RCB: A Simple and Practical Framework for Real-time Collaborative Browsing

Chuan Yue, Zi Chu, and Haining Wang
The College of William and Mary
{*cyue,zichu,hnw*}*@cs.wm.edu*

Abstract

Existing co-browsing solutions must use either a specific collaborative platform, a modified Web server, or a dedicated proxy to coordinate the browsing activities between Web users. In addition, these solutions usually require co-browsing participants to install special software on their computers. These requirements heavily impede the wide use of collaborative browsing over the Internet. In this paper, we propose a simple and practical framework for Real-time Collaborative Browsing (RCB). This RCB framework is a pure browser-based solution. It leverages the power of Ajax (Asynchronous JavaScript and XML) techniques and the end-user extensibility of modern Web browsers to support co-browsing. RCB enables real-time collaboration among Web users without the involvement of any third-party platforms, servers, or proxies. It allows users to perform fine-grained high quality co-browsing on arbitrary websites and webpages. We implemented the RCB framework in the Firefox Web browser and evaluated its performance and usability. Our evaluation results demonstrate that the proposed RCB is simple, practical, helpful and easy to use.

1 Introduction

Many end-user real-time applications have been widely used on the Internet. Real-time audio/video communication is enabled by voice/video over IP systems, real-time text-based communication is enabled by instant messaging systems, and real-time document sharing and collaboration is enabled by Web-based services such as Google Docs and Adobe Buzzword. However, one of the most popular Internet activities, Web browsing, is still heavily isolated. In other words, browsing regular webpages is still a process that is mainly between a user client and a remote Web server, and there is little real-time interaction between different users who are visiting the same webpages.

Collaborative browsing, also known as co-browsing, is the technique that allows different users to browse the same webpages in a simultaneous manner and collaboratively fulfill certain tasks. Co-browsing has a wide range of important applications. For example, instructors can illustrate online materials to distance learning students, business representatives can provide live online technical support to customers, and regular Web users can conduct online searching or shopping with friends.

Several approaches exist to achieve different levels of co-browsing. At one extreme, simple co-browsing can be performed by just sharing a URL in a browser's address bar via either instant messaging tools or Web browser add-ons (such as CoBrowse [5]) that are installed on each user's computer. URL sharing is lightweight, but it only enables very limited collaboration on a narrow scope of webpages. Collaboration is limited since users can only view webpages but cannot perform activities such as co-filling online forms or synchronizing mouse-pointer actions. Webpages eligible for this simple co-browsing method are also limited because: (1) most session-protected webpages cannot be accessed by just copying the URLs, and (2) in many dynamically-updated webpages such as Google Maps, the retrieved contents will be different even with the same URL.

At the other extreme, complex co-browsing can be achieved via screen or application sharing software such as Microsoft NetMeeting. To enable co-browsing activities, these solutions must grant the control of a whole screen or application to remote users. As a result, they place high demands on both security assurance and network bandwidth, and their use is more appropriate for some other collaborative applications than co-browsing.

A number of solutions have been proposed to support full functional co-browsing with moderate overhead. Based on the high-level architectures, these solutions can be classified into three categories: platform-based, server-based, and proxy-based solutions. Platform-based solutions build their co-browsing functionalities upon

specific real-time collaborative platforms [9, 11, 15, 30]. Server-based solutions modify Web servers to meet collaborative browsing requirements [2, 7, 13, 22, 28]. Proxy-based solutions use external proxies, which are deployed between Web servers and browsers, to facilitate collaborative browsing [1, 3, 4, 6, 12, 14]. However, as discussed in Section 2, the specific architectural requirements of these solutions limit their wide use in practice.

In this paper, we propose a simple and practical framework for Real-time Collaborative Browsing (RCB). The proposed RCB is a pure browser-based solution. It leverages the power of Ajax (Asynchronous JavaScript and XML) [20] techniques and the end-user extensibility of modern Web browsers to support co-browsing. RCB enables real-time collaboration among Web users without using any third-party platforms, servers, or proxies. The framework of RCB consists of two key components: one is RCB-Agent, which is a Web browser extension, and the other is Ajax-Snippet, which is a small piece of Ajax code that can be embedded within an HTML page and downloaded to a user's regular browser. Installed on a user's Web browser, RCB-Agent accepts TCP connections from other users' browsers and processes both Ajax requests made by Ajax-Snippet and regular HTTP requests. RCB-Agent and Ajax-Snippet coordinate the co-browsing sessions and allow users to efficiently view and operate on the same webpages in a simultaneous manner.

