Co-Browsing Dynamic Web Pages

Dietwig Lowet
Philips Research Laboratories
High Tech Campus 34
Eindhoven, The Netherlands
0031-40-2749543
dietwig.lowet@philips.com

Daniel Goergen
Philips Research Laboratories
High Tech Campus 34
Eindhoven, The Netherlands
0031-40-2749537
daniel.goergen@philips.com

ABSTRACT
Collaborative browsing, or co-browsing, is the co-navigation of the web with other people at-a-distance, supported by software that takes care of synchronizing the browsers. Current state-of-the-art solutions are able to do co-browsing of "static web pages", and do not support the synchronization of JavaScript interactions. However, currently many web pages use JavaScript and Ajax techniques to create highly dynamic and interactive web applications. In this paper, we describe two approaches for co-browsing that both support the synchronization of the JavaScript and Ajax interactions of dynamic web pages. One approach is based on synchronizing the output of the JavaScript engine by sending over the changes made on the DOM tree. The other approach is based on synchronizing the input of the JavaScript engine by synchronizing UI events and incoming data. Since the latter solution offers a better user experience and is more scalable, it is elaborated in more detail. An important aspect of both approaches is that they operate at the DOM level. Therefore, the client-side can be implemented in JavaScript and no browser extensions are required. To the best of the authors' knowledge this is the first DOM-level co-browsing solution that also enables co-browsing of the dynamic interaction parts of web pages. The presented co-browsing solution has been implemented in a research demonstrator which allows users to do co-browsing of web-applications on browser-based networked televisions.

Categories and Subject Descriptors
H.5.3 [**Information interfaces and presentation**]: Group and Organization Interfaces – *Computer supported cooperative work.*

General Terms
Algorithms

Keywords
Co-browsing, shared browsing, collaborative computing, Web4CE, collaboration

1. INTRODUCTION
The browser is turning into a ubiquitous platform for providing users access to data, services and applications. The browser is moving into the direction of a thin client computing platform in the PC domain and also gained importance in the mobile domain and will be present on networked TVs (based e.g. on Web4CE [1]) in the near future. Conjoined with the trend of the increasing amount of web-based multimedia services, a browser-based IPTV platform [9] is the logical next step.

The solution presented in this paper has been developed for a browser-based TV platform based onWeb4CE, but applies equally well for PC and mobile browsers. We see the browser-based TV platform not only as a possibility to bring well-established internet services and new TV related services to the user, but also as an opportunity to enable new types of services and user experiences. One aspect here is to enable the user to share content and experiences with their family and friends at distant locations from the comfort of their couch. Examples of experience sharing are watching pictures together (as depicted in Figure 1), watching online video clips together, doing online shopping together or playing games together.

Figure 1 Two couples watching pictures together

Bringing synchronous experience sharing to the user can be done in two ways. A first way is to develop new multi-user applications developed with multiple users in mind. But this would only be done for a small amount of web applications and will not include the vast amount of available web applications already out there. A second way is to develop a generic mechanism for synchronizing existing "single-user" web applications between two or more browsers. There are many "single-user" web applications for which synchronous sharing is an interesting option. Watching pictures together on Flickr[1], using Google Maps[2] for planning a trip together and choosing a movie together on a movie theatre web site, are some examples. To avoid the adaptation of all these single-user services, a generic co-browsing mechanism for synchronizing single-user web applications between two or more browsers is needed. As indicated in Figure 2, a generic co-browsing mechanism provides a low development cost solution for sharing many interesting applications together.

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

[1] http://www.flickr.com

[2] http://maps.google.com

Figure 2 Benefits of a generic co-browsing solution

Due to the extensive use of JavaScript and Ajax[3], many web pages are better described as a web application rather than as a web document. Notable examples are online word processors like Google Docs[4], the IM web client Meebo[5], and the Google e-mail web client[6]. But the use of these technologies also causes state-of-the-art co-browsing solutions as [4] and [7] to be not suited for such type of web applications, since they have been developed for static web pages. They do not support the synchronization of the dynamic parts and the JavaScript and Ajax interactions of web pages.

The main contribution of this work is that is presents two co-browsing solutions that enable the synchronization of Ajax-based web-applications which make extensive use of JavaScript and XMLHttpRequest interactions. Both solutions are based on synchronizing the browsers at the DOM level. The first solution synchronizes the browsers by directly synchronizing the changes made on the DOM tree (*JavaScript engine output synchronization*). The second solution synchronizes the browser by synchronizing UI events and incoming data and thus keeping the JavaScript engines synchronized (*JavaScript engine input synchronization*). Though the latter solution is potentially less robust it offers the better user experience and is more scalable. Both solutions can be implemented in JavaScript and require no extensions to a normal browsers. The presented solution has been developed for a internet-connected, browser-based TV. Such a platform has some limitations compared to the PC domain. One important aspect is that it is not as simple as on a PC to install additional plug-ins in a browser or even install a new browser on a TV. Moreover, the solution should work across browsers from different vendors and CE manufacturers adhering to the same (minimal) standard (e.g. Web4CE [1]). Due to that, we strive for a solution that requires no changes to the browser and can be implemented completely in JavaScript. Thus, the presented DOM level solution is a perfect match.

Besides the TV platform, a co-browsing solution is also applicable for other platforms, like PCs and mobile phones, which provide access to web-based application through a browser. As indicated above, a co-browsing solution can be applied for experience sharing applications in the consumer domain, but can also be used for collaboration frameworks in the professional domain. Here, next to frameworks that require a browser plug-in, like e.g. WebEx, a number of collaborative frameworks are currently emerging which use only the native browser at the client side, such as [11]. Sharing single-user web pages is clearly a valuable addition to any collaboration platform.

This paper is structured as follows. The next section gives an overview of the conceptual model of a browser. This model is used in Section 3 to provide an overview of JavaScript engine input and output synchronization as two methods for co-browsing dynamic (JavaScripted) web pages. This section also gives an overview of different deployment options and compares the two basic methods. In Section 4, the method based on JavaScript engine input synchronization is described and discussed in more detail. Section 5 gives a short description of our proof-of-concept demonstrator, based on Firefox, which makes use of the presented co-browsing solution in an experience sharing framework for browser-based networked TVs. An overview on related work is given in Section 6 and the paper concludes with Section 7.

