Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Modular and flexible causality control on the Web

Paul Leger[b,*], Éric Tanter[a,1], Rémi Douence[c]

[a] PLEIAD Lab, Computer Science Department, University of Chile, Chile
[b] Universidad Católica del Norte, Chile
[c] ASCOLA Project, INRIA, France

ARTICLE INFO

Article history:
Received 19 July 2011
Received in revised form 15 November 2012
Accepted 17 November 2012
Available online 21 December 2012

Keywords:
WeCa
AspectScript
OTM
JavaScript
Web 2.0 Applications
Distributed computing
Aspect-oriented programming

ABSTRACT

Ajax allows JavaScript programmers to create interactive, collaborative, and user-centered Web applications, known as Web 2.0 Applications. These Web applications behave as distributed systems because processors are user machines that are used to send and receive messages between one another. Unsurprisingly, these applications have to address the same causality issues present in distributed systems like the need (a) to control the causality between messages sent and responses received and (b) to react to distributed causal relations. JavaScript programmers overcome these issues using rudimentary and alternative techniques that largely ignore the distributed computing theory. In addition, these techniques are not very flexible and need to intrusively modify these Web applications. In this paper, we study how causality issues affect these applications and present WeCa, a practical library that allows for modular and flexible control over these causality issues in Web applications. In contrast to current proposals, WeCa is based on (stateful) aspects, message ordering strategies, and vector clocks. We illustrate WeCa in action with several practical examples from the realm of Web applications. In addition, we evaluate our proposal with a third-party application and its performance.

1. Introduction

There is a strong trend towards the use of Web 2.0 Applications (WebApps for now) [1], interactive, collaborative, and user-centered Web applications, like Twitter and Facebook. For the development of these applications, the JavaScript language and Ajax technologies [2] are widely used. This is so because JavaScript, a dynamic prototype-based language with higher-order functions, is supported by most modern browsers and Ajax technologies allow WebApps to send and receive messages from a server or other applications[2] asynchronously. The latter feature converts WebApps into distributed systems because processors are user machines that are used to send and receive messages between one another. As a consequence, these applications have to address the causality issues of distributed systems like the need (a) to control the causality between messages sent and responses received [3] and (b) to react to distributed causal relations [4].

The need to control the causality between messages sent and responses received arises when a WebApp retrieves various server responses in an arbitrary order. Instead, the need to react to distributed causal relations arises, for example, when it is necessary to analyze the flow of information at runtime that occurs among WebApps [5]. Surprisingly, there is not much technical support that addresses the previous issues and even less modularly and flexibly. The current proposals

* Corresponding author. Tel.: +56 051209839.
E-mail addresses: pleger@dcc.uchile.cl, pleger@ucn.cl (P. Leger).
1 É. Tanter is partially funded by FONDECYT project 1110051.
2 Using a server as intermediary.

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.11.005

[6–8] allow JavaScript programmers to overcome these issues using rudimentary techniques like explicit function wrappers and post-mortem techniques like dynamic graphs. In addition, these techniques largely ignore the distributed computing theory and/or their inflexible uses end up scattered throughout many places in the code of a WebApp entangled with other concerns. In this paper, we present WeCa, a practical library that allows for modular and flexible control over causality on the Web.

In contrast to current proposals, WeCa is based on Aspect-Oriented Programming (AOP) [9] and distributed computing concepts [10]. In particular, WeCa uses stateful aspects [11], vector clocks [4,12] and message ordering strategies [3]. Our proposal allows for modular and flexible definition of (a) message ordering strategies that control the causality between Ajax requests and server responses and (b) monitors that react to distributed causal relations. Currently, the WeCa implementation uses AspectScript [13], an aspect-oriented extension of JavaScript, to define aspects that enforce message ordering strategies in WebApps. In addition, the WeCa implementation uses OTM [14,15], an Open Trace-based Mechanism like tracematches [16] for JavaScript, to define stateful aspects that react to distributed causal relations in WebApps.

The rest of this paper is organized as follows. Section 2 presents and illustrates some causality issues on the Web through different WebApps. Section 3 introduces key concepts on which WeCa is based: (stateful) aspects, message ordering strategies, and vector clocks. Section 4 introduces WeCa, which combines the aforementioned concepts to address modularly and flexibly causality issues. Section 5 presents how our proposal addresses the causality issues described in Section 2. Section 6 presents different evaluations of WeCa. Section 7 discusses related work and Section 8 concludes.

2. Ajax & Web 2.0 Applications

Ajax [2], a shorthand for *Asynchronous JavaScript and XML*, is a group of interrelated Web technologies used on the client-side to create interactive Web applications. Using Ajax, Web applications can send and retrieve data from a server or other applications asynchronously without reloading the current Web page. The following piece of code written in JavaScript shows a simple request to a server using Ajax:

```
var request = new XMLHttpRequest();
request.open("GET","server.com");
request.onreadystatechange = function() {
  if (this.readyState == 4)
    updateWebPage(this.responseText);
}
request.send(parameters);
```

The request object represents the *Ajax request* to the server. The open method configures the communication with the server, and the send method sets the parameters of the Ajax request and sends it. The server responds with an arbitrary delay. When the server responds, the onreadystatechange method is executed.[3] The responseText property of request contains the *server response*, which can be used, for example, to asynchronously update the current Web page.

Using Ajax technologies, JavaScript programmers create interactive, collaborative, and user-centered Web Applications, known as Web 2.0 Applications [1]. For example, Housing Maps, an application for finding a house for sale, is created from server responses that come from different sources. Another example is Twitter, an application for social network and microblogging, which allows users to send and receive messages. When a WebApp uses Ajax technologies, some needs arise: (a) controlling the causality between messages and responses received [3] and (b) reacting to distributed causal relations [4].

2.1. Controlling message causality

A WebApp can send several Ajax requests to different servers. However, the application can retrieve and process the server responses in an arbitrary order; so this application may behave nondeterministically due to the lack of control of the causality between Ajax requests and server responses. For example, Fig. 1 shows that if a WebApp sends two Ajax requests to a server, two scenarios are possible: the server returns server response 1 followed by server response 2 or vice versa. Depending on the expected behavior of the WebApp, different strategies to control the message causality can be used. We now present WebApps that require three different strategies:

A FIFO strategy to create a Web application from different servers. A mashup application is created from the combination of information retrieved from different servers. Programmers have to overcome the issue of creating an incorrect Web page due to an arbitrary (and unexpected) order of server responses. For instance, consider a Web page that is created with two Ajax requests. If the second server response is processed before the first server response, the Web page is created incorrectly. A *FIFO* strategy, which processes the server responses in the same order as the Ajax requests are sent, ensures that the Web page is always created correctly.

A Discard Late strategy for the visualization of streaming of videos. Programmers have to overcome the issue of using obsolete data due to a late server response. For example, consider a Web application, like Ajax Video Player [17], that shows

[3] This method is executed every time that readyState changes. We only want to take an action when the server response is available, *i.e.* when readystate takes the value 4.

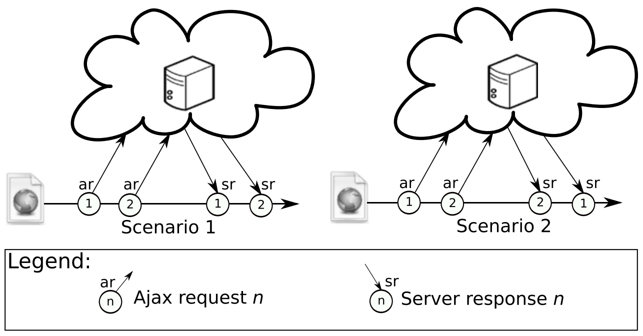


Fig. 1. Two possible scenarios when a Web application sends two Ajax requests.

streaming of videos using frame buffers retrieved from server responses. If some server responses are retrieved in a wrong order, the video cannot be observed correctly. Two unsatisfactory solutions are possible. The first and current solution of Ajax Video Player is to perform only one Ajax request at a time; this however introduces significant interruptions during the visualization of the video. The second solution is to wait for late buffers, with the risk of delaying too much the visualization of the video (*e.g.* in the extreme case, the first buffer could be received last). A better solution is to use simultaneous Ajax requests to retrieve a set of contiguous frame buffers and discard the late buffers. A *Discard Late* strategy that discards the late (and obsolete) server responses can always show a streaming video in a correct flow.

A Discard Early strategy for the visualization of discussions of posts. Another issue in the visualization arises due to the processing of early server responses. For example, consider a forum Web page that is frequently updated with posts from a server using Ajax (*e.g.* Slashdot) increases with more threads that are located on the top. Every post is shown with its title and its first comments, and the reader can click on a button "read more" to see the whole discussion of a given post. Every comment that is below the post title is loaded dynamically using an independent Ajax request. If server responses, which contain the comments, are retrieved and displayed in an incorrect order, the beginning of the discussion of the post could be misunderstood. A solution could be to wait for all late comments of every post. However, this solution could seriously delay the updating of the forum Web page. A better solution is to discard the early comments in order to correctly show the first comments of the post. The discarded comments can be fetched later when the reader clicks the given "read more" button. A *Discard Early* strategy, which discards early server responses, can ensure the beginning of the discussion of every post is read in a correct order.