The framework of RCB is simple, practical, and efficient. A user who wants to host a collaborative Web session only needs to install an RCB-Agent browser extension. Users who want to join a collaborative session just use their regular JavaScript-enabled Web browsers, and nothing needs to be installed or configured. End-user extensibility is an important feature supported by popular Web browsers such as Firefox [24] and Internet Explorer [26]. Thus, it is feasible to implement and run the RCB-Agent extension on these browsers. Meanwhile, currently 95% of Web users turn on JavaScript in their browsers [21], and all popular Web browsers support Ajax techniques [20]. As a result, joining a collaborative Web session is like using a regular browser to visit a regular website. The simplicity and practicability of RCB bring important usability advantages to co-browsing participants, especially in online training and customer support applications. RCB is also efficient because co-browsing participants are directly connected to the user who hosts the session, and there is no third-party involvement in the co-browsing activities.

Other distinctive features of RCB are summarized as follows. (1) *Ubiquitous co-browsing:* since no specific platform, server, or proxy is needed, co-browsing can be performed in many different places via any type of network connection such as Ethernet, Wi-Fi, and Bluetooth. (2) *Arbitrary co-browsing:* co-browsing can be applied to almost all kinds of Web servers and webpages. Web contents hosted on HTTP or HTTPS Web servers can all be synchronized to co-browsing participants by RCB-Agent. Our RCB-Agent can also send cached contents including image and Cascading Style Sheets (CSS) files to participants, hence improving performance and accessibility of co-browsing in some environments. (3) *Fine-grained co-browsing:* co-browsed Web elements and coordinated user actions can be very fine-grained. Since RCB-Agent is designed as a browser extension, the seamless browser-integration enables RCB-Agent to fully control what webpage contents can be shared and what actions should be allowed to participants, leading to full functional high quality co-browsing.

We implemented the RCB framework in Firefox. As a browser extension, RCB-Agent is purely written in JavaScript. Ajax-Snippet is also written in JavaScript and it works on different browsers like Firefox and Internet Explorer. We evaluated the real-time performance of RCB through extensive experiments in LAN and WAN environments. Based on two real application scenarios (collaboratively shopping online and using Google Maps), we also conducted a formal usability study to evaluate the high quality co-browsing capabilities of RCB. Our evaluation results demonstrate that the proposed RCB is simple, practical, helpful and easy to use.

2 Related Work

The existing co-browsing solutions can be roughly classified into platform-based, server-based, and proxy-based solutions. Platform-based solutions build their co-browsing architectures upon special real-time collaborative platforms. As an early work in this category, GroupWeb [11] is built on top of the GroupKit groupware platform [18], and similarly GroupScape [9] is developed by using the Clock groupware development toolkit [10]. Two banking applications [15] for synchronous browser sharing between bank representatives and customers are designed on top of a multi-party, real-time collaborative platform named CollaborationFramework [19]. Recently, SamePlace [30] is built upon the XMPP (eXtensible Messaging & Presence Protocol) platform [32] to support co-browsing of rich Web contents. The strong dependence on specific collaborative platforms is the major drawback of these co-browsing solutions.

Server-based solutions modify Web servers and integrate collaborative components into servers to support co-browsing [2, 7, 13]. CWB (Collaborative Web Browsing) [7] is a typical example in this category. CWB consists of a controller module that runs on a Web server and a control panel that runs on a Web browser. The controller module is implemented as a Java servlet and is the central control point for collaborative activities. The con-

trol panel reports local browser instance changes to the controller module on the Web server, and it also polls the controller module for changes made by other users. In addition to CWB, some commercial software like Backbase Co-browse & Chat suite [22] and PageShare [28] also adopt this approach. However, these solutions have two obvious limitations: (1) they require Web developers to add controller modules to Web servers, and (2) the server-side modification is usually tailored and dedicated to individual websites, and it is infeasible to apply such a modification to most Web servers.

