AN AGENT-BASED PROGRAMMING MODEL FOR DEVELOPING CLIENT-SIDE CONCURRENT WEB 2.0 APPLICATIONS

Giulio Piancastelli, Alessandro Ricci and Mattia Minotti
DEIS, ALMA MATER STUDIORUM—*Università di Bologna*

Keywords: Concurrent Programming, Agent-Oriented Programming, Web 2.0.

Abstract: Using the event-driven programming style of JavaScript to develop the concurrent and highly interactive client-side of Web 2.0 applications is showing more and more shortcomings in terms of engineering properties such as reusability and maintainability. Additional libraries, frameworks, and AJAX techniques do not help reduce the gap between the single-threaded JavaScript model and the concurrency needs of applications. We specialise the model in the context of client-side Web development, by characterising common domain agents and artifacts that form an extension of an existing programming framework. Finally, we design and implement a simple but significant case study to showcase the capabilities of the model and verify the feasibility of the technology.

1 INTRODUCTION

One of the most important features of the so-called Web 2.0 is a new interaction model between the client user interface of a Web browser and the server-side of the application.

Such *rich Web applications* allow the client to send multiple concurrent requests in an asynchronous way, avoiding complete page reload and keeping the user interface live and responding. Periodic activities within the client-side of the applications can be performed in the same fashion, with clear advantages in terms of perceived performance, efficiency and interactivity.

The client user interface of rich applications is programmed with extensive use of JavaScript and AJAX techniques.

Being JavaScript a single-threaded language, most of those programs are written in an event-driven style, in which programs register callback functions that are triggered on events such as timeouts. A single-threaded event loop dispatches the appropriate callback when an event occurs, and control returns back to the loop when the callback completes. To implement concurrency-intensive features, still keeping the interface responsive, programmers must chain callbacks together—typically, each callback ends by registering one or more additional callbacks, possibly with a short timeout.However, this style of event-driven programming is tedious, bug-prone, and harms reusability (Foster, 2008).

The limitations of the JavaScript programming model have been faced by introducing libraries that attempt at covering event-based low-level details behind the emulation of well-known concurrent abstractions.

Concurrent.Thread (Maki and Iwasaki, 2007) builds a thread abstraction on top of the event-driven JavaScript model, converting multi-threaded code into an efficient continuation-passing style. The WorkerPool API in Google Gears[1] simulates a collection of processes that do not share any execution state, thus can not directly access the browser DOM of Web pages. Albeit working in practice, neither approach is feasible to support a sound programming model for concurrency in Web 2.0 applications (Lee, 2006). Frameworks such as Rails,[2] with its Ruby-to-JavaScript compiler plug-in, and Google Web Toolkit (GWT)[3] approach the problem from a different angle, by adopting the programming model of the single language employed for implementing both the client- and server-side of the application. Even if the application development benefits from the use of object-oriented languages in terms of decomposition and en-

[1] http://gears.google.com
[2] http://www.rubyonrails.org/
[3] http://code.google.com/webtoolkit/

Piancastelli G., Ricci A. and Minotti M.
AN AGENT-BASED PROGRAMMING MODEL FOR DEVELOPING CLIENT-SIDE CONCURRENT WEB 2.0 APPLICATIONS.
DOI: 10.5220/0001822900130020
In *Proceedings of the Fifth International Conference on Web Information Systems and Technologies (WEBIST 2009)*, page
ISBN: 978-989-8111-81-4

capsulation, the limitations of the target execution environment make such solution less effective: for example, GWT does not allow the use of Java threads, and only offers a timer abstraction to schedule tasks (implemented as methods) within the single-threaded JavaScript interpreter.

We argue that, despite these several efforts, Web application client-side technologies do not offer suitable abstractions to manage the coordination and interaction problems of typical distributed applications in a simple yet complete fashion.

With the aim of supporting the development of Web applications as much similar as possible to distributed desktop applications, JavaScript and AJAX are not enough: a different programming model is needed, so as to provide mechanisms and abstractions really oriented to collaboration and cooperation among concurrent computational entities.

In this sense, even the object-oriented paradigm promoted by GWT shows its shortcomings.Indeed, mainstream object-oriented programming languages such as Java are currently undergoing a *concurrency revolution* (Sutter and Larus, 2005), where *(i)* support for multi-threading is extended by synchronisation mechanisms providing a *fine-grained* and efficient control on concurrent computations, and *(ii)* it is becoming more and more important to also provide more *coarse-grained* entities that help build concurrent programs from a set of higher-level abstractions.