2.2. Reacting to distributed causal relations

Social network applications like Twitter and Facebook are another kind of WebApp. Nowadays, these applications are widely used, making the analysis of their flow of information an active research topic [5,18–20]. This flow of information is analyzed through the messages sent and received between users of these applications. Such an analysis is complex due to the need to observe and react to distributed causal relations that occur among user interactions. As an example of the analysis of the flow of information in WebApps, consider the calculation of the popularity of user *tweets*[4] in Twitter:
Tweet popularity. This feature in Twitter [21] allows a user to know the popularity of every tweet published by him or her, which is measured by the number of *retweets*[5] of direct and indirect followers. For example, Fig. 2 shows four Tweeter users: Toti, Dacha, Kuky, and Paul. Toti follows Dacha and Dacha follows Paul; Kuky follows nobody and nobody follows Kuky. The figure shows that Paul publishes a tweet and Dacha receives this tweet and retweets it. The figure also shows that Kuky publishes a tweet and nobody receives it. Based on the popularity measurement, the popularity of Paul's tweet is 1 and that of Kuky is 0. Although Kuky and Paul would have published the same tweet,[6] the popularity of Kuky's tweet is 0 because his tweet did not cause any retweet. An analysis based on the distributed causal relations observed between tweets and retweets can determine how many users retweet a given tweet. For example, Paul's tweet caused Dacha's retweet.[7]

2.3. State of the practice

State of the practice provides JavaScript practitioners have at their disposal a number of JavaScript libraries and post-mortem tools to solve the kind of issues we described previously.

4 Messages posted via Twitter containing 140 characters or fewer. This word is indistinctly used as a noun and a verb.
5 Tweets reposted by another user.
6 Two tweets are equal if both contain the same string or have the same url associated.
7 We illustrate the causality concept in distributed Web applications with this example, acknowledging that in real life, this causality link is so important that it is already embedded in (services of) Twitter.

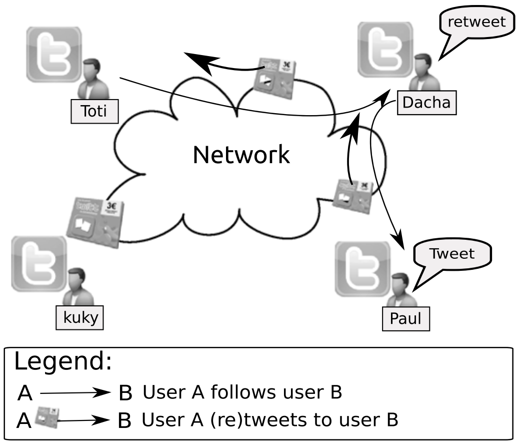
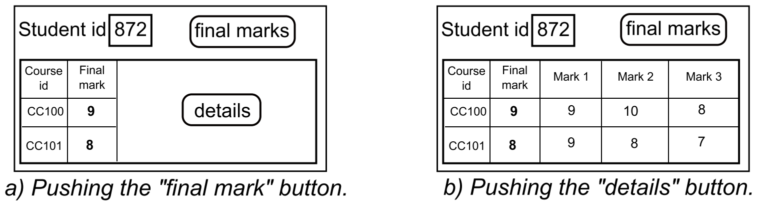

retweet
Toti
Dacha
Network
Tweet
kuky
Paul
Legend:
A → B User A follows user B
A → B User A (re)tweets to user B

Fig. 2. Retweeting a tweet.


a) Pushing the "final mark" button.
b) Pushing the "details" button.
final marks
details

Student id 872

Course id	Final mark
CC100	**9**
CC101	**8**

Student id 872

Course id	Final mark	Mark 1	Mark 2	Mark 3
CC100	**9**	9	10	8
CC101	**8**	9	8	7

Fig. 3. A Web application that shows the final marks to a student, and also allows the student to see each mark of his courses.

Libraries for controlling message causality. A common need for JavaScript developers is to control server responses [22]. We illustrate this need with the Web application shown in Fig. 3. Fig. 3(a) shows that a student can use this application to see the final marks of his courses. When the student pushes the button "details", the Web application shows the marks used to calculate his final marks (Fig. 3(b)). Final marks and all remaining marks are accessed by two different Ajax requests. Notice that using only one Ajax request is slower if the student only wants to know his final marks. Although both requests can be simultaneously sent, the response of the request for all marks cannot be processed before the request for final marks. This is so because to process the request of all marks, it is necessary to know where to insert each mark in the Web page (*e.g.* marks for the course "CC100" must be on the top of the table).

```
finalMarksButton.onclick = function() {
  //Creation and configuration of an \ajax request object
  request.onreadystatechange = function() {
    if (this.readyState == 4) {
      showFinalMarks(getFinalMarks(this.responseText));
      detailsButton.enabled = true; //the details button is enabled
    }
  }
  request.send("finalMarks"); };


detailsButton.onclick = function() {
  //Creation and configuration of an \ajax request object
  request.onreadystatechange = function() {
    if (this.readyState == 4)
      showMarks(getMarks(this.responseText));
  }
  request.send("allMarks");
};
```

The piece of code above represents the typical solution for developers. When the first request (*i.e.* final marks) is processed, the second one can be sent. This solution is slow because both requests cannot be simultaneously sent. In addition, the line that enables the "details" button mixes two concerns: the display of final marks and the display of all marks. Some lightweight JavaScript libraries [6,7] can be used. For example, the following piece of code uses the AjaxManager library [6] to enforce the processing of the first server response before the second server response.

```
//creating a manager that enforces the FIFO strategy
var am = manageAjax.create({
  queue: true, //this property means that FIFO is enabled
  //other properties...
});

finalMarksButton.onclick = function() {
  am.add({
      success: function(serverResponse) {
        showFinalMarks(getFinalMarks(serverResponse));
    },
    url://url
  });
};

detailsButton.onclick = function() {
  am.add({
      success: function(serverResponse) {
        showMarks(getMarks(serverResponse));
    },
    url://url
  });
};
```

With AjaxManager, both requests can be simultaneously sent. However, AjaxManager has three drawbacks. First, an AjaxManager object must be created and available in the scope of all Ajax requests that need to follow a strategy; this may imply passing it around as parameter, or having it in the surrounding scope of requests, as in the above example. Second, these requests must be wrapped to follow the strategy. Third, the AjaxManager library only provides a limited set of strategies, which are also not customizable (*e.g.* the Discard Early strategy is not supported). In general, libraries similar to AjaxManager share the same drawbacks.

In contrast, with WeCa the developer only needs to insert the following line before sending the two Ajax requests:

```
WeCa.applyStrategy(WeCa.FIFO) //FIFO is a JavaScript function (Section 4.1)
```

Post-mortem tools for reacting to distributed causal relations. Postmortem tools based on dynamic graphs [8] are used to observe distributed causal relations, which allow users to improve the understanding the people behavior in a social network application. However, these tools are post-mortem, therefore, they cannot be used to react to distributed causal relations at runtime. Nowadays, the runtime analysis is necessary, for example, to quickly mitigate the effect of the failure of a new product of a company. In particular, people, like social community managers,[8] are constantly analyzing the behavior of social network users in order to take fast and informed decisions of a new product or service [23].

2.4. WeCa overview

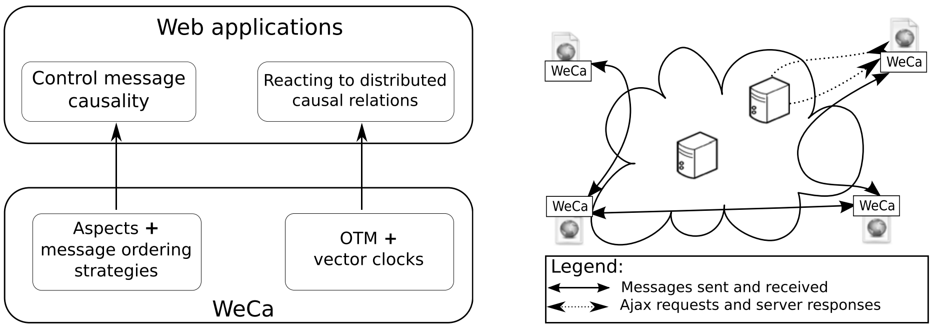
WeCa is a practical library that allows for modular and flexible control over causality as required in the examples presented previously (see Fig. 4(a)). By *modular* we mean that the control over causality is addressed in a separate module at the code level, and by *flexible* we mean that the definition of the control over causality is customizable using the power of the base language. The runtime of WeCa observes every Ajax request with its server response to enforce a certain strategy to deal with server responses (*e.g.* Discard Early). In addition, this runtime observes every message sent and received between these applications to detect distributed causal relations at runtime in interactive communication of WebApps (*e.g.* the popularity of a tweet).

Fig. 4(b) shows how WeCa works in a nutshell. Every WebApp has a WeCa runtime instance. Each instance can use an *aspect* [9] to match every server response, and using *message ordering strategies* [3], the aspect enforces that (some) server responses follow a certain message ordering strategy. In addition, each instance can use *stateful aspects* [11] to match and react to distributed computations. These stateful aspects utilize *vector clocks* [4,12] to match and react to distributed computations that satisfy certain causal relations.

3. Aspect-oriented programming & distributed computing

WeCa combines concepts from AOP and distributed computing [10]. In this section, we summarize these necessary concepts in order to clarify our proposal.

[8] http://www.toprankblog.com/2011/01/community-manager-do/.


Web applications
Control message causality
Reacting to distributed causal relations
Aspects + message ordering strategies
OTM + vector clocks
WeCa
WeCa
Legend:
Messages sent and received
Ajax requests and server responses

(a) The solution proposed by WeCa. (b) WeCa overview.

Fig. 4. WeCa.

3.1. *Aspect-oriented programming*

Aspect-oriented programming makes it possible to modularize *crosscutting concerns*, like the ability to react to distributed causal relations. Specifically, in the pointcut-advice model for aspect-oriented programming [24,25], crosscutting behavior is defined by means of *pointcuts* and *advice*. Execution points at which advice may be executed are called *(dynamic) join points*. A pointcut matches a set of join points, and a piece of advice is the action to be taken *before*, *around*, or *after* the matched join point. Around advice can invoke the computation of the join point matched, known as the *proceed* invocation. An *aspect* is a module that encompasses pointcuts and advice.