Proxy-based solutions rely on dedicated HTTP proxies to coordinate co-browsing among users [1, 3, 4, 6, 12, 14]. Users configure the proxy setting on their browsers to access the Internet via an HTTP proxy. The proxy serves co-browsing users by forwarding their HTTP requests to a Web server and returning identical HTML pages to them. The proxy also inserts applets (often in the form of Java applets [4, 12] or JavaScript snippets [3]) into the returned HTML pages to track and synchronize user actions. The major drawback of proxy-based solutions is the extra cost of setting up and maintaining such a proxy. Moreover, there are security and privacy concerns on using a proxy. Since all the HTTP requests and responses have to go through a proxy, each user has no choice but to trust the proxy.

3 Framework Design

In this section, we first present the architecture of the RCB framework. We then justify our design decisions. Finally, we analyze the co-browsing topologies and policies of RCB, and discuss the security design of RCB.

3.1 Architecture

The design philosophy of RCB is to make co-browsing simple, practical, and efficient. As shown in Figure 1, the architecture of the RCB framework consists of two major components. One is the RCB-Agent browser extension that can be seamlessly integrated into a Web browser. The other is Ajax-Snippet — a small piece of Ajax [20] code that can be embedded within an HTML page and downloaded to a user's regular browser. For a user who wants to host a co-browsing session, the user (referred to as a *co-browsing host*) only needs to install an RCB-Agent browser extension. For a user who wants to join a co-browsing session, the user (referred to as a *co-browsing participant*) does not need to install anything and just uses a regular JavaScript-enabled Web browser. Our design philosophy of making the participant side as simple as possible is similar to the basic concept of many thin-client systems such as VNC (virtual network computing) [17].

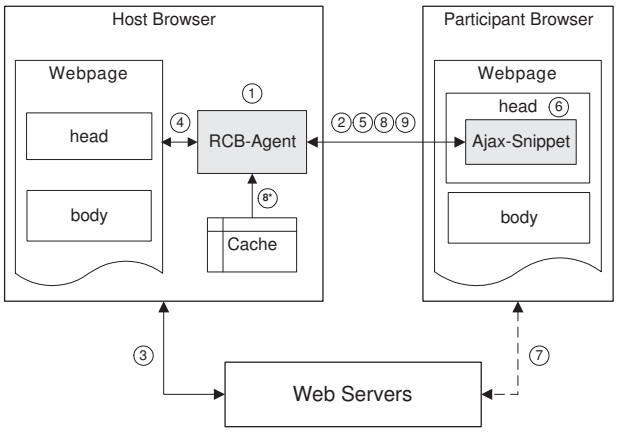

Figure 1: The architecture of the RCB framework.

In Figure 1, the *host browser* represents the browser used by a co-browsing host, and the *participant browser* corresponds to the browser used by a co-browsing participant. The *webpage* on each browser stands for a currently co-browsed HTML webpage. The displayed content of each webpage is the same on both browsers, but the source code of each webpage is different on the two browsers. The *cache* of the host browser is only read but not modified by RCB-Agent.

A co-browsing session can be broken down into nine steps. In step 1, a co-browsing host starts running RCB-Agent on the host browser with an open TCP port (e.g., 3000). In step 2, a co-browsing participant types the URL address of RCB-Agent (e.g., http://example-address:3000, where the example-address is a reachable hostname or IP address) into the address bar of the participant browser and sends a connection request to RCB-Agent. The RCB-Agent responds to a valid request by returning an *initial HTML page* that contains Ajax-Snippet. Ajax-Snippet will then periodically poll RCB-Agent, and the communication channel between the co-browsing host and participant is established.

On the host browser, whenever the co-browsing host visits a webpage (step 3), RCB-Agent monitors the internal browser-state changes and records file-downloading activities related to the webpage (step 4). When the webpage is loaded on the host browser, RCB-Agent creates an in-memory copy of the page's HTML document and makes necessary modifications to this copy. Then, in step 5, upon receipt of a polling request from a participant browser, RCB-Agent will send the content of the modified copy to the participant browser.

On the participant browser, Ajax-Snippet will analyze the received content and replace the corresponding HTML elements of the current page, in which Ajax-Snippet always resides, with the received content (step 6). In addition to the HTML document that describes the page structure, a webpage often contains supplementary

objects such as stylesheets, images, and scripts. Therefore, to accurately render the same webpage, the participant browser needs to download all these supplementary objects. Based on RCB-Agent's modifications on the copied HTML document, the RCB framework allows a participant browser to download these supplementary objects either from the original Web server (step 7), or directly from the host browser (step 8 and 8*).