We believe that, to effectively face challenges such as distributed computation, asynchronous communication, coordination and cooperation, *agents* represent a very promising abstraction, natively capturing and modeling concurrency of activities and their interaction; therefore, they can be considered a good candidate for defining a concurrent programming model beyond languages and technologies currently available within browsers on the client-side of Web applications.

The agent abstraction is meant to model and design the task-oriented parts of a system, encapsulating the logic and control of such activities. Not every entity in the system can (or should) be modeled in this way, though; we also introduce the companion *artifact* abstraction, as the basic building block to design the tools and resources used by agents during their activities.

Accordingly, in this paper we first describe the agent and artifact abstractions as defined by the A&A (Agents and Artifacts) programming model (Omicini et al., 2008), recently introduced in the field of agent-oriented programming and software engineering; then, we specialise the model in the context of client-side Web application development, by characterising common domain agents and artifacts that form a small extension of an existing programming framework based on A&A.

Further, we describe the design and implementation of a simple but significant case study to showcase the capabilities of the framework, and conclude with some final remarks.

2 THE AGENTS AND ARTIFACTS PROGRAMMING MODEL

In the A&A programming model, the term "agent" is used to represent an entity "who acts" towards an objective or task to do pro-actively, and whose computational behaviour accounts for performing *actions* in some kind of environment and getting information back in terms of *perceptions*. Differently from the typical software entity, agents have no interface: their interaction with the environment takes place solely in terms of actions and perceptions, which concerns in particular the use of artifacts. The notion of *activity* is employed to group related actions, as a way to structure the overall behaviour of an agent.

The A&A model is currently accompanied with the simpA framework[4] as a Java technology to prototype concurrent applications using the agent and artifact abstractions as basic building blocks (Ricci and Viroli, 2007). To define a new agent template in simpA, only one class must be defined, extending the `alice.simpa.Agent` base class provided by the framework API.For example, on the client-side of a Web e-mail application, an agent that fetches all the messages from an account, and periodically checks if new messages arrived, may be structured as follows:

```
public class MailAgent extends Agent {
  @ACTIVITY_WITH_AGENDA(
    todos={
      @TODO(activity="setup",
            persistent=false),
      @TODO(activity="fetch",
            persistent=false,
            pre=completed(setup)),
      @TODO(activity="check",
            persistent=true)
    }
  ) void main() {}
  @ACTIVITY void setup() { /* ... */ }
  @ACTIVITY void fetch() { /* ... */ }
  @ACTIVITY void check() { /* ... */ }}
```

Agent activities in simpA can be either atomic or structured, composed by some kinds of sub-activities. Atomic activities are implemented as methods with

[4] http://simpa.sourceforge.net

the `@ACTIVITY` annotation, with the body of the method defining the computational behaviour of the agent corresponding to the accomplishment of the activity. Structured activities introduce the notion of *agenda* to specify the hierarchical set of the potential sub-activities composing the activity, referenced as *todo* in the agenda. Each todo names the sub-activity to execute, and optionally a pre-condition. When a structured activity is executed, the todos in the agenda are executed as soon as their pre-conditions hold, but if no pre-condition is specified, the todo is immediately executed. Thus, multiple sub-activities can be executed concurrently in the context of the same (super) activity. A structured activity is implemented by methods with an `@ACTIVITY_WITH_AGENDA` annotation, containing todo descriptions as a list of `@TODO` annotations. A todo can be specified to be *persistent*—in this case, once it has been completely executed, it is reinserted in the agenda so as to be possibly executed again. This is useful to model cyclic behaviour of agents when executing some activities.