3.1.1. *AspectScript in a nutshell*

AspectScript [13] is an aspect-oriented extension of JavaScript, which follows the pointcut-advice model. We introduce AspectScript using an example: consider a Web page that is frequently updated through Ajax and the following aspect updates the Web page only when it retrieves a new server response from a server:

```
var aspSelectiveUpdate = {
  pointcut: function(jp,env) {
    return jp.isExec() && jp.target instanceof XMLHttpRequest &&
           jp.fun == jp.target.onreadystatechange?
         env.bind("serverResponse",jp.target.responseText): false;
  },
  advice: function(jp,env) {
    if (this.lastServerResponse != env.serverResponse) {
      jp.proceed();
      this.lastServerResponse = env.serverResponse;
    }
   },
   kind: AROUND,
   lastServerResponse: null //initial value
};

AspectScript.deploy(aspSelectiveUpdate); //deployment
```

In AspectScript, an aspect is a plain JavaScript object that defines at least three properties (pointcut, advice, and kind). The pointcut and advice properties are plain JavaScript functions parametrized by the jp and env objects: jp represents the current join point and env is an environment that is used to pass information from the pointcut to the corresponding advice. As a consequence, a join point is a first-class value, where context exposure and original computation (*i.e.* proceed) are properties of the join point. The pointcut function matches the jp join point; the advice function takes the action before, around, or after (according to the kind property) of the join point matched. In our example, the aspSelectiveUpdate aspect defines a pointcut that returns an environment that contains the server response when this pointcut matches the execution of the onreadystatechange method; the around advice of the aspect only executes the proceed method if the server response differs from the last one.

Pointcut Model. Unlike many aspect-oriented extensions like AspectJ [26], where a pointcut is defined by a pattern expressed in a domain-specific language, AspectScript uses the base language, in particular higher-order functions, to express a pointcut. Following standard practice [24], a pointcut can return (a) an environment if it matches the current join point or (b) false if it does not.[9]

[9] A peculiarity of AspectScript is the environment passed as parameter to AspectScript pointcuts. This environment is used to pass context information between pointcuts.

3.1.2. OTM in a nutshell

The pointcut of an aspect refers to the current join point. Therefore, a pointcut cannot refer to a trace of join points. For example, consider the implementation of an aspect that prevents a malicious application from inserting (random) credentials several times until logging successfully[10]:

```
var lg = function(jp,env) {
  return jp.isExec() && jp.fun == login? env:false;
};

var aspLogin = {
  pointcut: lg,
  advice: function(jp,env) {
    if (++this.counter > 3)
      throw "Only three attempts are permitted";
  },
  kind: BEFORE,
  counter: 0
}

AspectScript.deploy(aspLogin); //deployment
```

To match the execution trace of three logins, the aspLogin aspect maintains a counter to keep track of the number of login attempts. Every time the login function is called, we increase the counter by one; and if this counter reaches four, the exception is triggered. Note how the state of the matching process (*i.e.* the counter) is explicit in this aspect. In this example, the explicit matching process leads to only minor complications, however, the burden of such state maintenance is often much greater. For example, the solution of a counter is not sufficient if we need to match a sequence of different pointcuts.

Trace-based mechanisms like tracematches [16] support the definition of *stateful aspects* [11] that match and react to an execution trace of an application. OTM [14,15], a seamless AspectScript extension, is an Open Trace-based Mechanism for JavaScript. For example, consider the previous example using OTM:

```
var sAspLogin = {
  sequence: seqn(lg,4),
  advice: function(jp,env) {
     throw "Only three attempts are permitted";
  },
  kind: BEFORE
};
OTM.deploy(sAspLogin); //deployment
```

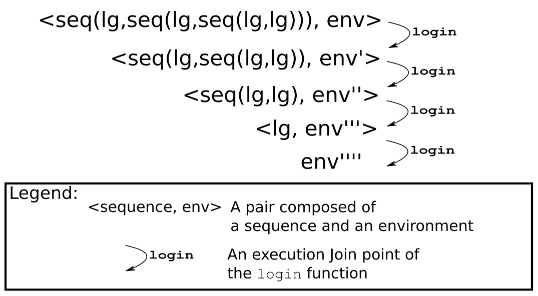
Similar to AspectScript, a stateful aspect in OTM is a plain JavaScript object that defines at least three properties. The sequence and advice properties are also plain JavaScript functions parametrized by a join point and an environment. The sequence function matches a trace of join points of the application execution; the advice and kind properties are used for the same purpose as in AspectScript. In our example, the sAspLogin stateful aspect defines a sequence that matches four executions of the login function and a piece of advice that triggers an exception before the fourth execution of login.

Sequence Model. The function that represents a sequence can return (a) an environment if it matches a trace of join points, (b) a pair composed of a function and an environment if the sequence advances in its matching, or (c) false otherwise. The pair returned by the sequence establishes what the next step will be within the matching process. Based on the definition of a sequence, an AspectScript pointcut is also a sequence because a pointcut returns an environment or false, which is a subset of what a sequence returns.

```
var seqn = function(subSeq,n) {
  return function(jp,env) {
    var mainSeq = subSeq;
    for (var i = 0; i < n; ++i)
       mainSeq = seq(subSeq,mainSeq);
    return mainSeq;
} };
```

The piece of code above shows the definition of the seqn sequence designator (*i.e.* a function that returns a sequence). This sequence designator is just a convenient abstraction on the top of the seq binary sequence designator. For example, the evaluation of seqn(lg,4) becomes seq(lg,seq(lg,seq(lg,lg)). We illustrate the sequence model of OTM with the seq sequence designator. This sequence designator takes two (sub)sequences as parameters and returns a sequence that matches a (sub)sequence followed by another. As shown in the piece of code below, the sequence returns a pair composed of the

[10] Using Ajax technologies, malicious applications can insert several credentials without reloading the current Web page.



<seq(lg,seq(lg,seq(lg,lg))), env>
<seq(lg,seq(lg,lg)), env'>
<seq(lg,lg), env''>
<lg, env'''>
env''''
login
Legend:
<sequence, env> A pair composed of a sequence and an environment
login An execution Join point of the login function

Fig. 5. Matching an execution trace.

right function and the result environment if left matches. The sequence returns false if left does not match. Note that the sequence never returns an environment (*i.e.* the sequence matches) because the responsibility to match is delegated to the right sequence, which is evaluated with the result environment.

```
var seq = function(left,right) {
  return function(jp,env) {
    var result = left(jp,env);
    if (isEnv(result))
      return [right,result];
    return false; } };
```

Fig. 5 shows how the pair composed of a sequence and an environment of the sAspLogin aspect varies throughout the matching of a trace of four executions of the login function. This pair varies every time the sequence matches a login execution. In the beginning, sAspLogin begins with the pair composed of the sequence expressed by the programmer and an empty environment. The first time login is executed, the pair varies to a new pair composed of a sequence that only matches the three login executions and an environment possibly modified. The third time login is executed, the sequence of the pair is only the lg sequence. Finally, the fourth time login is executed, there is only an environment, meaning that the whole sequence matches; therefore, the stateful aspect executes its advice.

Openness. OTM follows open implementation principles [27]. Thereby, OTM allows developers to customize the crucial semantics of stateful aspects, like the spawning of *matchers*. A matcher is an internal component of a stateful aspect that carries out the task of matching a sequence. A stateful aspect can have several matchers to match several sequences at time (*e.g.* multiple users trying to login at time). The semantics of spawning decides when a stateful aspect adds a new matcher (more on this in Section 5.2). Apart from the spawning semantics, other semantics of stateful aspects can be customized [14], but these customizations beyond the scope this paper.

Contribution for WeCa. This section showed the flexibility to express (stateful) aspects in AspectScript and OTM through the modularization of crosscutting concerns in WebApps. However, WeCa still needs some distributed computing concepts to address causality issues appropriately. The next section explains these concepts.

3.2. Distributed computing

In distributed systems, processes communicate with each other using messages that are sent over the network. The sending and receiving of these messages as well as beginnings and ends of executions of functions are considered events of a distributed computation. Whereas we can observe a total order among events of a single process, there is no global clock (or perfectly synchronized local clocks) that allows us to observe a total order between events of different processes. Nevertheless, it is possible to observe a *partial order* between events if we use the *happened before* model proposed by Lamport [4].

Aspects for distributed computations. In AOP, join points are execution points which correspond to events. Traditionally (stateful) aspects react to join points of a single process. To react to *distributed* traces of join points, they have to match join points of different processes; and to react to distributed causal relations, (stateful) aspects have to consider the causal relations among these join points. In this section, we explain distributed computing concepts using AOP terminology in order to understand how aspects can react to distributed causal relations. For example, we explain the happened before model using join points instead of events:

$$jp_1 \rightarrow jp_2.$$

The happened before model defines a relation between two join points noted with an arrow. The arrow indicates a join point jp_1 causes another join point jp_2, meaning there is a causal relation from jp_1 to jp_2. For example, Fig. 6 shows in both scenarios that the join point jp_{s1} causes the join points jp_{s2}, jp_{r1}, and jp_{r2} because the join point jp_{s1} *happened* before.

The happened before model makes it possible to address causality issues such as the need (a) to control the causality between messages sent and responses received and (b) to react to distributed causal relations.

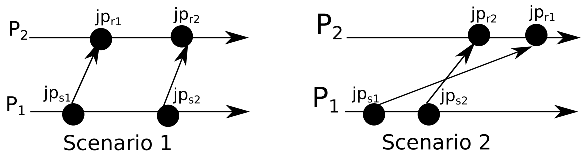


Fig. 6. Two possible distributed computations for the same input.