2. BROWSER MODEL

In this section, we will give a conceptual model of an HTML browser for the purpose of explaining different possible co-browsing solutions. This gives an insight into how two or more browsers can be synchronized for co-browsing. The information provided here is based mostly on documentation of the Firefox browser[7] and in some details it may not be completely accurate for other browsers.

2.1 Rendering a web page

What happens when the browser is directed to a new URL? Figure 3 gives an overview of the following description. First, the browser starts fetching the data from the network, using the networking library. As the data for the page streams in, it is transferred to the layout engine, which typically runs in a separate thread, the UI thread. The incoming HTML is parsed and a DOM tree representation is constructed. In addition to building up a DOM tree, modern CSS2[8]-compliant browsers also build up separate rendering trees. Based on these two trees, the frame tree is built. Once the frame tree is built, the "reflow" process determines where the frames have to be displayed on the screen. In a last step, the screen is painted, which is typically performed by making use of the local platform graphics primitives and widgets.

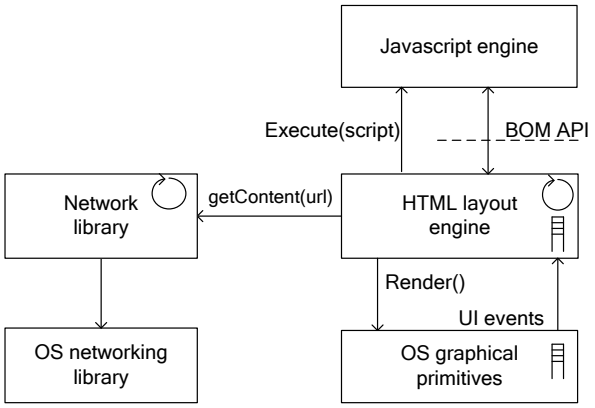

Figure 3 Conceptual overview of a Mozilla-based browser

After the page is loaded and rendered, the user can start interacting with the page. Actions like moving the mouse, scrolling a frame, filling-in a form or clicking on a link will generate UI events. The native OS windowing system will forward these events via the widget library to the layout engine. The layout engine will update the browser screen area accordingly and if necessary dispatch JavaScript UI events to registered event handlers. The term JavaScript UI event is used for the events a browser generates and sends to the JavaScript engine.

[3] Asynchronous JavaScript and XML

[4] http://docs.google.com/

[5] http://www.meebo.com

[6] http://gmail.com

[7] http://www.mozilla.org/newlayout/doc/

[8] http://www.w3.org/TR/REC-CSS2/

2.2 Interaction between the layout engine and the JavaScript engine

As shown in Figure 3, the HTML layout engine of a browser also takes care that the JavaScript included in the web page is executed by invoking the JavaScript engine. Execution of JavaScript code starts when the page is loaded[9]. The layout engine initiates the execution of all "initial" JavaScript code contained in a page. After this "initial" JavaScript is executed, the remaining JavaScript, i.e. the event handlers or the functions that have been scheduled by a timer, will be invoked by the layout engine when the corresponding event occurs or the corresponding timer expires. The JavaScript code embedded in a web page can influence the layout engine or even the browser in general by means of the following "standardized" objects, which are collectively known as the Browser Object Model (BOM). The main examples are:

- *Window object:*

  Interactions with the browser window can be done via the window[10] object. Examples are *window.location*, to redirect the browser to another URL, *window.scroll(),* to scroll the window, and *window.setTimeout()* - to tell the JavaScript engine to evaluate an expression after a specified amount of time.

- *Document object (DOM):*

  The document object represents the entire HTML document and can be used to access and change all elements in a web page. Examples are adding nodes, changing style attributes, adding event listeners etc. For a complete description of the document object, see [12].

- *XMLHttpRequest object:*

  This object allows JavaScript code to send and receive data to/from an HTTP server by creating an XMLHttpRequest object and calling its *open()* and *send()* functions.

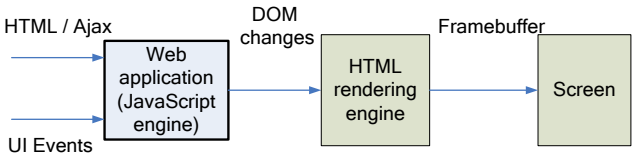

Figure 4 Browser model from the JavaScript engine perspective

However, from the perspective of the JavaScript engine a simpler browser model can be used as depicted in Figure 4. The JavaScript engine conceptually has two types of input: incoming data from the HTML service via the HTTP protocol and via the DOM and UI input from the user via the native graphical primitives and the DOM. The browser can be considered to consist of two parts: (1) a JavaScript engine which executes the application logic and produces HTML that is rendered by (2) the HTML rendering engine to the screen.

[9] JavaScript code can already be executed before the complete page has been loaded, similar to the way an initial part of the page can already be rendered before the complete page is loaded. It is completely up to the browser to decide when to start displaying content and executing JavaScript.

[10] http://www.w3schools.com/HTMLDOM/dom_obj_window.asp

3. CO-BROWSING SOLUTIONS

As described before, the co-browsing solution should support the synchronization of the JavaScript and XMLHttpRequest interactions of dynamic web pages. Based on the discussion in the previous section, this leads to two main methods for synchronizing scripted web pages: (1) Synchronization of the JavaScript engine input, i.e. UI event and HTML/ XMLHttpRequest synchronization, and (2) Synchronizing the JavaScript engine output, i.e. DOM tree synchronization[11]. As depicted Figure 5, the different methods reside on different levels.