In the A&A programming model, the artifact abstraction is useful to design passive *resources* and *tools* that are *used* by agents as basic building blocks of the environment. The functionality of an artifact is structured in terms of *operations* whose execution can be triggered by agents through artifact *usage interface*.Similarly to the interface of objects or components, the usage interface of an artifact defines a set of controls that an agent can trigger so as to execute operations, each one identified by a label and a list of input parameters.Differently from the notion of object interfaces, in this *use* interaction there is no control coupling: when an agent triggers the execution of an operation, it retains its control (flow) and the operation execution on the artifact is carried on independently and asynchronously. The information flow from the artifact to agents is modeled in terms of *observable events* generated by artifacts and perceived by agents; therefore, artifact interface controls have no return values. An artifact can also have some *observable properties*, allowing to inspect the dynamic state of the artifact without necessarily executing operations on it. Artifact templates in simpA can be created by extending the base `alice.simpa.Artifact` class. For example, an artifact representing a Web page, storing its DOM model in an observable property, and exposing an operation to change an attribute of an element in the model, may be structured as follows:

```
public class Page extends Artifact {
  @OBSPROPERTY Document dom;
  @OPERATION void setAttribute(String id,
                               String attr,
                               String val) {
    Element e = dom.getElementById(id);
    e.setAttribute(attr, val);
    updateProperty("DOM", dom);
  }
}
```

Each operation listed in the artifact user interface is defined as a method with an `@OPERATION` annotation. Besides the usage interface, each artifact may define a *linking interface*, applying the `@LINK` annotation on operations that are meant to be invoked by other artifacts. Thus, it becomes possible to create complex artifacts as a composition of simpler ones, assembled dynamically by the linking mechanism. An artifact typically provides some level of observability, either by generating observable events during execution of an operation, or by defining observable properties using the `@OBSPROPERTY` annotation. An observable event can be perceived by the agent that has *used* the artifact by triggering the operation generating the event; changes to observable properties, triggered by the `updateProperty` primitive, can be sensed by any agent that has *focussed* on the artifact, without necessarily having acted on it. Besides agents and artifacts, the notion of *workspace* completes the basic set of abstractions defined in A&A: a workspace is a logic container of agents and artifacts, and it is the basic means to give an explicit (logical) topology to the working environment, so as to help scope the interaction inside it. We conclude this section by focussing on the main ingredients of the agent-artifact interaction model: a more comprehensive account and discussion of these and other features of agents, artifacts and workspaces – outside the scope of this paper – can be found in (Omicini et al., 2008; Ricci and Viroli, 2007).

2.1 Agent-Artifact Interaction: Use and Observation

The interaction between agents and artifacts strictly mimics the way in which people use their tools. For a simple but effective analogy, let us consider a coffee machine. The set of buttons of the coffee machine represents the usage interface, while the displays that are used to show the state of the machine represent artifact observable properties. The signals emitted by the coffee machine during its usage represent observable events generated by the artifact.

Interaction takes place by means of a `use` action (stage 1a in Figure 1, left), performed by an agent in order to trigger the execution of an operation over an artifact; such an action specifies the name and parameters of the interface control corresponding to the operation. The observable events, possibly generated by the artifact while executing an operation, are collected

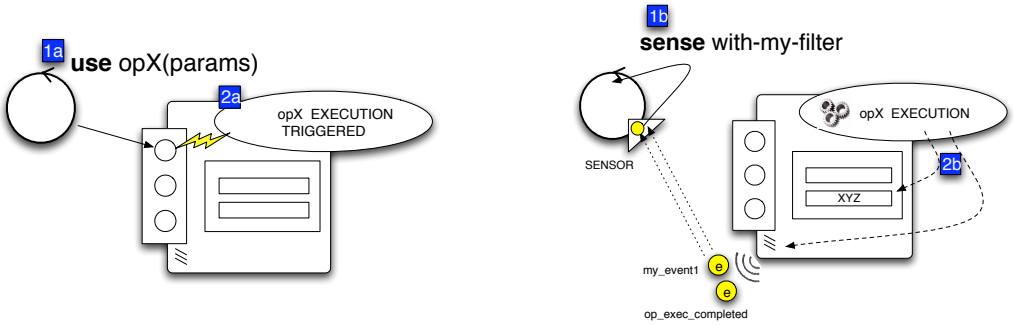

Figure 1: Abstract representation of an agent using an artifact, by triggering the execution of an operation (left, step 1a) and observing the related events generated by the operation execution (right, step 1b).

by agent *sensors*, which are the parts of the agent conceptually connected to the environment where the agent itself is situated.Besides the generation of observable events, the execution of an operation results in updating the artifact inner state and possibly artifact observable properties. Finally, a `sense` action is executed by an agent to explicitly retrieve the observable events collected by its sensors. It must be noted that, dually to indirect interaction through artifacts, agents can also directly communicate with each other, through a message passing mechanism provided by the framework.