3.2.1. Controlling message causality

Distributed systems are difficult to test because of their nondeterministic nature, that is, these systems may exhibit multiple behaviors for the same input. This nondeterminism is caused by an arbitrary ordering of messages received by processes in different distributed computations. Fig. 6 shows that a process P_1 sends two messages, represented by jp_{s1} and jp_{s2} join points, to a process P_2. P_2 can receive these messages, represented by jp_{r1} and jp_{r2} join points, in two different orders arbitrarily. Fortunately, message ordering strategies [3] can be used to control the message causality. We now detail some of these strategies:

FIFO. Any two messages from a process P_i to P_j are received in the same order as they are sent. In a formal manner, let jp_{s1} and jp_{s2} be any two join points sent and $jp_{s1} \to jp_{s2}$, then the jp_{r2} join point cannot be observed by any process before the jp_{r1}.

$$jp_{s1} \to jp_{s2} \Rightarrow \neg(jp_{r2} \to jp_{r1}).$$

Discard Late. Any two messages from a process P_i to P_j are received in the same order as they are sent or only the most recent message is received. In a formal manner, this strategy is just an extension of FIFO: if jp_{r2} is observed first, jp_{r1} cannot be observed by any process.

$$jp_{s1} \to jp_{s2} \Rightarrow \neg(jp_{r2} \to jp_{r1}) \vee \neg jp_{r1}.$$

Discard Early. Any two messages from a process P_i to P_j are received in the same order as they are sent or only the oldest message is received. In a formal manner, this strategy is just an extension of FIFO: if jp_{r1} is observed first, jp_{r2} cannot be observed by any process.

$$jp_{s1} \to jp_{s2} \Rightarrow \neg(jp_{r2} \to jp_{r1}) \vee \neg jp_{r2}.$$

3.2.2. Reacting to distributed causal relations

To react to distributed causal relations, stateful aspects have to match distributed traces of join points and to consider causal relations between these join points. Unfortunately, the absence of a total order among these join points does not allow stateful aspects to consider causal relations.

Fortunately, the algorithm of vector clocks [4,12] can allow stateful aspects to observe a partial order. We now explain a straightforward adaptation of this algorithm to allow stateful aspects to consider causal relations between join points of distributed traces.

To support vector clocks, every join point jp is tagged with an array of counters of size n, where n is the number of processes, that is provided by a process P_i. This array represents a vector clock V, which is accessed by $V(jp)$, and is filled according to the following algorithm:

1. Initially, $V_i[k] = 0$ for $k = 1, \ldots, n$.
2. On each join point that does not represent the sending or receiving of a message, process P_i increases V_i as follows: $V_i[i] = V_i[i] + 1$.
3. On each join point jp that represents the sending of a message m, process P_i updates V_i as in (2) and attaches the new vector clock to jp.
4. On each join point jp that represents the receiving of a message m, process P_i increases V_i as in (2). Then, for all k, P_i updates its current V_i as follows: $V_i = max\{V_i, V(jp)\}$.

If all join points are tagged according to this algorithm, it is possible to define rules that verify if two join points satisfy distributed causal relations such as *causal* and *concurrent*:

Causal rule. This rule allows us to verify if a join point jp_1 *caused* another join point jp_2. For example, Fig. 7 shows that jp_{12} caused jp_{22} and jp_{23}. In a formal manner, the causal rule is defined by the following relation:

$$jp_1 \to jp_2 \iff V(jp_1) < V(jp_2). \tag{1}$$

Where:

$$V(jp_1) < V(jp_2) \iff \forall k[V(jp_1)[k] \leq V(jp_2)[k]] \wedge \exists k'[V(jp_1)[k'] < V(jp_2)[k']].$$

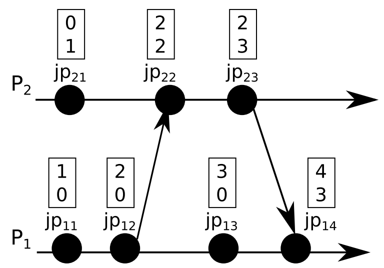


Fig. 7. Join points of a distributed computation tagged with vector clocks.

Concurrent rule. This rule allows us to verify that the jp_1 and jp_2 join points are *concurrent*: jp_1 does not cause jp_2 and vice versa. For example, Fig. 7 shows that jp_{22} and jp_{13} are concurrent. In a formal manner, the concurrent rule is defined by the following relation:

$$jp_1 \parallel jp_2 \iff \neg(jp_1 \rightarrow jp_2) \wedge \neg(jp_2 \rightarrow jp_1). \tag{2}$$

Contribution for WeCa. This section showed the necessary concepts to work on causality. The implementation of causality in distributed systems is a crosscutting concern because it is necessary to intercept, modify, and (possibly) postpone the evaluation of every join point. For example, the FIFO strategy postpones the evaluation of early join points. For this reason, the use of (stateful) aspects of AspectScript and OTM allows WeCa to modularize these causality concerns.

4. WeCa

This section describes WeCa, a practical library that allows for modular and flexible control over causality on the Web. This library combines stateful aspects, message ordering strategies, and vector clocks to allow JavaScript programmers to modularly and flexibly define (a) strategies that control the causality between Ajax requests and server responses and (b) monitors that react to distributed causal relations.

Fig. 4(b) shows that every WebApp has a WeCa runtime instance. Every instance observes join points generated on the application and other connected applications. If a programmer defines a message ordering strategy, the WeCa runtime enforces that server responses follow this strategy. In addition, if the programmer expresses a distributed causal relation pattern, the WeCa runtime frequently observes the distributed trace of join points to react whenever this trace is matched by such a pattern. We now explain how WeCa enforces message ordering strategies and reacts to distributed causal relations.

4.1. Controlling message causality

As mentioned in Section 2.1, a WebApp can send several Ajax requests to servers. However, this application can retrieve and process the server responses in an arbitrary order, therefore, this application behaves nondeterministically. The problem does not arise between the messages sent by different processes, the problem arises when a WebApp retrieves server responses (Fig. 1). Fortunately, message ordering strategies can also be used to control the causality between Ajax requests and server responses, and AspectScript aspects allow programmers to flexibly and modularly define these strategies. Using aspects and message ordering strategies, WeCa can be used, for example, to enforce server responses retrieved from a server that follows the Discard Early strategy (Section 2.1), which is provided as a library function in WeCa:

```
WeCa.deployStrategy(discardEarly,function(jp) {
  var request = jp.target;
  return request.url == "http://server.com";
});
```

In this piece of code, the deployStrategy method deploys the Discard Early strategy, which enforces server responses that come from http://server.com to follow this strategy. The two parameters of the deployStrategy method are plain JavaScript functions: the first represents the strategy and the second represents its scope. As message ordering strategies are plain JavaScript functions, the programmer can use the full power of the base language, in particular higher-order functions, to define a strategy. The second function of deployStrategy is parametrized by the join point that represents the execution of the onreadystatechange method, which processes the current server response. This function returns true if the server response must follow the strategy.

We now explain how WeCa implements and allows programmers to flexibly deploy these strategies.

Implementation details. Fig. 8 shows how WeCa only needs two aspects to support message ordering strategies. The first aspect, tagRequest, matches calls to the send method, which represents an Ajax request, of a request object. The piece of advice of this aspect tags the request object with a fresh and incremental identifier idServerResp that is generated to identify the associated server response.[11] The second aspect, strategy, matches the executions of the onreadystatechange method,

[11] As shown in the piece of code of Section 2, the context object of the execution of the server response is the request object.

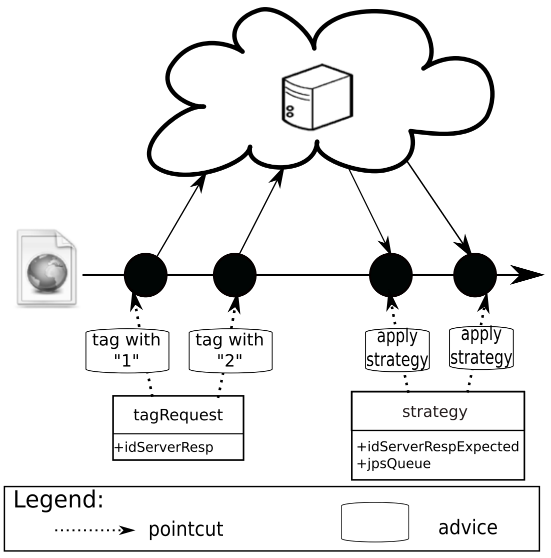


Fig. 8. Two aspects to apply a message ordering strategy.

which processes the server response of a request object. The piece of advice of this aspect is the actual message ordering strategy. This advice is defined by the programmer, who can use the idServerRespExpected and jpsQueue aspect instance variables to implement the strategy. The idServerRespExpected binding identifies the next server response expected, and jpsQueue is a queue of join points that contains methods to execute join point proceeds. As an example, the implementation of the Discard Early strategy is:

```
var discardEarly = function(jp,env) {
  var request = jp.target;
  if (this.idServerRespExpected == request.idServerResp) {
    jp.proceed();
    this.idServerRespExpected = request.idServerResp + 1;
} };
```

The jp join point represents the execution of the onreadystatechange method that processes the current server response. The proceed method is only executed if the server response is equal or older than the server response expected, *i.e.* if the server response is not early. Apart from the Discard Early strategy, WeCa provides other message ordering strategies as library functions.

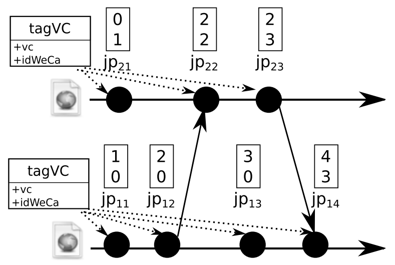
4.2. Reacting to distributed causal relations

As mentioned in Section 2.2 the analysis of the flow of information of social network applications is an active research topic. This flow of information is analyzed through the messages sent and received between users of these applications, meaning that it is necessary to observe and react to distributed causal relations given by these messages.