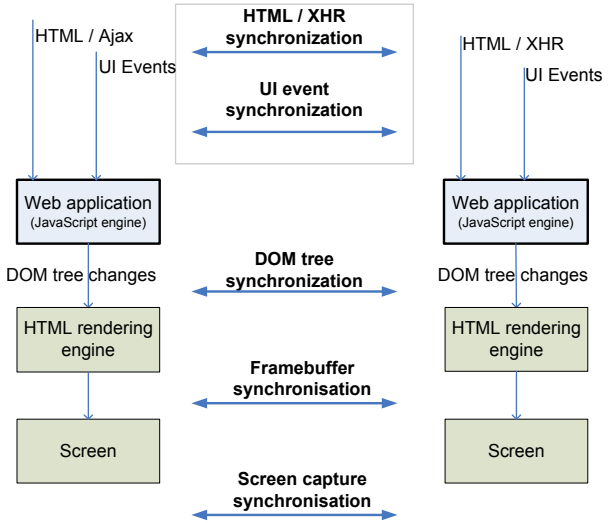

Figure 5 Overview of basic co-browsing options. (XHR stands for XMLHttpRequest)

1. *JavaScript engine input synchronization:*

   This method relies on keeping the JavaScript engines in all involved browsers synchronized, and thus indirectly also the DOM tree. To do this, both the data that the browsers receive from the web server and the JavaScript UI events happening at all browsers must be synchronized. UI event synchronization means that all user events injected by the layout engine into the JavaScript engine of one browser must also be injected into the JavaScript engine of the other browsers in the same order. Consequently, the JavaScript engine of every browser is running and only JavaScript UI events need to be sent over. If the co-browsed web service adheres to the web model in which each URL points to a unique piece of information the HTML/XMLHttpRequest data synchronization is automatically cared for. However, for HTML services that sometimes return different data for the same URL, or keep complicated state information (e.g. a transaction in an online shopping site), a co-browse proxy is needed. The co-browse proxy ensures that all the co-browsers receive the same data[12] and that, for the co-browsed HTML service, the co-browsers look like one single browser.

[11] Capturing the screen using e.g. an additional camera and doing framebuffer-level synchronization are non co-browsing solutions and will not be discussed further.

[12] There are timing issues which needs additional synchronization. This will be discussed in detail in section 4.

2. *JavaScript engine output synchronization:*

   In this method, the co-browsing code listens for DOM tree changes in one browser, the *reference browser*. If the JavaScript engine of the reference browser updates the DOM tree, this DOM tree update is sent to the other browsers, the *client browsers*. The co-browsing code in the client browsers receives incoming DOM tree updates and uses the DOM interface to adapt the DOM tree of the web application. In this solution, only one browser communicates with the web server. This also means that transferring control to another browser is not trivial, because it would involve sending over the state of the JavaScript engine from the reference browser to one of the client browsers. If all ends are allowed to interact with the web-application[13], all UI events have to be sent to the reference browser. As opposed to method 1, the events are only executed in the reference browser. Only the reference browser is visible to the co-browsed HTML service, so that also in thus variant only one request to the service is generated for all co-browsers. Therefore, also HTML services that return different data for the same URL or HTML services that keep complicated state information are by default handled correctly in this method.

3.1 Different deployment options

The two fundamental co-browsing methods explained in the previous sections can be deployed in multiple ways. We list here the most realistic ones. Note that for Figure 6 to Figure 9, a circular arrow in a browser indicates that in this browser the JavaScript code of the co-browsed web application is executed, otherwise it is not.

In the following figures, the *HTML service* denotes the backend components which are serving the web-application which is co-browsed. *Browser 1* and *Browser 2* denotes the clients doing co-browsing including all necessary co-browsing functionality and deployment specific functionality which is described in the following. All other components are deployment specific and are described for each option in detail.

1. *JavaScript engine input synchronization:*

   All locally generated UI events are intercepted by the co-browsing code and the default actions and associated event handlers are not executed. Instead, they are sent over to a *UI event ordering service*. This service does global ordering of the events so that all events are received at all ends in exactly the same order (so-called globally ordered multicast). Without a UI event ordering service only a master-slave co-browsing solution would be possible. All end points receive the ordered events and inject them into the event queue of the JavaScript engine executing the web-application. Without a UI event ordering service only a master-slave co-browsing solution would be possible. A co-browse proxy is needed to synchronize the content coming from the HTML service.

   For the co-browse proxy functionality, two deployment options can be distinguished:

   a. *Using a backend co-browse proxy server:*

   A *backend co-browse proxy* acts on behalf of the clients and requests all data (including XMLHttpRequests) from the web service. It caches the data and ensures all browsers receive the same data. This solution is shown in Figure 6.

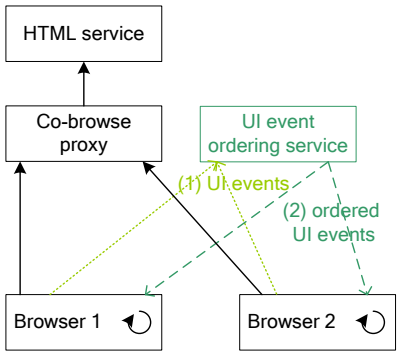

Figure 6 UI event synchronization combined with a backend proxy

   b. *Using a client-side co-browse proxy:*

   The co-browse proxy functionality is implemented on one of the co-browsers. Every time a new web page is loaded, the proxy code will send the HTML of this new page to the other co-browsers, using an *HTTP forwarding service.* Within the other co-browsers, the co-browsing code will receive this HTML and use it to overwrite the document of the co-browsed frame. Also all data received via an XMLHttpRequest object must be synchronized (see Section 4). Note that with this method it is not possible to proxy binary data like pictures and movies. Figure 7 gives an overview of this solution.

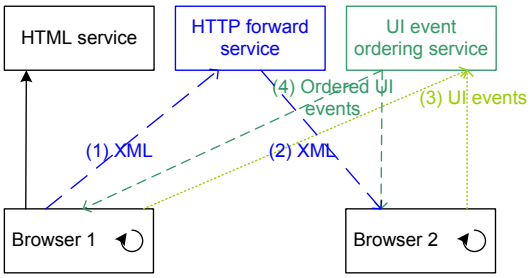

Figure 7 UI event synchronization combined with a client-side proxy

2. *JavaScript engine output synchronization:*

   All locally generated UI events are intercepted by the co-browsing code and the associated event handlers and default actions are not executed. Instead, they are sent to a reference browser, which injects the events into the event queue of the JavaScript engine executing the web-application. Only the reference browser runs the JavaScript code causing changes in the DOM tree. A description of these DOM tree changes are forwarded to all other co-browsers which apply them to their local DOM tree.