Allowing a participant browser to directly download cached objects from the host browser can bring two attractive benefits to the co-browsing participant. One is that the co-browsing participant does not need to have the capability of establishing network connection with the original Web server (the connection marked in step 7 is denoted by a dashed line due to this reason). The other is that if the co-browsing participant has a fast network connection with the co-browsing host (e.g., they are within the same LAN), downloading cached objects from the host browser rather than from the remote Web server can often reduce the response time.

In step 9, any dynamic changes made (e.g., by JavaScript or Ajax) to a co-browsed webpage can be synchronized in real time from the host browser to the participant browser. Meanwhile, one user's (either a host user or a participant user) browsing actions such as form filling and mouse-pointer moving can be monitored and instantly mirrored to other users. When the co-browsing host visits new webpages, the loop from steps 3 to 9 is repeated. In a co-browsing session, users can visit different websites and collaboratively browse and operate on as many webpages as they like.

3.2 Decisions

The design of the RCB framework is mainly based on three decisions with respect to the communication model, the service model, and the synchronization model, respectively.

3.2.1 Direct Communication Model

Our RCB framework uses a direct communication model to support the collaboration between a co-browsing host and a co-browsing participant. A participant browser establishes a TCP connection to a host browser, without the support of any third-party platform, server, or proxy.

This direct communication model is simple, convenient, and widely applicable. Users in the same LAN can use Ethernet or Wi-Fi to establish their TCP connections. For WAN environments, if the host browser is running on a machine with a resolvable hostname or reachable IP address, remote co-browsing participants can use the hostname or IP address and an allowed TCP port to establish the connections; otherwise, a co-browsing host can still allow remote participants to reach a TCP port on a private IP address inside a LAN using port-forwarding [29] techniques. We also consider to integrate some NAT (network address translation) traversal techniques into RCB-Agent to further improve its accessibility.

3.2.2 HTTP-based Service Model

In our RCB framework, RCB-Agent on a host browser uses an HTTP-based service model to serve co-browsing participants. The key benefit of using this model is that there is no need for a co-browsing participant to make any installation or configuration. With the direct communication model, other service models (e.g., a peer-to-peer model or a non-HTTP based service model) exist but they all require changes at the participant side.

Integrating this HTTP-based service model into a browser also simplifies the host side installation since a co-browsing host only needs to install an RCB-Agent browser extension. Meanwhile, this browser integration approach maximizes the co-browsing capability because a browser extension normally can access both the content and related events of the browsed webpages. Furthermore, the end-user extensibility provided by modern Web browsers such as Internet Explorer and Firefox makes the implementation of this service model feasible.

3.2.3 Poll-based Synchronization Model

After the connection between a co-browsing host and its participant is established, Ajax-Snippet will periodically poll RCB-Agent to synchronize the co-browsing session. HTTP is a stateless protocol [8], and the communication is initiated by a client. Since the HTTP protocol does not support the push-based synchronization model, we use poll-based synchronization to emulate the effect of pushing webpage content and user interaction information between co-browsing users. In addition to poll-based synchronization, an HTTP server can use "multipart/x-mixed-replace" type of responses to emulate the content pushing effect. However, compared with poll-based synchronization, this alternative approach increases the complexity of co-browsing synchronization and decreases its reliability.

Ajax-Snippet is written in pure JavaScript. All popular Web browsers support Ajax techniques [20] and currently about 95% of Web users turn on JavaScript in their browsers [21]. Therefore, this synchronization model is well supported on users' regular browsers.

3.3 Co-browsing Topologies and Policies

The use of RCB is very flexible. Each co-browsing host can support multiple participants, and a participant can

join or leave a session at any time. A user can even host a co-browsing session and meanwhile join sessions hosted by other users using different browser windows or tabs. RCB-Agent knows exactly which participants are connected, and it can notify this information to a co-browsing host or participant.

Each co-browsing session is hosted and moderated by a co-browsing host. A participant's actions such as mouse click and data input are synchronized to the co-browsing host, and the co-browsing host will decide on further navigating actions. A participant browser never leaves the URL address of RCB-Agent, and contents from different websites and webpages are simply pushed to the participant browser. This tightly coupled scenario is typical for co-browsing applications (e.g., online training and customer support) that need a user to preside over a session, and it is also typical for co-browsing applications (e.g., online shopping) that require users to accomplish a common task on session-protected webpages.