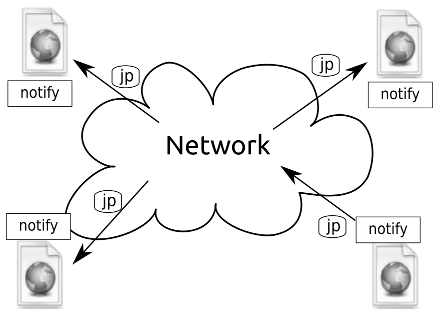
Fortunately, vector clocks can also be used to observe distributed causal relations in WebApps, and OTM stateful aspects allow programmers to react to these distributed causal relations in a flexible and modular manner. Using stateful aspects and vector clocks, WeCa can be used, for example, to deploy a stateful aspect that shows a message every time a follower retweets some tweet:

```
//necessary (sub)sequences to create the stateful aspect
var callTweet = function(jp,env) {
  if (jp.isCall() && jp.fun == tweet) {
    var tweet = jp.args[0]; //1st argument to tweet
    return env.bind("tweet",tweet);
  }
  return false;
};

var notifyRetweet = function(jp,env) {
  if (jp.isCustom("notification—call") && jp.fun == retweet) {
    var retweet = jp.args[0]; //1st argument to retweet
    return retweet == env.tweet? env: false;
  }
```



(a) The tagVC aspect that tags every join points with a vector clock.



(b) The notify aspect that notifies every join point to other applications.

Fig. 9. Aspects to support the matching of distributed causal relations.

```
  return false;
};


var sAspNotifyRetweet = {
  sequence: causalSeq(callTweet,notifyRetweet),
  advice: function(jp,env){
    alert("One of your followers retweeted:"+env.tweet);
  },
  kind: AFTER
};

WeCa.OTM.deploy(sAspNotifyRetweet); //deployment.
```

The sAspNotifyRetweet stateful aspect matches every time a tweet causes a retweet. The callTweet sequence returns an environment that contains the tweet, passed as parameter to the tweet function, if this sequence matches the call to tweet. The notifyRetweet sequence matches a *notification call join point*, a call join point that occurs in another connected application, of the call to the retweet function if both the tweet and retweet are equal. Finally, we use the causalSeq sequence designator to return a sequence that matches the two previous (sub)sequences if both satisfy the causal rule (Section 3.2.2).

We now explain how WeCa implements notification join points and sequences that match distributed causal relations.

Implementation details. WeCa only needs two aspects to react to distributed causal relations. Fig. 9(a) shows the first aspect, tagVC, which tags every join point with the corresponding vector clock. Fig. 9(b) shows the second aspect, notify, which notifies other WeCa runtime instances for every join point generated on the computation of a connected WebApp. When a WeCa runtime instance receives the information about a remote join point, this instance generates a notification join point of the corresponding kind,[12] which can be matched by any stateful aspect. The context information of a notification join point includes the join point generated plus the WeCa instance identifier. The communication between WeCa instance is carried out by a server application. This server application (a) broadcasts the notification join points, (b) can reset vector clocks of WeCa instances (which can be asked by these instances), and (c) gives a fresh and unique identifier to WeCa instance. For the last point, the server application notifies all instances to update their vector clocks whenever an instance is joined.

As we have shown in the previous piece of code, to react to distributed causal relations, WeCa uses OTM to define stateful aspects that react when they match distributed causal relations. In OTM, to define sequences that match distributed causal relations using the causal or concurrent rule (Section 3.2.2), it is necessary to have sequence designators like causalSeq to create sequences that consider causal relations. To achieve this, we only have to extend the seq sequence designator (Section 3.1.2):

```
var seq = function(left,right) {
  return function(jp,env) {
    var vc = jp.vc;  //current vector clock is stored
    var result = left(jp,env);
    if (isEnv(result))
      return [function(jpNext,env) {
        if ( vectorClockCondition(vc,jpNext.vc) )
          return right(jpNext,env);
        return false;},result];
```

[12] Like Ptolemy [28], AspectScript allows developers to explicitly trigger customized join points (used, in this case, to generate notification join points).

```
    return false;
} };
```

In Section 3.1.1, the seq sequence designator returns a sequence that if left matches this sequence directly returns right. Instead, this new seq returns a sequence that if left matches this sequence returns a function that first verifies if vector clocks of jp and jpNext satisfy some causal relation before evaluating right. The relation causal is verified through the vectorClockCondition function. Based on the piece of code above, we can create sequence designators like causalSeq, concSeq, or seq using higher-order functions:

```
var makeSeq = function(vectorClockCondition) {
  return function seq(left,right) {
    //the piece of code above
} } };
var causalSeq = makeSeq(caused);
var concSeq = makeSeq(areConcurrent);
var seq = makeSeq(function(vc1,vc2) {return true;});
```

The caused function verifies whether jp causes jpNext or not, and the areConcurrent function verifies whether both join points are concurrent or not. The functions caused and areConcurrent are developed according to the relations (1) and (2) described in Section 3.2.2. Finally, if we need a seq sequence designator that does not consider any causal relation, the makeSeq function is called with a function that always returns true. These sequence designators are provided as library functions in WeCa.

Verifications of causal and concurrent relations can be between any pair of join points (*e.g.* a notification join point and a local join point), therefore, every join point has to contain a vector clock. In other words, the comparison of vector clocks is a crosscutting concern that is addressed by an aspect in WeCa.

4.3. Summary

WeCa uses only four aspects in order to work. These four aspects extend AspectScript and OTM to address causality issues on the Web. Two aspects are used to support the definition of flexible message ordering strategies, and two aspects are used to react to distributed causal relations. Using the power of the base language, developers can define message ordering strategies and sequences that match distributed causal relations.

5. Revisiting Web 2.0 Applications

In Section 2, we have shown examples of how causality issues affect WebApps. This section presents how to address these issues using WeCa. First, we use WeCa to define appropriate message ordering strategies that address the different needs of WebApps described in Section 2.1. Second, we use WeCa to define a stateful aspect that determines the popularity of every tweet of a user (Section 2.2).

5.1. Controlling message causality

A WebApp can send several Ajax requests to different servers. However, this application can retrieve and process server responses in an arbitrary order. In Section 2.1, we presented three WebApps that needed three different message ordering strategies. This section presents and describes these strategies, which are provided as library functions in WeCa. As mentioned in Section 4.1, these strategies are really advice of an aspect. In addition, we extend one of these strategies to show the usefulness of the flexibility provided by WeCa.

A FIFO strategy to create a Web application from different servers. Programmers have to overcome the issue of creating an incorrect Web page due to an arbitrary order of server responses. The FIFO message ordering strategy, which processes the server responses in the same order as Ajax requests are sent, can ensure the Web page is always created correctly. This strategy postpones the use of an early server response until it is the expected one. We now present the function that describes the FIFO strategy:

```
var FIFO = function(jp,env){
  var request = jp.target;
  if (this.idServerRespExpected != request.idServerResp){
    this.jpsQueue.push(jp);
  }
  else{
    jp.proceed();
    this.idServerRespExpected = request.idServerResp + 1;

    //executing jp proceeds that can be executed now
    this.idServerRespExpected =
      this.jpsQueue.execRecEarlyJPs(this.idServerRespExpected);
   }
}

WeCa.deployStrategy(FIFO);
```

As mentioned in Section 4.1, the jp join point represents the execution of the onreadystatechange function. If this server response is not the expected one, jp is added to a queue and its proceed execution is postponed. Instead, if this server response is the expected one, the join point proceed is executed and the queue tries executing proceeds of join points stored because some of them can be the expected ones now (it is carried out by the execRecEarlyJPs method). This is so because if a join point proceed is executed inside of the queue, it can permit the execution of another join point proceed. As the scope is not specified in this deployment, the scope is global for this strategy, meaning that all server responses follow this strategy.

A Discard Late strategy for the visualization of streaming of videos. In the visualization of streaming videos, programmers have to overcome the issue of using obsolete data due to late server responses. A Discard Late strategy, which discards late server responses, always show a streaming video in a correct flow. We now present the function that describes the Discard Late strategy:

```
var discardLate = function(jp,env) {
   var request = jp.target;
   if (this.idServerRespExpected <= request.idServerResp) {
      jp.proceed();
      this.idServerRespExpected = request.idServerResp + 1;
   }
};

WeCa.deployStrategy(discardLate,function(jp) {
   var request = jp.target;
   return request.url == "http://streamingVideos.com";
});
```

If the server response is not late, the proceed of the jp join point is executed. If the server response is late, the join point proceed is not executed, meaning that the execution of the onreadystatechange method that processes this server response is discarded. Given the scope of the deployment, server responses that only come from http://streamingVideos.com must follow this strategy.