[13] It is also possible that only the user using the reference browser has control. Such a situation is called Master-Slave.

For the reference browser, two deployment options can be distinguished:

a. *Backend reference browser:*

The backend *reference browser* does not have any local user interaction. Instead, the user interaction of every co-browsing user is sent to the reference browser in form of UI event descriptions. These events are injected into the JavaScript engine executing the JavaScript of the web application. If the JavaScript causes changes to the application's DOM tree, updates of these changes are sent to the client browsers and applied to their DOM tree. The client browsers do not execute the web-application's JavaScript code. It is only executed on the back-end. See Figure 8 for an overview.

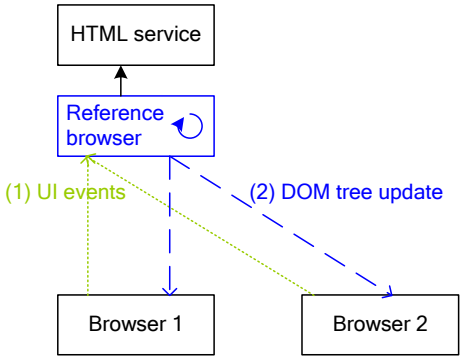

Figure 8 A backend reference browser

b. *Client-side reference browser:*

In the solution depicted in Figure 9, one of the co-browsers plays the role of the reference browser. All other co-browsers send the UI events to this reference browser using an *event forwarding service*. As opposed to the UI event synchronization, this service only enables peer-to-peer communication between the browsers and does not need to do any event ordering. The received UI events are injected into the reference browser, and all DOM tree updates caused by local and remote UI events are forwarded to the client browsers using a *DOM forward service*. Note that only the reference browser executes the web-application's JavaScript code. Compared to the backend reference browser, this solution is far more scalable.

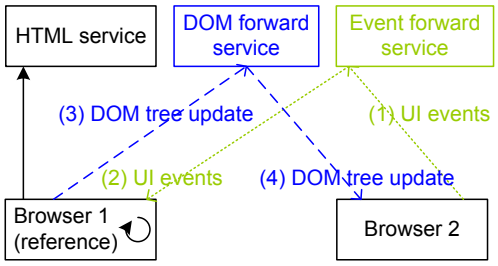

Figure 9 A reference browser on one of the client devices

3.2 Comparison

From the above description of the two methods for co-browsing, the following advantages and disadvantages of the methods can be deducted.

1. *Robustness*:

   JavaScript engine output synchronization is robust because resynchronization is easy to implement by just sending over the complete DOM tree. JavaScript engine input synchronization is less robust, because it may be difficult to keep the JavaScript engines in sync. Resynchronization is difficult to implement, because this would mean sending over the DOM tree and the state of the JavaScript engine.

2. *User experience:*

   JavaScript engine input synchronization has a better UI experience, especially for web page containing animations and video content. JavaScript engine input synchronization allows the users to quit a co-browse session at any point in time and continue browsing from the exact place they stopped the co-browsing session (thus a seamless "break-out" of the user is possible)[14]. The output synchronization cannot support that easily; the state of the JavaScript engine has to be transferred to all client browsers.

3. *Scalability:*

   JavaScript engine output synchronization generates more network traffic than input synchronization, since the co-browsers have to send both DOM tree updates and UI events to the reference browser. In case of input synchronization, only the UI events have to be sent. The actual amount of traffic depends on the deployment variants. From the perspective of multiple independent co-browsing sessions, the major disadvantage in both cases with infrastructure support is that a centralized solution is less scalable than the client site implementation. From the perspective of one co-browsing session, the local co-browse proxy limits the amount of users drastically, because the upload bandwidth needed for data forwarding is limited.

4. *Implementation effort:*

   DOM tree synchronization is a much simpler and robust approach and thus the implementation effort is lower than in the UI event synchronization.

For all variants, it holds that the concurrent interaction of multiple users with a web-application can cause interpretation problems of the user interaction. In case of JavaScript engine input synchronization, the co-browse functionality only ensures that the UI events are executed in exactly the same order on all ends. But this problem is not specific for co-browsing. Also multi-user interaction on the same device can cause problems. The only generic solution is to disallow concurrent interaction and use master-slave kind of co-browsing.

4. JAVASCRIPT ENGINE INPUT SYNCHRONIZATION

We consider the co-browsing based JavaScript engine input synchronization the most interesting because it offers the best user experience and the best scalability. In this section, we will describe this method (1.a in the previous section) in more detail and we will describe how this method can be implemented in JavaScript. We make use of the fact that JavaScript is an event-based language and is single-threaded. The co-browsing solution consists of three parts:

[14] Starting of co-browsing at any point in time is not supported by the proposed system. This is an aspect of future work.

the co-browsing code on the client side, the co-browse proxy and the UI event ordering service. The UI event ordering service ensures that the events sent by the co-browsing code of all browsers are received in exactly the same order at all ends (globally ordered multicast). This is achieved by merely following a first-in-first-out policy. Moreover, it also provides a communication channel for the case that peer-to-peer connections between the browsers are not possible due to firewalls or Network Address Translation (NAT).We assume that there is a proxy server that takes care that the two co-browsers receive the same HTTP packets. In this variant, receiving different data for the same URL from a server or a server that keeps state information is not an issue. First, we describe the synchronization of UI events. Then we describe the synchronization of other JavaScript engine inputs. Finally we describe how the client side JavaScript code can be deployed on the browsers.

4.1 UI event synchronization

The basic challenge is to keep the JavaScript engines in all browsers synchronized by triggering at both sides the same UI events in the exact same order. In all browsers, the co-browsing code should perform the following basic steps when a new web page is co-browsed:

1. After a new page has been loaded, take care that every HTML element has a unique identifier.
2. Intercept all locally generated UI events, disable them and send a description of them to the UI event ordering service.
3. Start listening to event descriptions coming from the event ordering service.
4. For every event description received from the event ordering service perform the necessary actions.