To coordinate co-browsing actions among users, RCB-Agent can enforce different high-level policies for different application scenarios. For example, when a participant clicks a link on a co-browsed webpage and this action information is sent back to the host browser, RCB-Agent can either immediately perform the click action on the host browser, or ask the co-browsing host to inspect and explicitly confirm this click action. Similarly, if multiple participants are involved in a co-browsing session, it is up to the high-level policy enforced on RCB-Agent to decide whom are allowed to perform certain interactions and whose interaction action will be finally submitted to a website. However, the specification and enforcement of co-browsing policies is usually application-dependent, and it is out of the scope of this paper.

3.4 Security Design and Analysis

For a co-browsing participant, using RCB is as secure as visiting a trusted HTTP website. This is because a participant only needs to type in the URL address of RCB-Agent given by a trusted co-browsing host and then perform regular browsing actions such as clicking and form-filling on a regular Web browser. We therefore keep the focus of our security design on the protection of RCB-Agent by authenticating its received requests.

Our current design on request authentication is based on a conventional mechanism of sharing a session secret key and computing the keyed-Hash Message Authentication Code (HMAC). On a host browser, a session-specific one-time secret key is randomly generated and used by RCB-Agent. The co-browsing host shares the secret key with a participant using some out-of-band mechanisms such as telephone calls or instant messages. On a participant browser, the secret key is typed in by a co-browsing participant via a password field on the *initial HTML page* and then stored and used by Ajax-Snippet.

Before sending a request, Ajax-Snippet computes an HMAC for the request and appends the HMAC as an additional parameter of the request-URI. After receiving a request sent by Ajax-Snippet, RCB-Agent computes a new HMAC for the received request (discarding the HMAC parameter) and verifies the new HMAC against the HMAC embedded in the request-URI. The data integrity and the authenticity of a request are assured if these two HMACs are identical. Since the size of a request sent by Ajax-Snippet is small, an HMAC can be efficiently calculated and any important information in a request can also be efficiently encrypted using a JavaScript implementation [25]. However, using JavaScript to compute an HMAC for a response (or encrypt/decrypt a response) is inefficient, especially if the size of the response is large. We plan to integrate other security mechanisms to address this issue in the future.

4 Implementation Details

Although the design of the proposed RCB framework is relatively simple and straightforward, its implementation poses several challenges. The implementation of RCB-Agent has two major challenges: (1) how to efficiently process requests so that participant browsers can be synchronized in real time, and (2) how to accurately generate response contents so that fine-grained high-quality co-browsing activities can be easily supported. The key implementation challenge of Ajax-Snippet lies in how to properly and smoothly update webpage contents on a participant browser. We have implemented the RCB framework in Firefox and successfully addressed these challenges. We present the implementation details of the framework in this section.

4.1 RCB-Agent

RCB-Agent is implemented as a Firefox browser extension, and it is purely written in JavaScript. Its request processing and response content generation functionalities are detailed as follows.

4.1.1 Request Processing

The request processing functionality of RCB-Agent is implemented as a JavaScript object of Mozilla's nsIServerSocket interface [33]. This interface provides methods to initialize a server socket and maintain it in the listening state. For this server socket object, we create a socket listener object which implements the methods of Mozilla's nsIServerSocketListener interface [33]. RCB-Agent uses this socket listener object to asynchronously

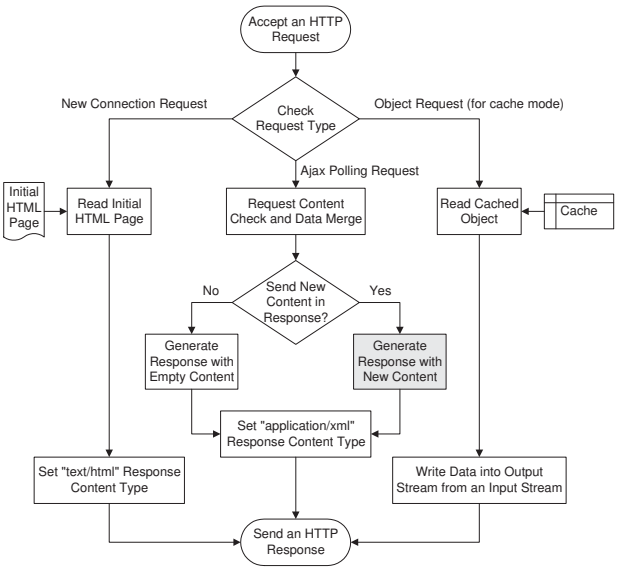

Figure 2: Request processing procedure of RCB-Agent.

listen for and accept new TCP connections. We also create a data listener object which implements Mozilla's nsIStreamListener interface [33]. We associate this data listener object with the input stream of each connected socket transport. Therefore, over each accepted TCP connection, RCB-Agent uses this data listener object to asynchronously accept incoming HTTP requests and efficiently process them.