A Discard Early strategy for the visualization of thread posts. In the visualization, programmers also have to overcome the issue of using early data due to early server responses. Remembering the forum Web page which updates posts of last threads using Ajax requests. If some server responses are retrieved and displayed in an incorrect order, the beginning of the discussion of a thread may be misunderstood. A *Discard Early* strategy, which discards early server responses, can ensure the reader better understands the beginning of the discussion of every thread. In Section 4.1, we presented the Discard Early strategy; we now present an extension of this strategy, which executes a callback function when it discards a server response.

```
var deShowReadMore = discardEarlyCallback(function callback(jp,env) {
    //show the "read more" button
});

WeCa.deployStrategy(deShowReadMore,function (jp) {
   var request = jp.target;
   return request.url == "http://forum.com/threadlist";
});
```

We use the callback function to show a button "read more" when a server response is discarded. The piece of code below shows that the evaluation of the discardEarlyCallback function returns a function that uses the Discard Early strategy and executes a callback function, when a server response is discarded.

```
var discardEarlyCallback = function(callback) {
   return function(jp,env) {
    //invoke the discard early strategy

      var request = jp.target;

      //is it discarded?
      if (this.idServerRespExpected < request.idServerResp)
         callback(jp,env);
} };
```

5.2. Reacting to distributed causal relations

The analysis of the flow of information of WebApps is analyzed through the messages sent and received among users of these applications. In this section, we use a stateful aspect to calculate the popularity of tweets in Twitter (Section 2.2):
Tweet popularity. This feature allows a user to know the popularity of its tweets, which is measured by the number of retweets. As Fig. 2 shows, if a user publishes a tweet that is retweeted by a follower, the popularity of this tweet is increased by one. An analysis based on the distributed causal relations between tweets and retweets can determine how many users retweeted a tweet. The following stateful aspect counts the retweets of a tweet until the user clicks a button "popularity":

```
var callTweet = //... as Section 4.2

var retweetCount = function (jp,env){
  var result = notifyRetweet(jp,env); //notifyRetweet as Section 4.2
  if (isEnv(result))
    return env.counter == undefined?
      env.bind("counter",0): env.bind("counter",env.counter + 1);
  return false;
};

var callPopularity = function(jp,env) {
  return jp.isCall() && jp.fun == clickPopularityButton;
}
//necessary (sub)sequences to create the stateful aspect

var sAspTweetPopularity = {
  sequence: causalSeq(callTweet,repeatUntil(retweetCount,callPopularity)),
  advice: function(jp,env){
    addPopularity(env.tweet,env.counter);
  },
  kind: AFTER,
};

WeCa.OTM.deploy(sAspTweetPopularity);
```

The sAspTweetPopularity stateful aspect is only an extension of the stateful aspect shown in Section 4.2. This new stateful aspect counts the number of retweets of a tweet and adds this counter to the analyzed tweet. The sequence begins the matching when the callTweet (sub)sequence matches and continues with matchings of the retweetCount (sub)sequence until the user clicks the "popularity" button. Every time retweetCount matches, the counter of retweet is increased by one. The repeatUntil sequence designator, which is provided as a library function in OTM, returns a sequence that matches the first (sub)sequence several times until the second (sub)sequence matches.

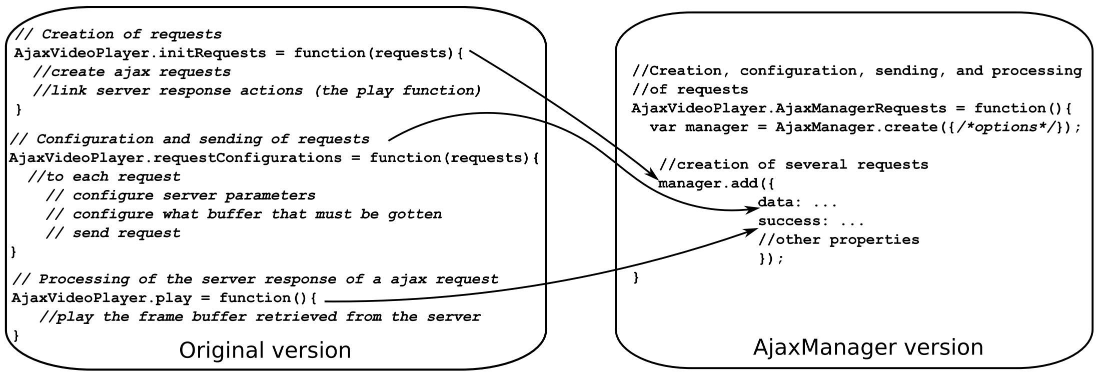
Although this stateful aspect works, it can only count the popularity of one tweet. A solution to count the popularity of every tweet could be to have a stateful aspect for every tweet. However, this solution is not very efficient because Twitter users commonly publish the same tweet several times.[13] A better solution would be that the same stateful aspect uses a different matcher (Section 3.1.2) of the sequence for every different tweet. As mentioned in Section 3.1.2, OTM allows developers to customize *when* a new matcher of the sequence is spawned. To customize the spawning of matchers in OTM, the spawn property is added to a stateful aspect definition. This property is a function that returns true if a new matcher is spawned and false otherwise. This function is parametrized by the candidate new matcher and current matchers of stateful aspects.

```
var sAspTweetPopularity = {
  //sequence, advice, and kind properties remain the same
  spawn: function(candidateMatcher,matchers){
    var candidateEnv = candidateMatcher.getEnv();
    var currentEnvs = getEnvs(matchers);

    return !currentEnvs.some(function(currentEnv){
      return currentEnv.tweet == candidateEnv.tweet;
    });
};
```

In our example, a new matcher is spawned when its associated environment binds a new tweet. The spawn function first gathers the environment of the candidate matcher and those of the current matchers. Then, spawn compares every

[13] http://holykaw.alltop.com/the-art-of-the-repeat-tweet.



```
// Creation of requests
AjaxVideoPlayer.initRequests = function(requests){
  //create ajax requests
  //link server response actions (the play function)
}
// Configuration and sending of requests
AjaxVideoPlayer.requestConfigurations = function(requests){
  //to each request
    // configure server parameters
    // configure what buffer that must be gotten
    // send request
}
// Processing of the server response of a ajax request
AjaxVideoPlayer.play = function(){
   //play the frame buffer retrieved from the server
}
```

```
//Creation, configuration, sending, and processing
//of requests
AjaxVideoPlayer.AjaxManagerRequests = function(){
  var manager = AjaxManager.create({/*options*/});

   //creation of several requests
   manager.add({
             data: ...
             success: ...
             //other properties
             });
}
```

Fig. 10. Modification of Ajax Video Player to support AjaxManager.

tweet of current matchers to the tweet of the candidate matcher, and if the latter tweet is different, the candidate matcher is spawned.

6. Evaluation

This section presents four evaluations of our proposal. First, we evaluate the modular use of WeCa. Second, we evaluate the effectiveness to control the order of server responses. In the third and fourth evaluation, we measure the WeCa overhead.

6.1. Modular use

Ajax Video Player [17] is a JavaScript application that allows a Web page to show streaming of videos. This application sends an Ajax request to a server which responds with a buffer of frames. When the associated server response is processed the application sends another Ajax request to retrieve the next buffer; this cycle is repeated. Sending an Ajax request at a time introduces significant interruptions during the display of a video. We extended this application to send simultaneous Ajax requests in order to decrease these interruptions. Later, we use WeCa and AjaxManager [6] to control the order of the server responses.

To work with WeCa, we only had to add the piece of code below, where the strategy variable is bound to a message ordering strategy:

```
WeCa.deployStrategy(strategy);
```

Fig. 10 shows the necessary modifications in Ajax Video Player to support AjaxManager. We had to create an AjaxManager object (Section 2.3) and modify functions related to the creation, configuration, sending, and processing of Ajax Video Player requests in order to wrap them. As the figure also shows, different concerns related to Ajax requests must be mixed in only one piece of code to support AjaxManager (see Appendix A and Appendix B for more details). In conclusion, the difference between both tools is that Ajax Video Player does not need to be modified to support WeCa.

6.2. Controlling server responses

As mentioned in the previous section, we extended Ajax Video Player to send simultaneous requests, which entail an unexpected order of server responses. Therefore, we use WeCa and AjaxManager to control the order of these responses.

We evaluate six different versions of Ajax Video Player: one request at a time, simultaneous requests, and four versions that use two different message ordering strategies of WeCa and AjaxManager. The application was tested with videos of different resolutions and lengths. The resolution of videos varies from 160x120 to 624x352 pixels and the length varies from 3 to 218 seconds (*i.e.* 03′38″). We measure the percentage of (a) interruptions while a video is displaying, (b) buffers that are incorrectly received (*i.e.* a buffer that is not the continuation of the previous one), and (c) buffers that are discarded because they are received late. For this experiment, we say that an interruption happens when a video is stopped until a new buffer is retrieved and processed.

As Table 1 shows, the version that sends one request at a time (original version) has more interruptions than other ones. When the original version is extended to support simultaneous requests, the interruptions decrease, but wrong buffers appear. The version that uses the Discard Late strategy of WeCa significantly decreases the interruptions, but some buffers are discarded. The version that uses the FIFO strategy of WeCa does not contain wrong and discarded buffers, but the interruptions are double of the previous version with WeCa. The AjaxManager versions behave similarly to WeCa, except for the FIFO strategy, which shows wrong buffers due to a bug in AjaxManager. To conclude, a version of Ajax Video Player that supports simultaneous requests improves the fluency of a streaming video; and WeCa or AjaxManager, with similar results, can be used to control the order of the buffers of frames.

Table 1
Evaluations of different versions of Ajax Video Player.

Versions/Evaluations	(a) Interruptions (%)	(b) Wrong buffers (%)	(c) Discarded buffers (%)
One request (original)	20.9	0	0
Simultaneous requests	18.79	18.29	0
WeCa Late	5.26	0	10.9
WeCa FIFO	11.77	0	0
AjaxManager Late	8.77	0	6.14
AjaxManager FIFO	13.53	7.39	0

Table 2
Benchmarks for the control of message causality.

Strategy/Server kind	Local (%)	Remote (%)
WeCa FIFO	338.3	2.9
WeCa Late	315.7	2.2
WeCa Early	338.2	2.5
AjaxManager FIFO	14	3.7
AjaxManager Late	0.5	1.7
AjaxManager Early	Not supported	Not supported

6.3. *WeCa overhead*

We ran tests of our library to evaluate the overhead of the control of message causality and reaction to distributed causal relations. For these tests, we used three machines: *a*, *b*, and *c*. The machine *a* is in Chile (Santiago) and is an Intel Core 2 Duo, 2.66 GHz with 2GB of RAM. The machines *b* and *c* are in USA (Newark) and are Intel(r) Xeon (R), 2.27 GHz with 512MB of RAM. These machines are running on Ubuntu 10.04 with Firefox 12.