We will now describe these steps in more detail. For all co-browsers, the first step is to make sure that every HTML node in the HTML document has a unique ID. Therefore, a script is used that assigns an identifier to every node without an identifier. It is executed after the new page has been loaded which is triggered by the "`load`" event. All default user actions are blocked until the `load` event had occurred since user interactions that take place before this event cannot (always) be transmitted.

All locally generated UI events must be intercepted, captured and sent to the event ordering service. These are the DOM level 2 core events [11] (*DOMFocusIn*, *DOMFocusOut*, *DOMActivate*, *mousedown*, *mouseup*, *click*, *mouseover*, *mousemove*, and *mouseout)*, the key events (*keydown*, *keyup*, and *keypress*) and the HTML events (*submit*, *focus*, *blur*, *resize,* and *scroll*). Capturing the UI events can be done by registering event handlers by means of the *addEventListener()* method provided by the DOM interface. Since the UI events must first be intercepted and sent to the UI event ordering service, the local handling of the user input must be disabled. This means that the default action of the browser should be blocked (e.g. loading of a new page when an anchor has been clicked) and that no JavaScript event handlers of the original web pages may be called. The former can be achieved by the use of the DOM function *preventDefault()*. To prevent JavaScript event handlers from processing an event, it is possible to use the *stopPropagation()* function. The code snippet in Figure 10 shows how all local event handling can be disabled for the "click" event.

```
document.addEventListener(
        'click',
        function(event){
                event.stopPropagation();
                event.preventDefault();
        },
       true
);
```

Figure 10 Pseudo-code for intercepting and blocking of local mouse click events

Sending over an event description to the UI event ordering service can be done by first serializing the event into an XML string and then sending it using the *XMLHttpRequest* object.

```
<event>
  <type>mouseover</type>
  <url>http://www.flickr.com</url>
  <target>cssf_351</target>
  <bubbles>true</bubbles>
  <cancelable>true</cancelable>
  <timeStamp>0</timestamp>
  <screenX>854</screenX>
  <screenY>494</screenY>
  <clientX>636</clientX>
  <clientY>354</clientY>
  <pageX>636</pageX>
  <pageY>2695</pageY>
  <ctrlKey>false</ctrlKey>
  <shiftKey>false</shiftKey>
  <altKey>false</altKey>
  <metaKey>false</metaKey>
  <button>0</button>
  <relatedTarget>null</relatedTarget>
</event>
```

Figure 11 UI event description in XML

Figure 11 gives an example of the syntax of an event description that is sent over the event ordering service.

All clients receive an ordered stream of event descriptions from the UI event ordering service. Receiving these events can again be done through the use of the *XMLHttpRequest* object as described e.g. in [10], or in case of Web4CE or Firefox by using a TCP connection. For every event description received, the corresponding event can be recreated and dispatched by using the functions *document.createEvent()*, *document.initEvent()* and *document.dispatchEvent()*, as exemplified by the following pseudo code.

```
document.createEvent(eventtype);
document.initEvent(event);
target.dispatchEvent(event);
```

Figure 12 Pseudo-code for generating JavaScript UI events

These functions cause the associated JavaScript event handlers to be called. This keeps the user input for the JavaScript engines synchronized. However, recreating and dispatching an "artificial" DOM UI event does not always cause the associated default action to be executed. For example, manually firing a "focus event" on a document element does not cause the element to receive focus. The *focus()* method must be used for that. Likewise, manually firing a "submit event" does not submit a form. The *submit()* method must be used for that. This is important for security reasons, as it prevents scripts from simulating user actions that interact with the browser itself. The only actions that happen after calling *target.dispatchEvent()* for a specific event is that the associated JavaScript event handlers are executed.

4.2 Synchronizing other JavaScript engine interactions

In the previous section, we have described the solution for co-browsing based on UI event synchronization and how the user input

for the JavaScript engines can be kept synchronized even though different UI events are generated at both sides simultaneously. However, to keep the JavaScript engines synchronized not only the external UI events have to be synchronized, but also every interaction of the JavaScript code from the co-browsed web page with the outside has to be synchronized. The JavaScript engine runs in a sandbox and can only interact with elements outside the sandbox through the BOM functions and the BOM properties provided by the browser. Some of these BOM functions and properties can cause loss of synchronization, for example: *window.random()*, *window.setTimeout()* and *window.setTimeInterval(), window.alert()*, *window.confirm()*, *window.show()*, *window.open()* etc. Also requests via the *XMLHttpRequest* object must be synchronized. An overview is given in the following Figure 13.

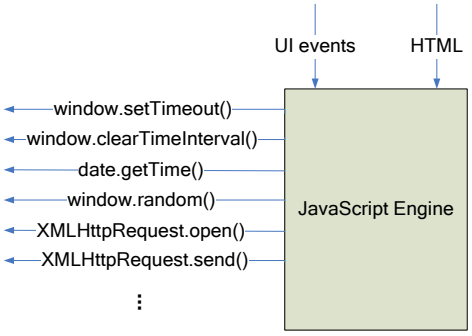

Figure 13 Interactions of the JavaScript engine via the BOM

The actions and return values of these functions need to be synchronized. This can be done by redefining these functions, before they are called. We will show how this can be done by taking the *window.setTimeout()* function as an example. The effects of the other functions can be synchronized in a similar way. The *window.setTimeout()* function will register a callback function that will be called by the JavaScript engine after the specified amount of time. If this callback function is not called between exactly the same event handlers on the different co-browsers, the synchronization can be broken. The exact timing of the callback execution is not the issue. It is important that within all browsers the timeout callbacks are inserted at the same place in the event queue. The basic idea of the solution is to redefine the *window.setTimeout()* function. If the redefined *window.setTimeout()* is called, the callback function is stored and a sequence number is assigned. Only on one browser (the *master*) a timeout is registered, which sends a notification with the sequence number to the co-browsers using the UI event ordering service when executed. All co-browsers execute the timeout's callback when receiving this event. The pseudo-code in Figure 14 shows how the *setTimeout()* function of the master browser can be redefined.

```
old_setTimeout = window.setTimeout;
new_setTimeout = function(callback,time){
    new_callback = new function(){
         send_to_ui_event_ordering_service
                  (callback);
    }
    old_setTimeout(new_callback, time);
}
window.setTimeout = new_setTimeout;
```

Figure 14 Pseude-code for synchronising the window.setTimeout() function

Other issues can be solved as follows:

File upload: File upload cannot be handled by JavaScript code on the client side alone. There are two options. Either file upload is disabled on both sides or file upload is only enabled on one client and the co-browse proxy is needed to make sure both clients receive the same response from the server after the file is uploaded.