3 AN AGENT-ORIENTED MODEL FOR CLIENT-SIDE WEB PROGRAMMING

The main objective of our programming model based on agent and artifact abstractions is to simplify development of those parts on the client-side of Web applications that involve elements of concurrency, by reducing the gap between design and implementation. At the design level, it is first necessary to identify the task-oriented and function-oriented parts of the client-side system; then, such organisation drives to the definition of agents and artifacts as depicted by the A&A model. At the implementation level, the design elements can be directly realised using the simpA framework. The notion of activity and the hierarchical activity model adopted in the agent abstraction allow a quite synthetic and readable description of possibly articulated behaviours, dealing with coordination management on a higher level than threads, timeouts, and callbacks. The adopted model of artifacts permits to specify possibly complex functionalities to be shared and concurrently exploited by multiple agents, abstracting away from low-level synchronisation mechanisms and primitives. In the domain of client-side Web development, all computation takes place in the browser environment, that can be straightforwardly mapped onto the *workspace* abstraction; agents and artifacts downloaded as the client-side part of Web applications will join this workspace, interacting and cooperating in a concurrent fashion. Among those agents and artifacts, there are a number of recurring elements in the majority of applications; here follows a description of the main elements that we identified, along with the role they will play in our programming model.

Page. The page is typically represented by an accessible, tree-like, standardised DOM object, allowing to dynamically update its content, structure, and visualisation style; also, the page generates internal events in response to user's actions. The direct mapping of the DOM API onto operations, and of the response to user's actions onto observable events, suggests the *artifact* as the most natural abstraction to model pages.

HTTP Channel. An entity is needed to perform transactions in the HTTP protocol, allowing to specify the operation to execute (e.g. GET, POST, PUT, DELETE), set the header values and possibly a payload in the request; such an entity also has to receive responses and make them available to the other entities in the system. The channel does not account for autonomous actions, but it is supposed to be used by a different, pro-active entity; it is therefore modeled as an *artifact*, so that asynchronous communication towards the server can be ensured by the agent-artifact interaction model.

Common Domain Elements. Common components that are frequently and repeatedly found in specific application domains can also be modeled in terms of agents and artifacts. As an example, consider the shopping cart of any e-commerce Web site: since its control logic and functionalities are almost always the same, it could be implemented

as a client-side component, in order to be used by a multiplicity of e-commerce transactions towards different vendors, and allowing comparisons and other interesting actions on items stored for possibly future purchase. Given the possibly complex nature of the computations involved, such common domain elements would probably need to be modeled as a mix of both agents and artifacts.

There also are some important issues in client-side Web application development that, due to the lack of space, we only intend to acknowledge without providing a complete description of their possible modeling and programming solution. First, security needs to be addressed, in order to manage read/write permissions on the file system, mobile code authentication, origin control, and access authorisation; to this purpose, the RBAC model (Sandhu et al., 1996) provided by simpA can be exploited. Then, albeit architecturally deprecated, also cookies need to be taken into account as a popular mechanism to allow data persistence between working sessions; they can be devised as another particular kind of artifact template.

3.1 From simpA to simpA-Web

simpA-Web is a thin layer that exploits classes and mechanisms of the simpA framework to define agent and artifact templates oriented to client-side Web development. Whereas simpA supports A&A abstractions, simpA-Web offers specific agents and artifacts representing the common elements comprised by the programming model explained above.

For example, simpA-Web provides implementations for the HTTP Channel and the Page artifacts. The HTTP Channel artifact represents a HTTP working session used by a part of the client-side Web application. The artifact allows communication through the HTTP protocol, and is able to store and manage both local and remote variables. The user interface of HTTP Channel exposes three operations:

- `setDestination`. stores the URI of the server to which the session represented by this artifact will communicate. This operation takes the destination URI as a parameter, and generates a DestinationChanged observable event.
- `setHeader`. adjusts a HTTP header for subsequent requests. It takes the header name and value as parameters, and generates a HeaderChanged observable event.
- `send` transmits a HTTP request to the server identified by the stored URI. It takes the HTTP payload as a parameter, and generates three events: RequestSent, as soon as the HTTP request has been committed; Response, containing headers and body, when a HTTP response is received; Failure, if no response has been sent back.