6.3.1. *Overhead of controlling message causality*

We measured the overhead time of forcing each message ordering strategy presented in this paper and compared these results to AjaxManager. To measure this overhead, we develop a Web application that sends 10,000 identical Ajax requests to a server application, and we use the WeCa message ordering strategies to order the associated server responses. To execute these tests, we first used the machine *a* for the Web and server application, therefore, the communication is local between both. Later, we used the machine *a* for the Web application and the machine *b* for the server application, meaning that the wide-area network is used. Finally, we compare these results to AjaxManager.

Table 2 shows a clear difference in the overhead between AjaxManager and WeCa when the Web and server application are in the same machine: 7.25% for AjaxManager and 330.3% for WeCa on average. This is so because the aspect used by WeCa observes and tries matching every join point of the application execution. This difference practically disappears when the server application is on a remote machine because of the latency of the wide-area network. The observed overhead is comparable to that produced by AjaxManager. These results indicate that WeCa is useful for most current Web applications, where the network latency overshadows the induced overhead.

6.3.2. *Overhead of reacting to distributed causal relations*

This section presents the overhead of time and space introduced by WeCa to match (and react to) sequences that satisfy distributed causal relations.

Time. We measured the time overhead of the insertion of vector clocks to join points plus the comparison between these vectors. For this experiment, we develop a Web application that sends empty messages to other Web applications. This application is deployed on the machines *a* and *b*. The messages are sent through a server application that is in the machine *c*. The Web applications contain a WeCa instance. In addition, the Web application of the machine *a* deploys a stateful aspect to non-simultaneously match 1,000 causal and concurrent sequences composed from 5 to 30 join points.

Fig. 11 shows the increment of the overhead of our library when the length of a sequence of join points enlarges. This is so because there are more insertions and comparisons of vector clocks when a sequence is longer. In addition, the figure shows that the overhead is greater when the concurrent relation is verified; this is because the concurrent rule requires one comparison more than the causal rule (Section 3.2.2). Note that as in the previous experiment, if we consider network latency, the observed overhead is negligible. Finally, we do not compare these results with other approaches because, to the best of our knowledge, WeCa is the only tool that reacts to distributed causal relations on the Web at runtime.

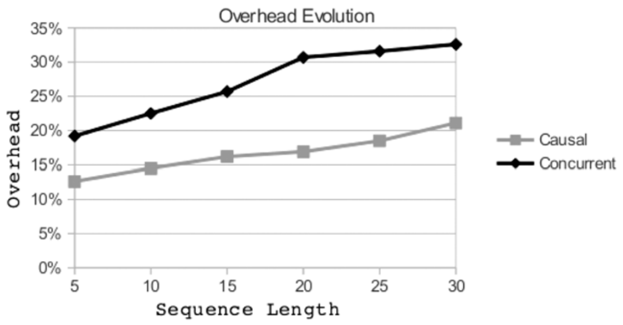


Fig. 11. Benchmarks for the reaction to distributed causal relations.

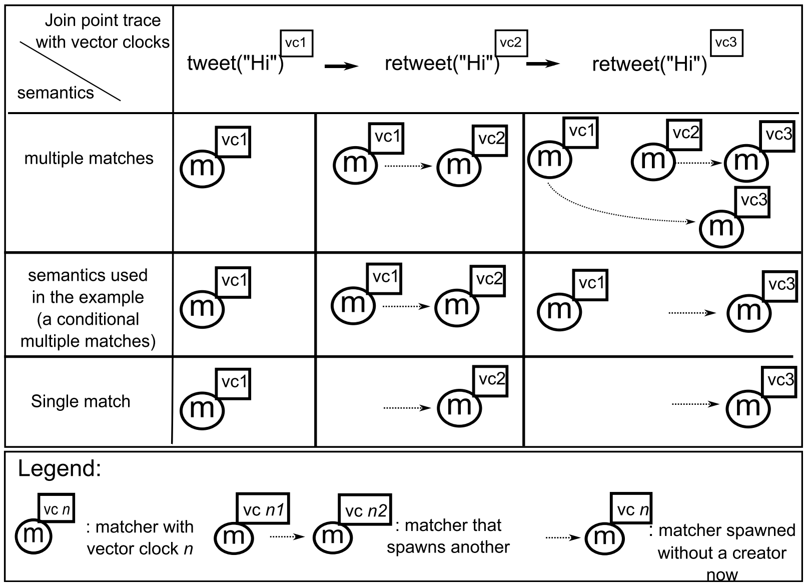

Join point trace with vector clocks
semantics
tweet("Hi")
retweet("Hi")
retweet("Hi")
vc1
vc2
vc3
multiple matches
semantics used in the example (a conditional multiple matches)
Single match
m
Legend:
vc n
: matcher with vector clock n
vc n1
vc n2
: matcher that spawns another
: matcher spawned without a creator now

Fig. 12. Number of vector clocks stored according to the spawning semantics used. In this figure, rows show different spawning semantics of the stateful aspect described in Section 5.2, and columns contain the join points of the trace $tweet(\text{“}Hi\text{”}) \rightarrow retweet(\text{“}Hi\text{”}) \rightarrow retweet(\text{“}Ho\text{”})$.

Space. As a first approximation of space overhead, it seems that the space needed to support vector clocks is only an array of size *n* (by application), where *n* is the number of current Web applications connected. A deeper analysis however reveals that more arrays need to be stored in order to compare the vector clocks of different application messages. The seq implementation of Section 4.2 illustrates such a situation: when the left sequence matches a join point, its associated vector clock has to be stored to compare it to the join point vector clock that the right sequence will potentially match. Because every matcher has a vector clock, the number of vector clocks that must be stored therefore depends on the number of matchers. This number is in turn determined by the spawning semantics of an aspect. Fig. 12 illustrates that three spawning semantics store different numbers of vector clocks. Using the semantics of *multiple matches*, the stateful aspect keeps all matchers in order to maximize the matches of the specified sequence. With the semantics used in Section 5.2, the aspect only keeps the first matcher and matchers that represent different tweets of a user. Finally, the *single match* semantics only keeps one matcher at a time. In conclusion, the space overhead increases if a stateful aspect keeps more potential matches of its sequence.

7. Related work

Although the relation between AOP and distributed computing to address the causality issues on the Web is a distinguishing contribution of this paper, there are already proposals that address these issues using different techniques. We now review JavaScript libraries to control the causality between Ajax requests and server responses and postmortem tools to observe distributed causal relations. In addition, we also review a proposal to address causality issues in distributed systems using AOP.

Libraries to control the message causality. A number of lightweight JavaScript libraries have been developed by programmers that rely on function wrappers. In the simplest case, Rico [7], a domain-specific library implicitly uses the FIFO message ordering strategy to sort server responses that are shown in an HTML table. A more elaborate case is AjaxManager [6],[14] a general-purpose library that offers a fixed set of message ordering strategies. As mentioned in Section 2.3, every Ajax request of a WebApp must manually be written to follow a strategy in AjaxManager. WeCa uses the AOP interception to transparently enforce strategies, *i.e.* Ajax requests are not manually written. In addition, WeCa provides as library functions the strategies available in AjaxManager, and our proposal also allows programmers to customize/add new strategies (*e.g.* the Discard Early strategy). Unlike WeCa, extending the AjaxManager library to customize/add strategies is not flexible due to the library does not provide the computation of the processing of Ajax responses as first-class values. The absence of these first-class values does not allow programmers to implement strategies using appropriated abstractions (*i.e.*. server response computations).

Frameworks to observe distributed causal relations. On the one hand, a large number of tools based on dynamic graphs are available to observe distributed causal relations in interactive WebApps [8]. On the other hand, Kossinets et al. [29] use a modified version of vector clocks, which uses timestamp instead of counters, to analyze the minimum time required for information to spread from one user to another. However, these proposals are post-mortem. Therefore, they cannot react to distributed causal relations at runtime, which is essential for companies that use social networks to promote their new products/services [23].

Using AOP in distributed systems. The extension of AWED [30], a system that explicitly supports the monitoring of distributed computations in Java, takes into consideration distributed causal relations in tasks of debugging and testing of middleware [31]. Although AWED is not developed to address causality issues on the Web, it is a related proposal for WeCa because AWED uses vector clocks and stateful aspects to react to distributed causal relations. Unlike the flexibility provided by WeCa, AWED stateful aspects are expressed using a (limited) domain-specific language[15]; therefore, the stateful aspects semantics cannot be modified in order to take into account different discarding policies. In particular, AWED only uses the FIFO strategy in an implicit way and does not present a modular (or semantic) separation between enforcing message ordering strategies and reacting to distributed causal relations.

8. Conclusions

We have presented WeCa, a practical library that addresses flexibly and modularly some causality issues on the Web through stateful aspects, ordering message strategies, and vector clocks. Concretely, our proposal allows JavaScript developers (a) to control the causality between Ajax requests and server responses and (b) to react to distributed causal relations in WebApps. To the best of our knowledge, WeCa is the first (practical) attempt to address these causality issues on the Web in a modular and flexible manner. The use of previous proposals are crosscutting, inflexible, and/or largely ignore the distributed computing theory.

We showed that, contrary to current proposals, WeCa uses general-purpose approaches instead of ad hoc solutions. The key element for this is the flexibility of AspectScript and OTM to define aspects and stateful aspects respectively. This flexibility makes it possible the extension of AspectScript and OTM using plain aspects that enable the supporting of message ordering strategies and vector clocks. To validate our proposal, we used WeCa with several practical examples from the realm of WebApps. For example, we analyzed the flow of information in WebApps like Twitter. In addition, we extended a third-party Web application, Ajax Video Player, to support different message ordering strategies provided the WeCa library. Finally, we ran benchmarks in order to test how practical is WeCa for the real Web applications.