Cookies: Web pages have the possibility to save state at the client side by means of cookies. The co-browse proxy has to decide of which client the cookies are used. Another approach is that JavaScript code is used to synchronize the cookies at both sides.

Mouse pointer: Another issue is how to deal with the mouse pointer during a co-browsing session. There are basically two options: only one pointer is used, which is shared by all clients, or every client has his or her own visible pointer, such that multiple pointers are visible on the screen. Both options are equally valid and depend on the user preferences. In our implementation we have opted for the second option.

4.3 Co-browsing client code deployment

As can be seen from the previous section, the code needed for keeping the JavaScript engine synchronized can be implemented in JavaScript[15]. We describe now how this co-browsing JavaScript code can be added to the browser:

1. *Insertion by co-browse proxy:*

   The co-browse proxy needed in the UI event synchronization is able to add additional code into the downloaded web-pages. In this case, the web-applications will get "build-in" support for co-browsing. Since the proxy is not always necessary the co-browsing solution gets less scalable because the browser always has to download at least the initial page through the co-browse proxy.

2. *JavaScript browser add-on:*

   Many browsers support the development of browser add-ons developed in JavaScript. A variant of this is that the co-browsing JavaScript code can be in the form of a user script such as Greasemonkey[16] for Firefox or Greasekit[17] for Safari.

3. *Co-browsing JavaScript code in a (hidden) iframe:*

   In this option, the user has to load a portal site first. The user than can use this portal site to browse to single-user web applications and start a co-browsing session with others. Here, the code for co-browsing (and also for session set-up) resides in a (hidden) frame. This frame must be permanently available during the co-browsing session. The JavaScript in this frame should also have special privileges (cross-domain scripting) to be able to listen to events happening in the co-browsed frames and to makes changes in these frames. This option is depicted in Figure 15. The top frame of the browser consists of two sub-frames. One frame is the *community frame* which serves for communication purposes (buddy list, IM, video chat) and it can be hidden when not needed. The other frame is the *application frame*, which is shared, and where the web service being co-

[15] Plug-ins or native implementations in the browser are of course always possible if needed to improve performance on embedded devices.

[16] http://www.greasespot.net/

[17] http://pimpmysafari.com/plugins/greasekit-10

browsed is shown. The community frame also contains the necessary JavaScript code for co-browsing.

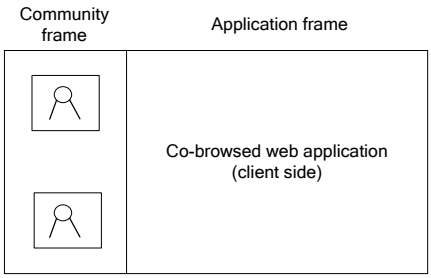

Figure 15 A possible UI layout for a co-browsing application.

5. IMPLEMENTATION

We have implemented the co-browsing method as described in the previous section in a proof-of-concept demonstrator. The client side of the demonstrator is based on the Firefox browser.

Figure 16 Two people remotely watching vacation pictures together on Google Maps.

In a first step, we had implemented the co-browsing as a Greasemonkey script. Though this worked well, we wanted to extend this co-browsing application with presence, instant messaging and video chat functionality. To this end, we moved the co-browsing JavaScript code into a separate iframe, which we refer to as the community frame as explained in the previous sections. We also implemented an XMPP client in JavaScript and added this to the community frame. The XMPP client provides presence, instant messaging and session initialization for video chat or co-browsing. We added video chat functionality by means of the Adobe Flash Player[18] and the Adobe Flash Media Server 2[19]. Figure 16 shows a screen shot from this first prototype version where the community frame is still clearly visible on the left.

In later versions of the demonstrator (Figure 17), the community frame is hidden, but it overlays its content transparently over the application frame when needed, for example to show the buddy list, show the video chat and to show incoming invitations.

Moving the JavaScript code for co-browsing to the community frame means that it needs extra security privileges (cross-domain scripting) to be able to listen to events happening in the co-browsed application frame and to make changes to it. For the event ordering service, we use the XMPP chat room mechanism which already provides the necessary ordering functionality. The co-browse proxy is implemented as a backend service in Java.

[18] http://www.macromedia.com/software/flash/about/

[19] http://www.adobe.com/products/flashmediaserver/

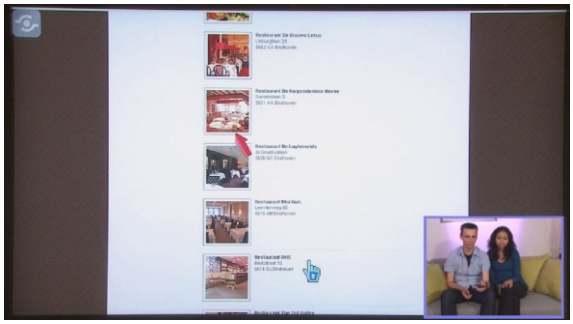

Figure 17 Bob and Alice looking a nice restaurant together with their (remote) friends.

To communicate with the XMPP server, we use the TCP socket mechanism provided by the Firefox browser available for trusted or signed JavaScript code. However, there also exist XMLHttpRequest-based implementations in JavaScript for XMPP [10]. For CE devices, the Web4CE standard supports outgoing TCP connections via the Web4CE specific Notifsocket mechanism. Figure 18 gives an overview of the high-level architecture of the demonstrator.

