Modeling data-intensive Rich Internet Applications with server push support *

Giovanni Toffetti Carughi

Politecnico di Milano,
Dipartimento di Elettronica e Informazione,
Via Giuseppe Ponzio, 34/5 - 20133 Milano - Italy
giovanni.toffetti@polimi.it

Abstract. Rich Internet applications (RIAs) enable novel usage scenarios by overcoming the traditional paradigms of Web interaction. Conventional Web applications can be seen as reactive systems in which events are 1) produced by the user acting upon the browser HTML interface, and 2) processed by the server. In RIAs, distribution of data and computation across the client and the server broadens the classes and features of the produced events as they can originate, be detected, notified, and processed in a variety of ways. Server push technologies allow to get over the Web "pull" paradigm, providing the base for a wide spectrum of new browser-accessible collaborative on-line applications. In this work, we investigate how events can be explicitly described and coupled to the other concepts of a Web modeling notation in order to specify server push-enabled Web applications.

1 Introduction

Rich Internet Applications (RIAs) are fostering the growth of a new generation of usable, performant, reactive Web applications. Whilst RIAs add complexity to the already challenging task of Web development, among the most relevant reasons for their increasing adoption we can consider: 1) their powerful, desktop-like interfaces; 2) their accessibility from everywhere (along with the fact that final users tend to avoid installing software if a Web-accessible version exists); 3) the novel support they offer for on-line collaborative work. This last aspect is based on getting over the limits of Internet standards to provide server push techniques, enabling applications such as instant messaging, monitoring, collaborative editing, and dashboards to be run in a Web browser.

In this paper we focus on push-enabled Rich Internet Applications and the lack of existing modeling methodologies catering for their specific features. The most significant contributions of this work are:

1. the extension of a Web engineering language to consider collaborative Web applications using push technologies (Section 3) through the individuation

* We wish to thank Alessandro Bozzon, Sara Comai, and Piero Fraternali for the precious help and insightful discussions on this work.

of a set of primitives and patterns (Section 4) catering for the different aspects of distributed communication such as (a)synchronicity, persistence, and message filtering. We stress that the extensions we propose are general and can be applied to the most relevant Web engineering approaches;
2. a validation by implementation of the proposed concepts and notations (Section 5).

1.1 Running example

To ease the exposition we will use an example: as a case study, we consider a collaborative on-line application for project management. The aim of the application is to serve as an internal platform to let users communicate and organize projects and tasks. The purpose is to have a simple but consistent example we can use throughout the different sections: we will keep it as naive as possible so as to avoid cluttering diagrams with unnecessary detail, but the concepts we will introduce are general and can be applied to complex industrial scenarios.

Application users impersonate different roles: project managers and project participants. A project manager is responsible of creating projects and connecting them to their participants. For each project, tasks are created, precedence relationships among them are defined, and they are assigned to project participants for execution. Project participants can exchange messages with their contacts, while performing assigned tasks and signaling their completion.

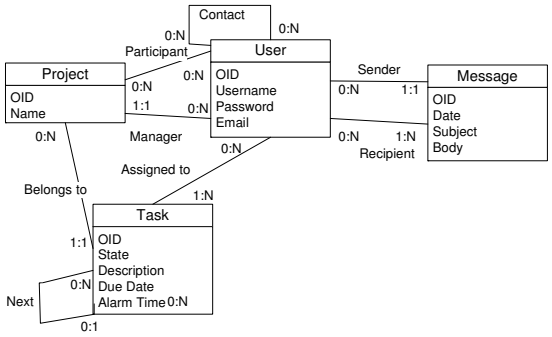

Fig. 1. Data Model of the project management application

Figure 1 shows the data model for the collaborative on-line application: the *user* entity represents all application users, a self-relationship connects each user with his contact list. A single user can participate in one or more *projects*, while each project can be directed by a unique manager. Users are assigned *tasks*: each task belongs to a project and can have precedence constraints w.r.t. other tasks. *Messages* can be exchanged between users, they have a sender and a set of recipients.

2 Background

RIAs extend the traditional Web architecture by moving part of the application data and computation logic from the server to the client. The aim is to provide more reactive user interfaces by bringing the application controller closer to the final user, minimizing server round-trips and full page refreshes.