Many WebApps are highly interactive in nature and are not performance-sensitive. For the other applications, it is important to consider the WeCa performance. We plan to improve performance through the use of partial evaluation techniques for higher-order languages like JavaScript [32].

Availability. WeCa, along with the examples presented in this paper, is available online at http://pleiad.cl/weca. Our proposal currently supports the Mozilla Firefox browser without the need of an extension.

Appendix A. Ajax Video Player

The following piece of code shows the three main functions of Ajax Video Player (used in Fig. 10). We simplify these functions to improve readability of their implementations. The complete implementation can be found on http://pleiad.cl/otm/wiki/weca_ajaxvideoplayer.

```
/* 1) In the original code, this function is named "initXMLHTTPReq". Here we
change the name in order to clarify what this function really does.*/
AJAXVideoPlayer.initRequests = function (requests) {

  for (var indice = 0; indice < requests.length; ++indice) {
    requests[indice] = createRequest();
    requests[indice].onreadystatechange = AJAXVideoPlayer.play;
  }
```

14 AjaxManager is an active project: https://github.com/aFarkas/Ajaxmanager/graphs/traffic.

15 A regular expression language.

```
  AJAXVideoPlayer.configurationRequests(requests);
};


/* 2) In the original code, this function is named "sendRequest". Here we
change the name in order to clarify what this function really does.*/
AJAXVideoPlayer.RequestConfigurations = function (requests) {

  for (var indice = 0; indice < requests.length; ++indice) {
    requests[indice].open("POST", AJAXVideoPlayer.XMLBase64FramesService);
    requests[indice].setRequestHeader("Host", "localhost");
    requests[indice].setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    requests[indice].setRequestHeader("Content-Length", "12");
    requests[indice].send("strFileName=" + AJAXVideoPlayer.VideoFileName +
              "&startPosition=" + getStart(indice) + "&endPosition=" + getEnd(indice));
  }
};


/* 3) In the original code, this function is named "reqDone". Here we
change the name in order to clarify what this function really does. */
AJAXVideoPlayer.play = function () {

  AJAXVideoPlayer.scene[AJAXVideoPlayer.sceneCacheCounter++] = this.responseXML;
  AJAXVideoPlayer.sceneCache = this.responseXML;
  AJAXVideoPlayer.animate();
};
```

Appendix B. Ajax Video Player with WeCa & AjaxManager

As we mentioned in Section 6.1, to support WeCa in Ajax Video Player, it is only necessary to add the following line at the beginning of the application:

```
WeCa.deployStrategy(WeCa.discardLateStrategy);
```

Instead, to support AjaxManager in Ajax Video Player, the developer has to remove the implementation of the functions initRequests and requestConfigurations (Appendix A) and create a new function. Like the previous appendix, we simplify the code in order to improve its readability.

```
AJAXVideoPlayer.AjaxManagerRequests = function (requestNumber) {

  /* Configuration for the discard late strategy in AjaxManager.
     NOTE: AjaxManager can only control a limited number of requests (property maxRequests) */
  var fifoManager = manageAjax.create({queue: clear, preventDoubleRequests:false, maxRequests: numberRequest});

  for (var indice = 0; indice < requestNumber; ++indice) {
    fifoManager.add({
                  complete: AJAXVideoPlayer.play,
                  url: AJAXVideoPlayer.XMLBase64FramesService,
                  contentType: "application/x-www-form-urlencoded",
                  type: "POST",
                  data: {strFileName: AJAXVideoPlayer.VideoFileName,
                         startPosition: getStart(indice), endPosition: getEnd(indice)},
                  beforeSend: function(xhr) {
                       xhr.setRequestHeader("Host", "localhost");
                       xhr.setRequestHeader("Content-Length", "12");
                  }
            });
} };
```

References

[1] T. O'Reilly, What is web 2.0: design patterns and business models for the next generation of software, Communications & Strategies (2005).
[2] J.J. Garrett, Ajax: A new approach to Web applications, 2005.
[3] V. Murty, V. Garg, Characterization of message ordering specifications and protocols, in: 17th IEEE International Conference on Distributed Computing Systems, ICDCS '97, IEEE Computer Society, Los Alamitos, CA, USA, 1997, pp. 492–499.
[4] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (1978) 558–565.
[5] F. Benevenuto, T. Rodrigues, M. Cha, V. Almeida, Characterizing user behavior in online social networks, in: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference, IMC '09, ACM, New York, NY, USA, 2009, pp. 49–62.
[6] A. Farkas, A JavaScript library to block, abort, queue, and synchronize Ajax requests, 2011.
[7] Rico, A JavaScript library for rich internet applications, 2011.
[8] Wikipedia, Social network analysis software, 2011.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (Eds.), Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP 97, in: Lecture Notes in Computer Science, vol. 1241, Springer-Verlag, Jyväskylä, Finland, 1997, pp. 220–242.

[10] V.K. Garg, Elements of Distributed Computing, John Wiley & Sons, Inc., New York, NY, USA, 2002.

[11] R. Douence, P. Fradet, M. Südholt, Trace-based aspects, in: R.E. Filman, T. Elrad, S. Clarke, M. Akşit (Eds.), Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005, pp. 201–217.

[12] F. Mattern, Virtual time and global states of distributed systems, in: C.M. et al. (Eds.), Proceeding Workshop on Parallel and Distributed Algorithms, North-Holland/Elsevier, 1989, pp. 215–226.

[13] R. Toledo, P. Leger, É Tanter, AspectScript: Expressive aspects for the Web, in: Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development, AOSD 2010, ACM Press, Rennes and Saint Malo, France, 2010, pp. 13–24.

[14] P. Leger, É. Tanter, An open trace-based mechanism, in: J. Aldrich, R. Massa (Eds.), Proceedings of the 14th Brazilian Symposium on Programming Languages, SBLP 2010, Salvador - Bahia, Brazil, 2010, pp. 123–138.

[15] P. Leger, É Tanter, Towards an open trace-baced mechanism, in: G.T. Leavens, S. Katz, M. Mezini (Eds.), Proceedings of the Ninth Workshop on Foundations of Aspect-Oriented Languages, FOAL 2010, Rennes and Saint Malo, France, 2010, pp. 25–30.

[16] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, Adding trace matching with free variables to AspectJ, in: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2005, ACM Press, San Diego, California, USA, 2005, pp. 345–364. ACM SIGPLAN Notices, 40(11).

[17] Shahin, Ajax Video Player, streaming over HTTP with Javascript and Ajax. http://www.codeproject.com/Articles/16092/Streaming-over-HTTP-with-JavaScript-AJAX-video-pla, 2006.

[18] J. Jiang, C. Wilson, X. Wang, P. Huang, W. Sha, Y. Dai, B.Y. Zhao, Understanding latent interactions in online social networks, in: Proceedings of the 10th Annual Conference on Internet Measurement, IMC '10, ACM, New York, NY, USA, 2010, pp. 369–382.

[19] C. Wilson, B. Boe, A. Sala, K.P. Puttaswamy, B.Y. Zhao, User interactions in social networks and their implications, in: Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys '09, ACM, New York, NY, USA, 2009, pp. 205–218.

[20] S. Wasserman, K. Faust, Social network analysis: methods and applications, 1st ed., in: Structural Analysis in the Social Sciences, vol. 8, Cambridge University Press, 1994.

[21] Mashable, Website for news in social and digital media, technology and web culture, 2010.

[22] P. Leger, É. Tanter, R. Douence, Javascript developers and message ordering strategies - http://pleiad.cl/otm/wiki/weca_js_mos, 2012.

[23] J. Owyang, J. Bernoff, C. Spivey, S. Wright, Online Community Best Practices — A Social Computing Report, Technical Report, Forrester, 2008.

[24] H. Masuhara, G. Kiczales, C. Dutchyn, A compilation and optimization model for aspect-oriented programs, in: G. Hedin (Ed.), Proceedings of Compiler Construction, CC2003, in: Lecture Notes in Computer Science, vol. 2622, Springer-Verlag, 2003, pp. 46–60.

[25] M. Wand, G. Kiczales, C. Dutchyn, A semantics for advice and dynamic join points in aspect-oriented programming, ACM Transactions on Programming Languages and Systems 26 (2004) 890–910.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, An overview of AspectJ, in: J.L. Knudsen (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP 2001, in: Lecture Notes in Computer Science, vol. 2072, Springer-Verlag, Budapest, Hungary, 2001, pp. 327–353.

[27] G. Kiczales, J. Lamping, C.V. Lopes, C. Maeda, A. Mendhekar, G. Murphy, Open implementation design guidelines, in: Proceedings of the 19th International Conference on Software Engineering, ICSE 97, ACM Press, Boston, Massachusetts, USA, 1997, pp. 481–490.

[28] H. Rajan, G.T. Leavens, Ptolemy: A language with quantified, typed events, in: J. Vitek (Ed.), Proceedings of the 22nd European Conference on Object-oriented Programming, ECOOP 2008, in: Lecture Notes in Computer Science, vol. 5142, Springer-Verlag, Paphos, Cyprus, 2008, pp. 155–179.

[29] G. Kossinets, J. Kleinberg, D. Watts, The structure of information pathways in a social communication network, in: Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '08, ACM, New York, NY, USA, 2008, pp. 435–443.

[30] L.D.B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, D. Suvée, Explicitly distributed aop using awed, in: Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development, AOSD 2006, ACM Press, Bonn, Germany, 2006, pp. 51–62.

[31] L.D. Benavides Navarro, R. Douence, M. Südholt, Debugging and testing middleware with aspect-based control-flow and causal patterns, in: Proceedings of the 9th ACM/IFIP/USENIX International Middleware Conference, in: Lecture Notes in Computer Science, vol. 5346, Springer-Verlag, Leuven, Belgium, 2008, pp. 183–202.

[32] M. Might, Y. Smaragdakis, D. Van Horn, Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis, in: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 305–315.