Figure 2 illustrates the high-level request processing procedure of RCB-Agent. From a participant browser, RCB-Agent may receive three types of HTTP requests: a *new connection request*, an *object request*, and an *Ajax polling request*. RCB-Agent identifies the type of request by simply checking the *method* token and *request-URI* token in the request-line [8]. Both a new connection request and an object request use the "GET" method, but they can be differentiated by checking their request-URI tokens. The former has a root URI, but the later has a URI pointing to a specific resource such as an image file. Ajax polling requests always use the "POST" method because we want to directly piggyback action information of a co-browsing participant onto a polling request.

A new connection request is sent to RCB-Agent after the URL of RCB-Agent is entered into the address bar of a participant browser. RCB-Agent responds to this request by sending back a "text/html" type of HTTP response to the participant browser with the content of an *initial HTML page*. The head element of this initial HTML page contains Ajax-Snippet, which will later send Ajax polling requests to RCB-Agent periodically.

An object request is sent to RCB-Agent if the *cache mode* is used to allow a participant browser to directly download a cached object from the host browser. RCB-Agent keeps a mapping table, in which the request-URI of each cached object maps to a corresponding cache key. After obtaining the cache key for a request-URI, RCB-Agent reads the data of a cached object by creating a cache session via Mozilla's cache service [33]. To save time and memory, RCB-Agent directly writes data from the input stream of the cached object into the output stream of the connected socket transport.

An Ajax polling request is sent by Ajax-Snippet from a participant browser to check if any page content changes or browsing actions have occurred on the host browser. RCB-Agent follows three steps to process an Ajax polling request: data merging, timestamp inspection, and response sending.

Data merging: RCB-Agent examines the content of a "POST" type Ajax polling request and may merge data if the content contains piggybacked browsing action information of the co-browsing participant. For example, if users are co-filling a form, the form data submitted by a co-browsing participant can be extracted and merged into the corresponding form on the host browser.

Timestamp inspection: RCB-Agent looks for any new content needs to be sent back to the co-browsing participant. RCB-Agent uses a simple timestamp mechanism to ensure that only new content, which has never been sent to this participant before, is included in the response message. A timestamp used here is the number of milliseconds since midnight of January 1, 1970. RCB-Agent maintains a timestamp for the latest webpage content on the host browser. Whenever this new content is sent to a participant browser, its timestamp is also included in the same response. Each Ajax polling request from a participant browser carries back the timestamp of its current webpage content, so RCB-Agent can compare the current timestamp on the host browser and the received one to accurately determine whether the page content on each particular participant browser needs to be updated.

Response sending: if any new content needs to be sent to a participant browser, RCB-Agent generates a response with the new content. Response content generation is an important functionality of RCB-Agent, and it is detailed in the following subsection. To facilitate efficient content parsing in a participant browser, RCB-Agent sends out the new content in the form of an XML document using the "application/xml" type of HTTP response. If no new content needs to be sent back, RCB-Agent sends a response with empty content to the participant browser in order to avoid hanging requests.

4.1.2 Response Content Generation

The response content generation functionality of RCB-Agent generates responses with new content for Ajax polling requests. It guarantees that webpage content can

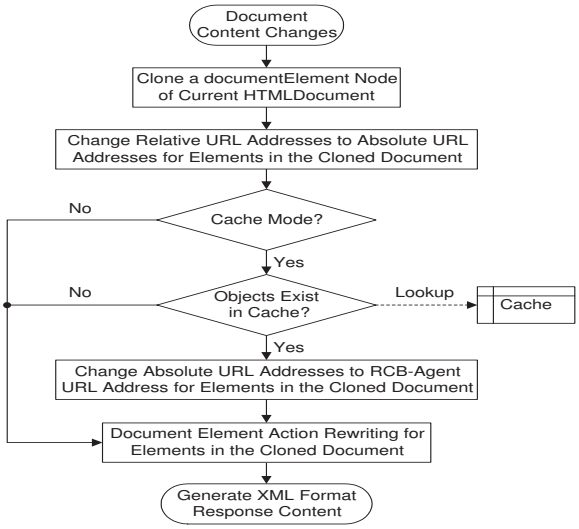

Figure 3: Response content generation procedure of RCB-Agent.

be efficiently extracted on a host browser and later on accurately rendered on a participant browser. The high-quality implementation of this functionality is essential for adding upper-level co-browsing features such as form co-filling and action synchronization.

