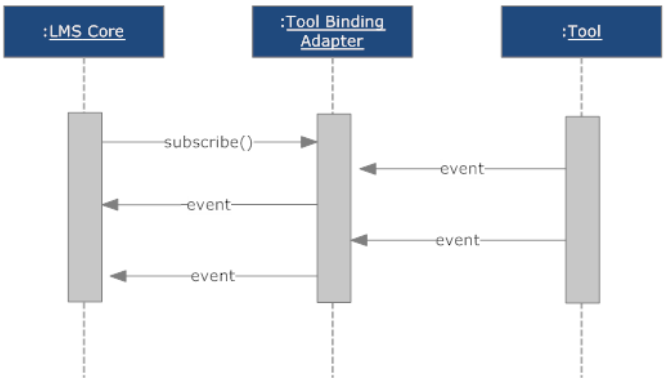

Figure 5: UML sequence diagram of the process to support push notifications.

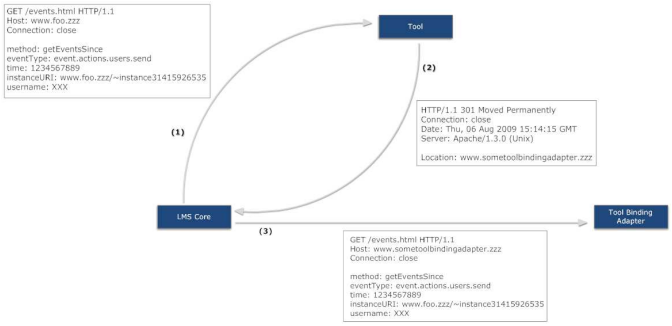

Figure 6: Process to request events in ELENA using pull notifications.

5.3 Message Format

The ELENA message format has been designed as a simple, fast and interoperable way to request and notify events between the LMS Core and the Tool Binding Adapter. It consists of the exchange of standard HTTP messages with special header extensions and a special structure for their payload in order to request and notify events.

There are basically two types of messages, depending on whether the information is encoded in the header or in the payload. In this section we depict a simple (but complete) scenario involving a pull request followed by a reply. An example featuring a push notification would be exactly the same, with the difference of the method invoked.

The process begins when the LMS Core opens a connection with the Tool and invokes the `getEventsSince()` method. Both the method name and its input parameters are encoded in the header of the HTTP message. The purpose of the Tool is that the Tool Binding Adapter handles all the subscriptions and notifications, and therefore replies with a HTTP message with status code 301 (`Moved Permanently`) and a `Location` header with the URI of the Tool Binding Adapter. The LMS Core resends the original message to the Tool Binding Adapter, concluding the process to request events. The structure of these three messages is summarized in Figure 6. It is important to say that the first two messages are just a redirection to the Tool Binding Adapter. For future requests they are not necessary because the LMS Core already knows about its existence as a component where all the requests must be forwarded to. Also notice the `Connection: close` header in message number three, which indicates that when the Tool Binding Adapter has replied with the message containing the requested events the connection will be closed. This particular issue differs with regards to push notifications, which use the `Connection: open` HTTP header to keep the connection alive for the notification of future events.

The Tool Binding Adapter makes the appropriate queries and, when it has retrieved all the events matching the characteristics of the request, serializes them in the payload of the HTTP message that is sent back in response. For each event notified several fields are specified:

- `Subject`: the user that originated the event, given by a unique username. Optional parameter.
- `Event-type`: the action that triggered the notification. The particular nomenclature is given by the Primitive Event Vocabulary (see Section 5.1). Mandatory parameter.
- `Object`: the user (given by a unique username) or data element (given by its URI) that suffers the action. Optional parameter.
- `To-whom`: the beneficiary (a user given by a unique username, or a data element given by its URI) of the action. Optional parameter.
- `Time`: the time when the action took place, in POSIX format. Mandatory parameter.
- `Instance`: the instance where the action took place, given by its URI. Mandatory parameter.
- `Using`: the data element or functionality (given by their URI) that was used by the `Subject` in order to carry out the action. Optional parameter.
- `Cause`: indicates some action whose consequence was the action denoted by `Event-type`. Optional parameter.
- `Other`: reserved for future use.

This notation provides a good support to encode any kind of information regarding what happened at the tool [1], while at the same time being compatible with any notation specified in the Primitive Events Vocabulary. Using this notation it is possible to send complex notifications such as “User XXX (`Subject`) sent (`Event-type`) the document www.foo.zzz/ instanceURI/doc.xml (`Object`) to user YYY (`To-whom`) in the context of instance www.foo.zzz/ instanceURI (`Instance`) in March 14th at 12:34 (`Time`), after its automatic generation by a XML generator service (`Cause`), using the messaging service www.foo.zzz/ instanceURI/fileexchange (`Using`).”.

The abovementioned fields are the information about a single event, and is copied line by line in plaintext in the payload of the HTTP message. If more than one event is notified, the information of the several events is concatenated

with the `AND` word. Optionally the payload can be compressed using the zip algorithm, indicating this by the use of the `Content-Type: application/zip` header of the message. Figure 7 gives an example of a message following this structure.