The general architecture of a RIA is shown in Figure 2: the system is composed of a (possibly replicated) Web server[1] and a set of *user applications* (implemented as JavaScript, Flash animations, or applets) running in browsers. The latter are downloaded from the server upon the first request and executed following the *code on demand* paradigm of code mobility [7].

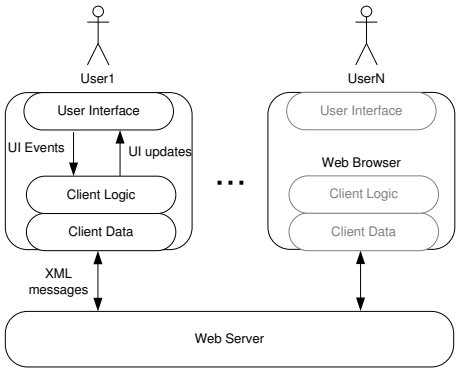

Fig. 2. A Rich Internet Application architecture

Each user interacts with his own application instance: as part of the computation is performed on the client, communication with the server is kept down to the minimum, either to send or receive some data, generally in XML.

Server-push and communication The communication between client and server is bidirectional in the sense that, after the first client request, messages can also be pushed from the server to the client. This is technically achievable because of the novel system architecture including client-side executed logic.

In a "traditional" Web application the HTML interface is computed by the server at each user's request. When the user interacts with a link or a button on the page the server is invoked and a new page is computed and sent back to the client. The role of the browser is simply to intercept the user's actions, deliver the request to the server, and display a whole new interface even if the change is minimal.

[1] We abstract from the server-side internal components as they are not relevant for the discussion

In a RIA instead, the client-side logic handles each user interaction updating interface sub-elements only. Furthermore, when client-server communication is needed to transfer some data, it can be performed in the background, asynchronously, allowing continuous user interaction with the interface. This, together with HTTP/1.1 persistent connections, are the key ingredients of a technique called *HTTP trickling*, one of the solutions enabling servers to initiate server-to-client communication (server-push) once a first HTTP request is performed. The programming technique for server-push is also known as "Comet", relevant implementations are Cometd[2], Glassfish[3], and Pushlets[4], but most Rich Internet Application development frameworks provide their own.

A user can therefore be notified of events occurring *outside* of his application instance, either other users actions or in general occurring on the server. Direct communication between client applications is in general not possible[5], but the server can be used as an intermediary (i.e., a *broker*).

Most on-line applications (especially collaborative) can benefit from this approach: think for example of work-flow-driven applications, shared calendars or whiteboards, stock trading platforms, plant monitoring applications, and so on. Interaction of other users, server internal or temporal events, Web service invocations can all be occurrences that can trigger a reaction on a user client application.

Considering the project management case study, server push can be used for instant messaging, or to signal task assignments and task completions in order to immediately start the execution of new tasks.

2.1 Problem statement

As Rich Internet Application adoption is experiencing constant growth, a multitude of programming frameworks have been proposed to ease their development. While frameworks increase developer productivity, they fail in providing instruments to abstract from implementation details and provide an high level representation of the final software; this becomes necessary when tackling the complexity of large, data-intensive applications.

While Web engineering methodologies offer sound solutions to model and generate code for traditional Web applications, they generally lack the concepts to address Rich Internet Application development. In a previous work [4] we suggested an approach to tackle these shortcomings concerning data and computation distribution among components; here, we focus on another essential

[2] http://cometd.com/
[3] http://glassfish.dev.java.net/
[4] http://www.pushlets.com/
[5] Web clients are in general not addressable from outside their local area network; in addition most browser plug-ins run in *sandboxes* preventing them to open connections to other machines than the code originating server. For this reason RIAs only allow direct communication between clients and the server; no direct communication can take place between client instances.

aspect of RIAs: server push and the new interaction paradigms and application functionalities it enables.

3 Approach overview

In a traditional Web application, data and computation reside solely on the server. Thus, the whole application state, and all actions upon it, are concentrated in a single component.

In a RIA, distribution of data and computation across the server and different clients causes:

- user actions to be performed asynchronously w.r.t. the server and other user applications;
- client-side data for each user and server-side data to evolve independently from each other.

In order to achieve better reactivity client-server communication in RIAs is reduced to the minimum: therefore a mechanism is needed to signal relevant events, either to reduce the misalignment between replicated and distributed data, or simply to signal that, at a specific instant, an action is being performed in the distributed system. The classes of actions that are relevant for a system are application-specific.