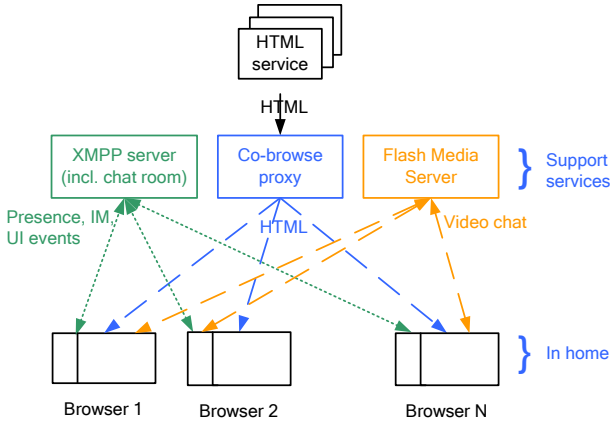

Figure 18 Overview of the demonstrator architecture.

The Firefox browser is used without making any changes, except for configuring it to allow trusted or signed JavaScript to receive cross-domain scripting permissions and to make use of the Firefox socket mechanism. We also configured the Firefox browser to not allow window pop-ups or open new windows which would break the co-browsing synchronization.

Though the co-browsing solution was not developed with SVG in mind, first tests with SVG documents are promising and indicate that also SVG documents can be co-browsed with the current solution

Our current demonstrator shows that it is possible to do co-browsing of highly scripted web pages like e.g. Google Maps[20], Flickr[21], BrowseGoods[22] without loss of synchronization. However, for many pages the co-browsing solution can still loose synchronization. This is due to a number of issues with real-life web pages:

[20] http://maps.google.com

[21] http://www.flickr.com

[22] http://browsegoods.com/

- Flash is used a lot, but it is not possible to synchronize flash on the DOM level of the browser. In the demonstrator, a Flash object is simply removed from the page.
- CSS style rules are not triggered by JavaScript generated UI events. Style rules do not have lasting effect and most often don't cause loss of synchronization. A synchronization loss can happen when the application of a style rule changes the size of an HTML element and hence causes a change in the layout of the web page. A possible solution is to listen with DOM Mutation event listeners to changes in the size of HTML elements.

Many other issues can be solved by implementing customized solutions in the co-browsing methods for common web page errors and browsers peculiarities. This is the approach we have taken in our demonstrator.

However, web pages that want to support co-browsing can adhere to a few simple guidelines that make the co-browsing solution significantly simpler to implement and more robust. Examples are to use always a unique URL for a unique piece of data, to only start executing JavaScript code after the *onload* event and to use SVG instead of Flash.

There are also a number of browser features already supported by some browsers, which can make the co-browsing solution simpler and/or more robust. Here are a few important examples:

- *Support for the beforeExternalScript event:*

  This event is currently supported only by the Opera browser and is raised before any JavaScript code of a newly loaded web page is executed. It provides a convenient entry point to make the necessary redefinitions of various JavaScript methods.

- *Support for getElementFromPoint():*

  This function is currently only supported by Firefox and Internet Explorer and returns the HTML element located at the specified client screen coordinates. This removes the need to assign an identifier for every HTML element

- *Support for SVG:*

  SVG can then be an alternative for Flash.

- *Triggering of CSS rules by events:*

  CSS rules should also be triggered by an event which is generated through JavaScript by the *event.dispatch()* function.

The demonstrator implementation introduced in this section is under ongoing development and is extended continuously. Currently we are working on video object synchronization. This will provide synchronized play-out of video objects like Flash video or HTML 5 video. This enables the users to co-watch web-based video on demand. Another ongoing work is the extension of our sharing framework to support more than two users in a co-browsing session. We are also working on an API that allows third party developers to use the communication and synchronization platform to develop multi-user web applications.

6. RELATED WORK

There exist already a number of solutions that provide DOM-level synchronization. All solutions provide the basic web co-navigation functionality that ensures that all the browsers of a session display the same URL. Each co-browsing solution enhances this basic functionality in one or more ways: synchronous scrolling, a shared pointer, chat functionality, co-annotating web pages in real-time, etc. However, current solutions only provide mechanisms for synchronizing the static part of web pages and in some cases support for additional features such as synchronizing forms or synchronized scrolling. None of these co-browsing solutions provide support to synchronize the JavaScript interactions of the web pages. A second difference is that the client side of existing co-browsing solutions is often implemented as Java applets whereas in our solution, the client side can be implemented in JavaScript. This is for example the case in [5], [6] and [7].

The GroupWeb [6] co-browsing solution supports synchronous scrolling, telepointers for enacting gestures, and group annotations that can be attached to pages. GroupWeb is based on a backend proxy server and client-side Java-applet technology.

In [3] a co-browsing solution is described in which each user's computer has two instances of the web browser running. One instance contains the target web page that is being collaboratively viewed. The other window contains a control panel and a monitor routine. The monitor routine periodically analyzes the browser instance containing the target web page, to see if any changes have occurred. If a change is detected, such as if the user has scrolled a scrollbar, then that change is transmitted to the other browser. This solution synchronizes attributes such as the browser window size, window and frame web page sources, and scroll bar positions. It also synchronizes form element content (e.g. text fields, checkboxes, select lists).

The co-browsing solution presented in [5] is based on a backend co-browse proxy server and Java applet technology at the client side. This work focuses on symmetric co-browsing where each participating user can take the lead and guide others while browsing web pages. This work proposes the use of a token to determine which browser is the master browser at any moment in time; only the user actions of the browser that has the token have effect. In our solution, all users are at any moment in control. The UI event ordering service ensures that all browsers process incoming UI events and data in exactly the same order. In this way, we keep the browsers always synchronized.

Colab [7] is a co-browsing solution that is also based on a backend co-browse proxy server and requires support for Java applets by the browser. The main focus of this work is on easily creating and releasing synchronization relations among users.