The Page artifact represents the interface to access the page visualised by the browser, encapsulating its events and its main characteristics. The Page artifact features an observable property called DOM, representing the Document Object Model of the whole page. The artifact interface exposes six operations, each generating a corresponding observable event at the end of a successful execution:

- `changeDOM`. substitutes the old DOM with a new object taken as a parameter, thus changing the representation of the whole page.
- `setElement`. changes the content of a DOM element identified by the first `id` parameter to the `string` passed as second parameter.
- `setAttribute`. does the same as the previous operation, but on a property of a DOM element.
- `addChild`. appends a DOM element as a child to an element with a given `id`.

The presented entities in the simpA-Web layer represent a common agent-oriented infrastructure deployed as an additional library alongside simpA, so as to free client-side Web application developers from dealing with such a support level and let them focus on application domain issues.

4 A SAMPLE AGENT-ORIENTED WEB 2.0 APPLICATION

To verify the feasibility of our A&A-based programming model and test-drive the capabilities of the simpA-Web framework, we designed a sample Web application to search products and compare prices from multiple services. The characteristics of concurrency and periodicity of the activities that the client-side needs to perform make this case study a significant prototype of the typical Web 2.0 application.

We imagine the existence of N services (of type A) that offer product lists with features and prices, codified in some standard machine-readable format. Each type A service lets users download an agent (typically programmed by the service supplier) that allow product searching on that service; each agent communicates with the corresponding service using its own protocol, possibly different from the protocols of the other agents in the system.We further imagine the existence of an additional service (of type B) offering a

static list of type A services or allowing to dynamically search the Web for such services.

The client-side in this sample Web application needs to search all type A services for a product that satisfies a set of user-defined parameters and has a price inferior to a certain user-defined threshold. The client also needs to periodically monitor services so as to search for new offerings of the same product.

A new offering satisfying the constraints should be visualised only when its price is more convenient than the currently best price. The client may finish its search and monitoring activities when some user-defined conditions are met—a certain amount of time is elapsed, or the user interrupts the search with a click on a proper button in the page displayed by the browser. Finally, if such an interruption took place, by pressing another button it must be possible to let the search continue from the point where it was blocked.

It's worth remarking that the sample application should be considered meaningful *not* for evaluating the performance or efficiency of the framework – which is part of future work – but for stressing the benefits of the agent-oriented programming model in facing the design of articulated and challenging application scenarios. The main value in this case is having a small set of high-level abstractions that make it possible to keep a certain degree of modularisation, separation of concerns, and encapsulation in designing and programming applications, despite of the complexities given by aspects such as concurrency, asynchronous interactions, openness/dynamism, distribution.

4.1 Design

From the above description, it is fairly easy to use high-level abstractions such as agents and artifacts in order to represent the different computational elements in the client-side of the application.

We define the **Application Main (AM) Agent** as the first agent to execute on the client, with the responsibility to setup the application once downloaded. Immediately after activation, the agent populates the workspace with artifacts and other agents needed to perform the product search as defined by the requirements. After validating user-defined data for product querying, the agent spawns another agent to interface with the B service so as to get a list of A services and commence the product search. The AM Agent also needs to control results and terminate the research when a suitable condition is verified; moreover, it has to manage search interruption and restarting as a consequence of user's explicit commands.

The task of the **Service Finder (SF) Agent** is to use the type B service so as to find a list of type A services, and to concurrently download search agents from their sites, asking them to start their own activity. The SF Agent also has to periodically monitor the type B service for new type A services to exploit.

The **Product Finder (PF) Agent** instances interact with their own type A service by a possibly complex communication protocol, so as to get a list of products satisfying the user-defined parameters. After, each agent passes the resulting list to another element in the workspace dealing with product data storage. Each PF Agent also periodically checks its service in order to possibly find new products or removing from the list products that are no more available.

The **Product Directory (PD) Artifact** stores product data found by the PF agents, and applies filtering and ordering criteria to build a list of available products, possibly containing only one element.The artifact also shows a representation for its list on the page in the browser, updating results as they arrive.

Finally, an additional **Pausing Artifact** is introduced to coordinate the agents in the workspace on the interruption or restarting of the research activities in response to a user's command.While the control logic of search interruption and restarting is encapsulated in the AM agent, the agent does not directly know all the PF agents; in such a case when direct communication between agents is unfeasible, an intermediate coordination element is needed to ask agents to perform the actions requested by the user.