Example For instance, considering our running example, each action upon a task instance can be considered a specific *event type*, we have event types:

Task assigned: when a project manager performs the action of assigning a task to a specific project participant;
Task completed: when a project participant marks one of her assigned tasks "completed".

Both actions can be signaled to application users to begin working on the associated or next tasks.

3.1 Notification communication

Considering that in a RIA all communication must go through the server (as described in Section 2), only two scenarios apply:

1. An event occurring on a client application has to be notified. In this case the notification starts from a client application, goes to the server, and is eventually delivered to other users;
2. An event occurring on the server has to be notified. In this case only server to client communication takes place.

Different aspects can influence the process of communicating event notifications across the system components: for instance how to identify notification recipients, or whether the communication happens synchronously or not. We

Dimension Name	Values
Filter location	Sender, Broker, Recipient
Filter logic	Static, Rule-based, Query-based
Communication Persistence	Persistent (asynchronous), Transient (synchronous)

Table 1. Semantic dimensions in RIA event notification

call these aspects semantic dimensions, they are listed in Table 1. Semantic dimension identification is necessary in order to correctly draw the primitives and devise the appropriate patterns covering all their possible combinations. In the following paragraphs we give a brief definition of each aspect.

Event filtering: location and logic. Not all users need to be notified of all event occurrences; in distributed event systems the process of defining the set of recipients is generally indicated as *event filtering* [16] and two dimensions can influence it: *where* it takes place and *how*.

The former dimension considers the most general architecture for publish / subscribe systems [12] that involves three kinds of actors: a set of publishers, a broker, and a set of subscribers. Events occur at publishers that alert the broker who, in its turn, notifies the subscribers. Thus, the decision of which subscribers to notify can be taken at the publisher, at the broker, or all subscribers can be notified leaving to them the decision of whether to discard or not an already transmitted notification.

The latter dimension considers the logic that is used to determine notification recipients: the spectrum of possibilities ranges from statically defined recipients, to conditions on event attributes, to interpreted run-time defined rules (e.g., using a rule engine to detect composite event patterns).

Communication persistence. In distributed systems message communication can be [20]:

- *Persistent*: a message that has been submitted for transmission is stored by the communication system as long as it takes to deliver it to the recipient. It is therefore not necessary for the sending application to continue execution after submitting the message. Likewise, the receiving application need not be executing while the message is submitted;
- *Transient*: a message is stored by the communication system only as long as the sending and receiving applications are executing. Therefore the recipient application receives a message only if it is executing, otherwise the message is lost.

Depending on the application, some events may need to be communicated in a persistent way to prevent their loss, others only need transient communication. For example, email transmission uses persistent communication: the message is accepted by the delivery system and stored until it is deleted by the recipient;

the sender completes the communication as soon as the message is accepted by the delivery system.

Transient communication, instead, does not store the message and requires both sender and recipient to be running and on-line: it is often used when the loss of some event notification is not critical. For instance, in a content management system the event that two users are trying to edit the same resource is important for the colliding users, but if one of them disconnects before receiving the notification it's no use storing it persistently.

4 Proposed Extensions

In this section we present the extensions we propose to cover all the combinations of the previously introduced communication aspects. We will illustrate them in WebML [9], although we stress that they apply in general to most Web engineering languages. First we will consider the data model, then the navigation model.

4.1 Extension to the data model

Application-specific event types are represented by adding new entities to the data model organised in a specialisation hierarchy. All event types extend a predefined *Event* entity and can have relationships with application domain entities.

We chose to extend the existing data model instead of adding a new event model so that we could leverage existing CRUD[6] WebML operations leaving full control to the application designer to:

- enable persistent communication by directly storing event occurrences in the database (or client-side storage);
- represent and instantiate relations between event occurrences and domain model entities;
- instantiate an event base and explicitly draw ECA rules with provision for specific dimensions such as granularity, composite event detection, and event consumption [13].

Example Considering the project management example: the event types "Task assigned", and "Task completed" apply. They are represented in the event hierarchy in Figure 3. The WebML data model of Figure 1 is extended with entities representing the needed event types that are connected by means of relations to the application domain entities. Thus, the events "Task assigned" and "Task completed" have a relation with the task to which they refer. In addition to being related to a task, the event "Task assigned" also has a relationship to the users to which the assignment was made.