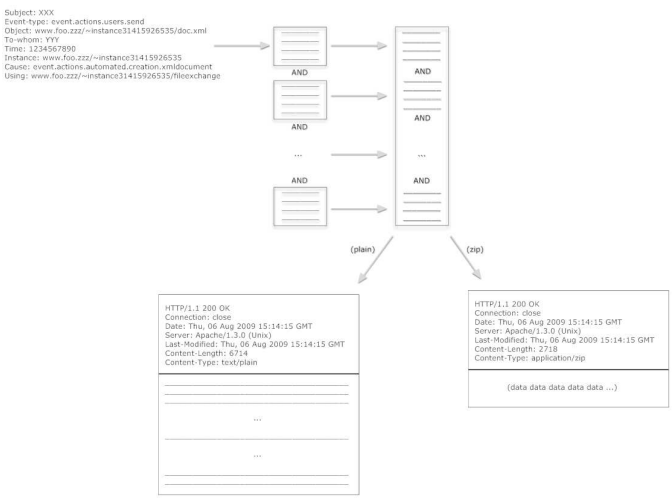

Figure 7: Event notification message.

6. CONCLUSION

Current LMSs are playing an important role in providing access to educational contents all around the world, avoiding spatial and temporal barriers. However, their possibilities are limited due a clear "one size doesn't fit all" problem. These limitations have been the starting point of our research. The work described in this paper tries to identify mechanisms to tackle these issues with tailorability and extensibility in mind by the use of the SOA paradigm, and propose a solution to one of them: the notification of events.

The concept of hard integration implies an evolution to current integration mechanisms, and is at the heart of our research. It allows an LMS to take advantage of external tools, and work in coordination with them as if they were local plugins. As a result, the LMS can extend its functionality, and can do it in a controlled way. Many of the design principles of ELENA are based on this idea.

Currently we are working on giving full support to the other aspects of hard integration, apart from the notification of events. Nonetheless, the six parts we have considered are completely separate from each other, and they can be used in a standalone way if desired.

We would be remiss if we did not make some comments on two important issues. The first one is the negotiation of the Primitive Events Vocabulary, which we have deliberately omitted. During this paper we have assumed that a previous negotiation process has taken place between the LMS and the tool, with the result of a commonly agreed vocabulary for future transactions. We have designed ELENA so that it is agnostic in terms on how the negotiation is carried out, for the sake of keeping the protocol simple. However, such negotiation is necessary, and therefore ELENA should be complemented with another protocol. ELENA does not impose any restrictions on the characteristics of this protocol, as long as the result is a shared vocabulary.

The other aspect we have omitted is privacy, which has to be addressed from two complementary points of view. Firstly, one may wonder whether it is possible for a fraudulent LMS to subscribe to events from any instance of a given tool. The approach we follow is to make the legitimate subscriber the only one that knows the URI of the instance, understanding by "legitimate subscriber" the one that created the instance [14]. Finally, eavesdropping should also be taken to account depending on the kind of tool and the information that is notified. Again, ELENA is deliberately neutral and can be used over SSL if desired.

7. ACKNOWLEDGMENTS

Authors want to thank Spanish Ministerio de Ciencia e Innovacion for its partial support to this work under grant "Methodologies, Architectures and Standards for adaptive and accessible e-learning (Adapt2Learn)" (TIN2010-21735-C02-01) and to CYTED Program under Coordinating Action 508AC0341 SOLITE.

8. REFERENCES

[1] *English Syntax*. The MIT Press, 1995.
[2] *Computers and Design in Context*. The MIT Press, 1997.
[3] Survey of learning-related services, April 2010.
[4] Blackboard Web site, March 2011.
[5] E-Learning Framework home page, March 2011.
[6] Five benefits of software as a service, March 2011.
[7] IMS Tools Interoperability specification, March 2011.
[8] Mashups: The new breed of Web app, March 2011.
[9] Microsoft patterns and practices. publish/subscribe, March 2011.
[10] Moodle modules and extensions, March 2011.
[11] Moodle Web site, March 2011.
[12] F. Coyle. *XML, Web Services and the data revolution*. Addison-Wesley Professional, 2002.
[13] D. Dagger et al. Service-oriented e-learning platforms: From monolithic systems to flexible services. *IEEE Internet Computing*, 11(3):28–35, 2007.
[14] J. Fontenla et al. A Middleware for the Integration of Third-Party Learning Tools in SOA-Based Learning Management Systems. Supporting Instance Management and Data Transfer. *Proceedings of EDUCON 2010*, 2010.
[15] G. Gross et al. Google, ibm promote cloud computing. *PC World*, 2007.
[16] R. Perez-Rodriguez et al. Design of a Flexible and Adaptable LMS Engine in Conformance with PoEML. *International Journal of Emerging Technologies in Learning (iJET)*, 4(0), 2009.
[17] M. Roqueta. Learning Management Systems. A focus on the learner. *Distance Learning*, 5(4).