The complete architecture of the client-side application is depicted in Figure 2, where interactions with pre-defined artifacts and agents in the browser are shown. In particular, it must be noted that the Service Finder and Product Finder agents use their own instances of the HTTP Channel artifact to communicate through the Web with the correspondent sites; also, the Product Directory artifact links the Page artifact in order to visualise the product list by directly use the DOM API operations that are made available through the Page artifact linking interface.

It's worth noting that we chose to place all the agents and artifacts described above on the client side to maximise the distribution of the workload (with respect to the server-side): however, it is straightforward to devise different design solutions where some of the agents (and artifacts) run on the server side and interact with agents on the client side by using proper artifacts like the HTTP Channels.

4.2 Implementation

Since we wanted our application case study to exploit existing Web clients, so as to verify the feasibility of

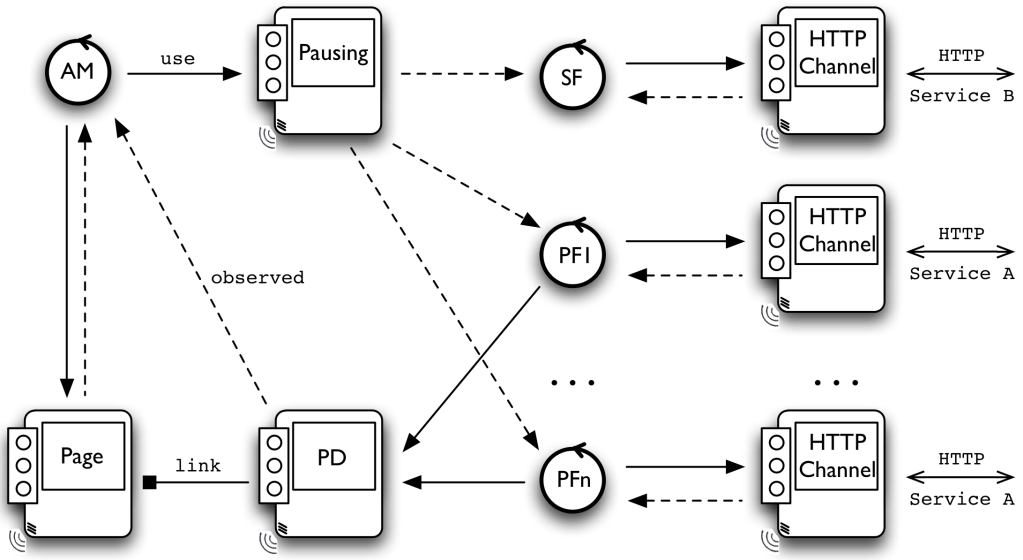

Figure 2: The architecture of the sample client-side Web application in terms of agent, artifacts, and their interactions.

current technologies w.r.t. our new model, resorting to design or implementation compromise when needed, we had to face some discrepancies that emerged during the implementation phase.

As we could not use JavaScript, due to its single-threaded nature, as a language for the client-side of our sample Web application, we adopted the other mechanism for code mobility that browsers make available, that is Java applets. Applets allow to transfer code from a server to a browser and have it executed within a controlled secure environment known as sandbox; in particular, *signed applets* drop much of the security constraints of the sandbox, for instance allowing Java classes to open their own connections towards multiple servers. Furthermore, the Java Virtual Machine invoked by the browser does not force any restriction on the number of threads that a program may spawn, thus providing a truly concurrent environment where to execute our application. Access to the page DOM representation in the browser is granted by a suitable JavaScript library called LiveConnect,[5] that can be conveniently invoked from the JVM multi-threaded execution context. Another issue is that browsers start without any notion of agents or artifacts: for this reason, the simpA-Web framework has to be entirely downloaded at the same time as the agents and artifacts belonging to the real client-side application. In particular, we exploit the applet mechanism allowing a specific class to start as soon as the JAR files have been completely fetched from the Web site in order to activate the Application Main agent. The Page artifact is programmed to access the DOM API by using the Rhinohide[6] library, a convenient wrapper around LiveConnect that offers a simpler and more complete support for page event management. The HTTP Channel artifact does not exploit the HTTP protocol support in the browser, instead relying on functionalities offered by the Java standard library. As a relevant code snippet, we show the Product Finder Agent interacting with the HTTP Channel artifact to get the product list from its type A service.

```
public class ProductFinder extends Agent {
  @ACTIVITY void find() {
    ArtifactId http =
          lookupArtifact("HTTP_" + getId());
    String url = getServiceURL() + getQuery();
    // Make the Request
    use(http, new Op("setDestination", url));
    SensorId sid = getSensor("s0");
    use(http, new Op("send", "GET", ""), sid);
    // Sense the Response
    Perception p = sense(sid, "Response");
    String body = p.stringContent(0);
    // ...
  }
}
```

The source code of the application – including a PHP back-end simulating type A and B services, used to test the system – can be downloaded from simpA web site.