[6] Create, Read, Update, Delete

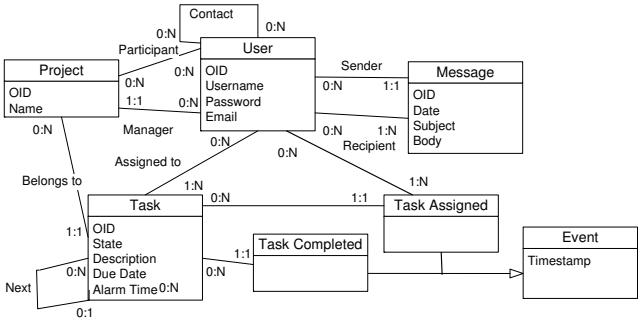

Fig. 3. The data model of Figure 1 extended with event types

4.2 Extension to the hypertext model

In order to support event notifications, we added to the WebML hypertext model two operations: *send event* and *receive event*. The former allows one to send an event notification to a (set of) recipient(s); the latter is used to receive the notification and trigger a reaction. Send and receive event operations allow (indirect) communication among users without the need to use data on the server. Each operation is associated with an event type as defined in the extended data model. The event type provides both mapping between send and receive event operations (i.e., operations on the same event type trigger each other) as well as their specific parameter set. Operations for conditional logic (e.g., switch-operation, test-operation, as introduced in [5]) or queries can be used in conjunction with event-related operations in order to specify event filtering: retrieving notification recipients from a database, or discarding notifications upon application-specific conditions. Different patterns apply, catering for the possible combinations of filtering and communication needs (see Section 3.1).

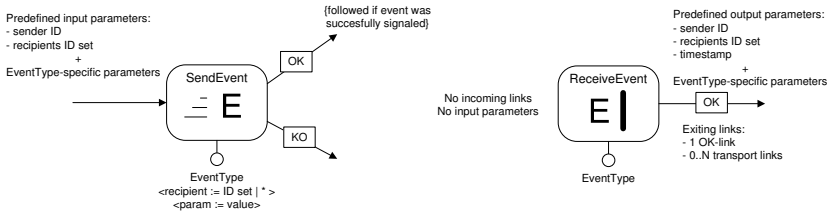

Fig. 4. On the left, Send Event operation usage. On the right a Receive Event operation

Send event operation A send event operation (Figure 4) triggers the *notification of an event*. It needs to be explicitly activated by a navigation link or

by an OK or KO-link. It is associated with an event type (see Section 4.1), and consumes the following predefined input parameters:

1. sender [optional]: the unique identifier of the sender of the event (e.g., a user ID, or the server)
2. recipient: the (set of) identifier(s) of the recipient(s) (e.g., user IDs, the server) . The '*' wild-card is used to indicate that the event is to be notified to all possible recipients.

Additional input parameters stem from the associated event type. Send event operations have no output parameters.

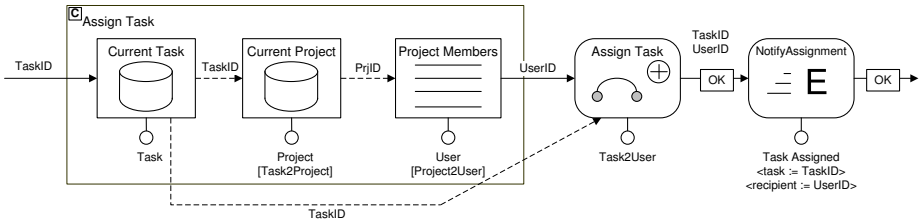

Fig. 5. The hypertext to assign a task and send a notification

Example Considering the project management case-study, the send event operation can be used to signal the assignment of a task to a project participant as in Figure 5. The data-unit *Current Task* is used to show the selected task to be assigned, *Current Project* provides the current project identifier to show project participants in the *Project Members* index-unit. Selecting one, the *Assign Task* connect-operation is invoked to instantiate a relationship between the current task (with ID *TaskID*) and the selected user (with ID *UserID*): the same IDs are provided to the *NotifyAssignment* send event operation to define the notification recipient and to set the notification parameter identifying the assigned task.