Screen sharing solutions, which are based on synchronizing the frame buffer, are also used for co-browsing. A screen sharing solution has the main advantage that it is conceptually very simple and the risk of loss of synchronization is very low. However, screen sharing solutions require a lot of bandwidth when there are frequent changes on the screen through animations, slide shows or movie clips for example. Due to the asymmetrical nature of most internet connections currently available, the upstream bandwidth is the limiting factor in shared applications systems. A simple calculation[23] shows that for the JavaScript Engine input synchronization method and implementation presented in this paper the upload bandwidth is realistically at most 120 Kbit/s. Initial measurements on our demonstrator confirm this estimation. Peaks in bandwidth happen when many UI events occur. This occurs, for example, when the

[23] One UI event description as described in Section 4 can easily be reduced to a maximum of 150 bytes. During browsing, a maximum of 100 UI events per second are realistically possible. This leads to a maximum upstream bandwidth of 15 Kbyte/s or 120 Kbit/s at one client.

mouse is moved (generating a lot of *"mousemove"* events), when a key is pressed continuously or during scrolling. To compare this with screen sharing solutions, we have tested two screen-sharing products: Microsoft NetMeeting[24] and Citrix GoToMeeting[25] on a 100 Mbit LAN. Given enough bandwidth and for static web pages, both NetMeeting and GoToMeeting performed relatively well. However, when web pages contain frequent screen updates, both systems needed high bandwidth and still had a low frame rate. GoToMeeting used a maximum of around 4 Mbit/s in case of frequent changes on the screen and Microsoft NetMeeting around 20Mbit/s. Even when using this high bandwidth, NetMeeting achieved a very low frame rate (at most 1 frame per second).

7. CONCLUSIONS

This paper introduces a co-browsing solution that allows the co-browsing of dynamic, JavaScripted web pages. The solution works at the DOM level, can be implemented in JavaScript and requires no extensions to the browser. Compared with screen sharing solutions, the needed bandwidth for synchronization is relatively low and it offers a better user experience.

While the presented solution is aimed at Web4CE, it is also applicable to dynamic (X)HTML web pages on PCs and mobile devices. Also for professional collaboration in the PC world, where there is currently a trend to also provide collaborative applications via a native browser, we believe that our co-browsing solution can play a valuable role.

The co-browsing solution that we propose is generic in that it does not pose requirements to web pages. However, if web developers adhere to a few basic guidelines to support co-browsing, the solution can be made much simpler and also a co-browsing proxy can be left out. Similarly, the support of a few features by the browser can simplify the implementation and can make it more robust against synchronization losses. Moreover, the solution could also be directly integrated into standard browsers which also would improve the performance on embedded platforms (e.g. mobiles and TVs).

The presented co-browsing solution enables users to start a co-browsing session from a given URL. This does not allow a user to start a co-browsing session at any point in time, since the JavaScript application may have changed state after loading the URL e.g. due to user interaction. We refer to this type of immediate sharing as catch-up co-browsing and we are currently investigating possible mechanisms to provide such functionality. Other aspects of further work include the co-watching of web-based multimedia content and an extended session management which includes session with multiple users, content items and services.

With our proof-of-concept demonstrator we've shown the technical feasibility of our solution. The demonstrator provides an "Experience sharing" framework which combines the co-browsing of arbitrary (multimedia) web application with communication (audio and video chat) and targets the usage on a TV from the comfort of the living room.

[24] http://www.microsoft.com/windows/NetMeeting/default.ASP

[25] https://www.gotomeeting.com/

8. REFERENCES

[1] Consumer Electronics Association (CEA), *CEA-2014 Web-based Protocol and Framework for Remote User Interfaces on UPnP™ Networks and the Internet (Web4CE)*, June 2006, http://www.ce.org/standards/StandardDetails.aspx?Id=2865&number=CEA-2014

[2] Dees, W., Shrubsole, P.; Web4CE: Accessing web-based applications on consumer devices; 2007; http://www2007.org/program/poster.php?id=1017

[3] Esenther, A.; Instant Co-Browsing: Lightweight Real-Time Collaborative Web Browsing; In Proc. of the 11th Int. WWW Conference; 2002.

[4] Farnham, S., Zaner, M., Cheng, L.,; Supporting Sociability in a Shared Browser; Virtual Worlds Group, Microsoft Research;

[5] Gerosa, L., Giordani, A., Ronchetti.; Symmetric Synchronous Collaborative Navigation; Proceedings of the 2004 IADIS International WWW/Internet Conference, Madrid, Spain; 2004.

[6] Greenberg, S., Roseman, M.; GroupWeb: a www browser as real-time groupware; ACM; 1996.

[7] Hoyos-Rivera, G. , Lima-Gomes, R., Courtiat, J.; CoLab: A Flexible Collaborative Web Browsing Tool; Proceedings of the 19th International Conference on Advanced Information Networking and Applications; 2005.

[8] Lowet D., Shrubsole P.; Content sharing and experience sharing with Web4CE; TICSP Adjunct Proceedings of EuroITV 2007.

[9] OpenIPTV forum; Functional Architecture – V 1.1,; 2008; http://www.openiptvforum.org/docs/OpenIPTV-Functional_Architecture-V1_1-2008-01-15_APPROVED.pdf

[10] Paterson, I., Smith, D., Saint-Andre, P.; XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH); Version 1.7; 2008; http://www.jabber.org/jeps/jep-0124.html

[11] Pixley, T.; Document Object Model (DOM) Level 2 Events specification; Version 1.0; 2000; http://www.w3.org/TR/DOM-Level-2-Events/

[12] Stenback, J., Le Hegaret, P., Le Hors, A.; Document Object Model (DOM) Level 2 HTML Specification; W3C Recommendation; 2003.

[13] Wang, F., Rabsch, C., Liu, P.. Native Web Browser Enabled SVG-based Collaborative Multimedia Annotation for Medical Image. ICDE 2008: 1219-1228

[14] Waterson, C.. *Gecko overview;* June 2002; www.mozilla.org/newlayout/doc/gecko-overview.ppt