[5] http://developer.mozilla.org/en/LiveConnect

[6] http://zelea.com/project/textbender/o/rhinohide

5 CONCLUDING REMARKS

Current languages and techniques used to develop the client-side of Web 2.0 applications have clearly shown their shortcomings in terms of programming model and code organisation. Even for a mildly complex application such as the presented case study, a possible JavaScript-based approach is not feasible to effectively manage concurrent and periodical activities so as to promote engineering properties such as maintainability, extendibility, and reusability. In particular, the use of callbacks and timeouts to fit the single-threaded event-based programming style of JavaScript constrains developers to work at such a low abstraction level that a high degree of flexibility and a solid design are hard to achieve within reasonable amounts of time and effort. In this scenario, toolkits such as GWT appear only to partially help: while they may reduce the time-to-market and improve the overall structure and organisation of applications, their programming abstractions still lack the expressivity to represent the coordinating and cooperating entities of typical concurrent systems.

Agents have been shown to natively capture concurrency of activities and their interaction, both at the design and the implementation level. In the pre-AJAX era, agents had already been used as the basic building block for client-side concurrent Web programming (Hong and Clark, 2000). This suggests that agent-oriented programming models such as A&A may effectively help construct the kind of highly interactive applications that populate the Web 2.0 and exploit concurrency as their primary computational paradigm. As a significant prototype of this kind of applications, the presented case study has been designed and implemented by employing a set of engineering best practices and principles (including modularisation, separation of concerns, encapsulation) that are directly supported by the A&A model, but that are hardly represented in the current technologies for client-side Web development. The integration difficulties that agent-oriented technologies such as simpA have to overcome in order to seamlessly work on modern Web clients should not be considered as diminishing the value of the programming model, but as a temporary accident due to the lack of proper support for the concurrency paradigm in the mainstream.

As a final note, the use of agents to represent concurrent and interoperable computational entities already sets the stage for a possible evolution of Web 2.0 applications into *Semantic Web* applications. From the very beginning (Berners-Lee et al., 2001; Hendler, 2001), research activity on the Semantic Web has always dealt with *intelligent agents* capable of reasoning on machine-readable descriptions of Web resources, adapting their plans to the open Internet environment in order to reach a user-defined goal, and negotiating, collaborating, and interacting with each other during their activities. Concurrency and intelligence being two orthogonal programming concepts, agent-oriented models that combine both aspects are likely to be adopted for future mainstream technologies.

REFERENCES

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*.

Foster, J. S. (2008). Directing JavaScript with Arrows. In *ICFP 2008: The 13th ACM SIGPLAN International Conference on Functional Programming*, Victoria, British Columbia, Canada. Poster paper.

Hendler, J. (2001). Agents and the Semantic Web. *IEEE Intelligent Systems*, 16(2):30–37.

Hong, T. W. and Clark, K. L. (2000). Concurrent programming on the Web with Webstream. Technical report, Department of Computing, Imperial College, London.

Lee, E. A. (2006). The problem with threads. *IEEE Computer*, 39(5):33–42.

Maki, D. and Iwasaki, H. (2007). A portable JavaScript thread library for Ajax applications. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 817–818, Montreal, Quebec, Canada.

Omicini, A., Ricci, A., and Viroli, M. (2008). Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456.

Ricci, A. and Viroli, M. (2007). simpA: an agent-oriented approach for prototyping concurrent applications on top of Java. In *PPPJ '07: The 5th international symposium on Principles and practice of programming in Java*, pages 185–194, Lisboa, Portugal.

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. (1996). Role-based access control models. *IEEE Computer*, 29(2):38–47.

Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *ACM Queue: Tomorrow's Computing Today*, 3(7):54–62.