Receive event operation A receive event operation (Figure 4) is an *event notification listener*: it is triggered when a notification concerning the associated event type is received. It is therefore associated with an event type, it has no input parameters, and the following predefined output parameters:

1. sender: the unique identifier of the sender of the event
2. recipient: the id set of recipients of the notification
3. timestamp: the timestamp at server when the event was signaled

In addition to the predefined output parameters, a receive event operation also exposes the parameters of the associated event type to be used by other units (e.g., for condition evaluation). Receive event operations only have exiting transport links or OK-links and cannot have incoming links.

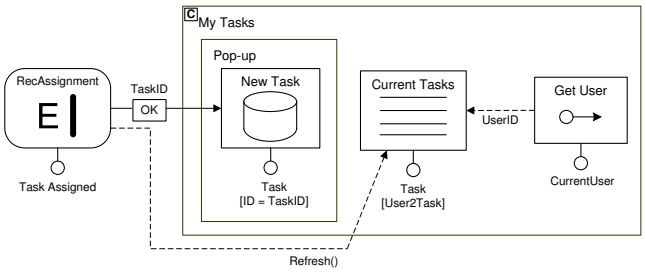

Fig. 6. A notification is received and the user interface is updated

Example Figure 6 shows the hypertext model of a page receiving a task assignment notification. Page *My Tasks* shows a list of task assigned to the current user with index-unit *Current Tasks*. It is a RIA page, marked with 'C' in the upper left corner, to specify that the page contains client-side executed logic. Thus, the page can establish a persistent connection with the server and receive notifications: when a *Task Assigned* notification is received, the *RecAssignment* receive event operation is triggered. Upon activation, the unit passes the *TaskID* parameter to the *New Task* data-unit that will retrieve the task details from the server to be shown in a pop-up window; the *refresh()* signal on the transport link connecting *RecAssignment* with *Current Tasks* causes the latter to refresh its content to show the updated task assignments list.

A receive event operation can be placed both in a siteview[7] (starting an operation chain ending in a RIA page), or in a new diagram: the *event view*. Respectively the former solution specifies the reaction that will be performed by the client application if the notification recipient is on-line and viewing a determined page, the latter specifies what condition evaluation and actions will be performed when the server (which is supposed to be always on-line) receives the notification.

Event view The event view models the *server* reaction upon receipt of event notifications. Reactions to events are modeled by means of operation chains[8] starting with a receive-event-operation. They can trigger any server-side operation such as invoking Web services, sending emails, performing CRUD operations on the database, or signaling new event occurrences.

The event view:

– provides a mechanism for asynchronous or persistent communication, by letting a designer specify the server reaction to an event notification. This can include making the notification persistent using the database, to asynchronously signal the event when the intended recipient is back on-line;

[7] The WebML diagram representing the hypertext structure for an application user
[8] i.e. Sequences of WebML operations

– can be used together with conditional and query operations to specify *broker filtering* (Section 3.1) using server-side resources (e.g., databases, rule-engines, subscription conditions);
– allows reuse by factoring out operation chains triggered from different sources (e.g., different site views, areas, pages).

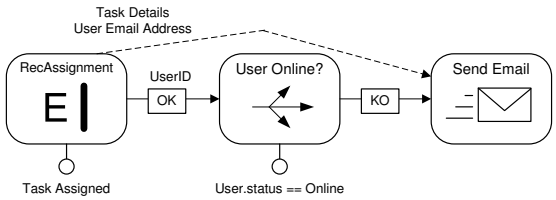

Fig. 7. If the assignment user is offline, send her an email

Example Figure 7 depicts an event-view operation chain for the running example. The application requires that a notification be sent to a user when a task is assigned to her. When the user is off-line, she cannot receive notifications with server push: on the server, the *User Online?* switch-operation triggers a send-email-operation if the recipient is not immediately reachable.

5 Considerations and experience

The primitives and models we propose provide a simple notation for the specification of server push-enabled Rich Internet Applications. They have been designed so that the combination with WebML primitives enables the complete coverage of the semantic dimensions space individuated in Section 3.

– Receive-event-operations used in conjunction with switch-operations provide a mechanism to draw Event-Condition-Action rules distributed both on the server (event view) and client (siteview) applications. This caters for concerns such as filtering at recipient and broker, as well as detecting composite events[9].
– Send-event operations in concert with query-based units allow the design of different communication paradigms including publish-subscribe, uni-, multi-, and broadcast. WebML units such as the selector unit can be used to cater for both static and query-based filtering to select event notification recipients.
– The event view lets a designer specify the behaviour on the server upon event notifications in order to provide support for asynchronous communication and event persistence.

[9] e.g. by storing event occurrences in an event base and considering conditions querying it upon event occurrences

The same primitives can be used to represent other classes of system events such as, for instance, temporal events, external database updates, or Web service invocations. We have implemented the integration of such events in our prototype, but, due to space reasons, we do not discuss them here[10]. Concerning database updates, our solution was inspired by the work in [22].

5.1 Implementation

The implementation of the presented concepts builds on the runtime architecture for WebRatio we developed for our previous work presented in [4]. We developed a working prototype of a code generator from the extended WebML notation to Rich Internet applications implemented using the OpenLaszlo [1] technology. To validate our proposal, we extended it with the needed components for server-push: the resulting architecture is shown in Figure 8.

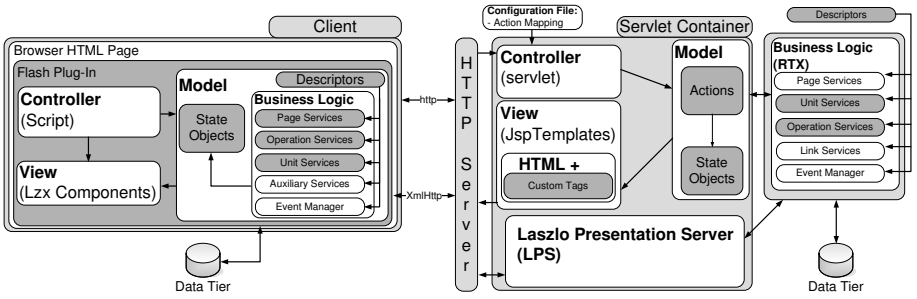

Fig. 8. The runtime architecture of our prototype implementation

OpenLaszlo provides natively the concept of a *persistent connection*[11] to enable server to client communication. Thus, the implementation of the receive event and send event operations on the client-side are quite straightforward. An event handler (*Event Manager*) is triggered whenever the *Dataset*[12] associated with the persistent connection receives a message (an event notification) from the server. The message is an XML snippet, its structure reflects that of the associated event type, each message carries its type information. The Event Manager checks the type information and triggers the appropriate receive event operation instance passing the pointer to the received message. The receive event operation extracts relevant parameters from the message, sets the values for its outgoing parameters and calls the next operation in chain.

A send event operation on the client builds, when triggered, an XML snippet reflecting the associated event type structure and whose attribute and text values

[10] A complete exposition can be found in [21]
[11] http://www.openlaszlo.org/lps/docs/guide/persistent_connection.html
[12] a data store of XML in OpenLaszlo

stem from the operation input parameters values upon invocation; then it invokes the *sendXML()* method of the persistent connection.

The server-side stub of the persistent connection is a servlet and thus accessible through a regular HTTP request. We extended it so as to be able to intercept event reception on the server and trigger the appropriate server-side operations in the event view (using the server-side Event Manager).

The code-generator was used to produce the running code of a personal information management application with features such as a shared calendar, and instant messaging.

6 Related work

Although RIA interfaces aim at resembling traditional desktop applications, RIAs are generally composed of task-specific client-run pages deeply integrated with the architecture of a traditional Web application. Web Engineering approaches build on Web architectural assumptions to provide simple but expressive notations enabling complete specifications for code generation. Several methodologies have been proposed in literature like, for example, Hera [23], OOHDM [19], WAE [10], UWE [14], and W2000 [3], but, to our knowledge, none of them addresses the design and development of RIAs with server push support.

This work, instead, considers system reaction to a collection of different events depending on *any* user interaction, Web service calls, temporal events, and data updates. Apart from defining the generic concept of event in a Rich Internet Application, our approach also individuates the primitives and patterns that can be used to notify and trigger reactions upon external events. This is something that, to our knowledge, all Web engineering approaches lack as reactions are only considered in the context of the typical HTTP request-response paradigm. Also, the approach we suggest provides the interfaces and notations necessary to integrate external rule engines (e.g., to detect composite events) with the Web application enabling the specification of reactions to events in terms of hypertext primitives (pages, content units, and operations).

Both a survey and a taxonomy of distributed events systems are given in [16]: most of the approaches bear the concepts individuated in [18] concerning event detection and notification, or in [12] w.r.t. the publish-subscribe paradigm. To cite but a few relevant works we can consider [2], [8], and [11]. Reactivity on the Web is considered for example in [6] where a set of desirable features for a language on reactive rules are indicated (the actual language is proposed in [17]).

With respect to these works, ours addresses events and notifications in a single *Web application* where on-line application users are the actors to be notified. In contrast, the above proposals aim at representing Internet-scale event phenomena with the traditional problems of wide distributed systems such as, for instance, clock synchronization and event ordering [15]. The system we are considering, instead, is both smaller in terms of nodes, and simpler in terms

of topology: the server acts as a centralized broker where all events are ordered according to occurrence or notification time. Nevertheless, it enables the implementation of complex collaborative on-line applications accessible with a browser.

7 Conclusions

Server push in RIAs provides the means to implement a new generation of on-line collaborative applications accessible from a Web browser. Although notification-based communication of events (e.g., publish / subscribe) and server-push technologies (e.g., based on AJAX) are well-known and established technologies, the explicit definition of the primitives supporting the modeling of Web applications employing said style of communication, and their introduction to modeling languages are new contributions. In this work we have presented the extension we propose to a Web Engineering language to represent the novel interaction paradigms enabled by Rich Internet application technologies. We considered the possible ways in which events occurring across different system components can be detected, notified, and processed in order to identify a set of simple, yet expressive, primitives and patterns.

References

1. OpenLaszlo. http://www.openlaszlo.org.
2. J. Bacon, J. Bates, R. Hayton, and K. Moody. Using events to build distributed applications, 1996.
3. L. Baresi, F. Garzotto, and P. Paolini. Extending UML for modeling Web applications, 2001.
4. A. Bozzon, S. Comai, P. Fraternali, and G. Toffetti Carughi. Conceptual modeling and code generation for Rich Internet Applications. In D. Wolber, N. Calder, C. Brooks, and A. Ginige, editors, *ICWE*, pages 353–360. ACM, 2006.
5. M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in Web applications. *ACM Transactions on Software Engineering and Methodology*, 2006.
6. F. Bry and M. Eckert. Twelve theses on reactive rules for the Web. In *Proceedings of Workshop "Reactivity on the Web" at the International Conference on Extending Database Technology , Munich, Germany (31st March 2006)*, LNCS, 2006.
7. A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA, 1997.
8. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
9. S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*, pages 137–157, Amsterdam, The Netherlands, 2000. North-Holland Publishing Co.

10. J. Conallen. *Building Web applications with UML, 2nd edition*. Addison Wesley, 2002.
11. G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS, 2001.
12. P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
13. P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. *ACM Trans. Database Syst.*, 20(4):414–471, 1995.
14. N. Koch and A. Kraus. The expressive power of UML-based Web engineering. In *Second Int. Workshop on Web-oriented Software Technology*. Springer Verlag, May 2002.
15. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
16. R. Meier and V. Cahill. Taxonomy of Distributed Event-Based Programming Systems. *The Computer Journal*, 48(5):602–626, 2005.
17. P.-L. Patranjan. *The Language XChange: A Declarative Approach to Reactivity on the Web*. PhD thesis, University of Munich, Germany, July 2005.
18. D. S. Rosenblum and A. L. Wolf. A design framework for Internet-scale event observation and notification. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 344–360. Springer–Verlag, 1997.
19. D. Schwabe, G. Rossi, and S. D. J. Barbosa. Systematic hypermedia application design with OOHDM. In *Hypertext*, pages 116–128. ACM, 1996.
20. A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
21. G. Toffetti Carughi. *Conceptual Modeling and Code Generation of Data-Intensive Rich Internet Applications*. PhD thesis, Politecnico di Milano, 2007.
22. L. Vargas, J. Bacon, and K. Moody. Integrating databases with publish/subscribe. In *ICDCS Workshops*, pages 392–397. IEEE Computer Society, 2005.
23. R. Vdovjak, F. Frasincar, G. Houben, and P. Barna. Engineering semantic Web information systems in Hera. *Journal of Web Engineering*, 2(1–2):3–26, 2003.