Figure 3 illustrates the high-level response content generation procedure of RCB-Agent. When document content changes on the host browser need to be sent to a participant browser, RCB-Agent uses the following five steps to generate the XML format response content. First, RCB-Agent clones the *documentElement* node (namely the "<html>" root element of an HTML webpage) of the current *HTMLDocument* object on the host browser. The following changes are made only to the cloned *documentElement* node (referred to as the cloned document) so that the content generation procedure will not cause any state change to the current document on the host browser.

In the second step, for the supplementary objects of the cloned document, RCB-Agent changes all the relative URL addresses to absolute URL addresses of the original Web servers. This URL conversion is necessary to support RCB's *non-cache* mode in which a participant browser needs to use absolute URL addresses to correctly download supplementary objects from original Web servers. To achieve an accurate URL conversion, we create an observer object which implements the methods of Mozilla's nsIObserverService [33]. Using this observer object, RCB-Agent can record complete URL addresses for all the object downloading requests.

In the third step, if the cache mode is used, for the supplementary objects of the cloned document that exist in the browser cache, their absolute URL addresses are changed to RCB-Agent URL addresses. Subsequently,

```
<?xml version='1.0' encoding='utf-8'?>
<newContent>
  <docTime>documentTimestamp</docTime>
  <docContent>
    <docHead>
      <hChild1><![CDATA[escape(hData1)]]></hChild1>
      <hChild2><![CDATA[escape(hData2)]]></hChild2>
      ......
    </docHead>
    <!-- for a page using body element -->
    <docBody><![CDATA[escape(bData)]]></docBody>
    <!-- for a page using frames -->
    <docFrameSet><![CDATA[escape(fData)]]>
    </docFrameSet>
    <docNoFrames><![CDATA[escape(nData)]]>
    </docNoFrames>
  </docContent>
  <userActions>userActionData</userActions>
<newContent>
```

Figure 4: XML format response content.

when a participant browser renders the page content, it will automatically send "GET" type of HTTP requests to RCB-Agent to retrieve cached objects. For the non-cache mode, nothing needs to be done in this step. Switching between these two modes is very flexible and fully controlled by RCB-Agent. For example, RCB-Agent can allow different participant browsers to use different modes, allow different webpages sent to a particular participant browser to use different modes, and even allow different objects on the same webpage to use different modes.

In the fourth step, RCB-Agent rewrites event attributes such as *onclick* and *onsubmit* for children elements of the cloned document. The purpose of this rewriting is to enable upper-level co-browsing features such as form co-filling and action synchronization. For instance, to support the form co-filling feature, RCB-Agent changes the *onsubmit* event attribute values of form elements in the cloned document. More specifically, RCB-Agent adds a call to a specific JavaScript function residing in Ajax-Snippet to each form's *onsubmit* event handler. So later on, when a form is submitted on a participant browser, this JavaScript function is called and the related form data can be carried back by an Ajax polling request to the host browser.

Finally, after making the above changes, RCB-Agent generates an XML format response content for this Ajax polling request. From top-level children of the cloned document, RCB-Agent follows their order in the DOM (Document Object Model [23]) tree to process these elements, including extracting their attribute name-value lists and *innerHTML* values. For most webpages, the cloned document only contains two top-level children: a head element and a body element. For some webpages, their top-level children may include a head element, a frameset element, and probably a noframes element.

Figure 4 illustrates the simplified XML format of the generated response content. The *newContent* element contains a *docTime* element that carries the document

timestamp string, a *docContent* element that carries the data extracted from the cloned document, and a *userActions* element that can carry additional browsing action (such as mouse-pointer movement) information.

Within the *docContent* element, for each child element of the cloned document head, its attribute name-value list and innerHTML value are encoded using the JavaScript *escape* function and carried inside the *CDATA* section of a corresponding *hChild* element. For example, *hChild1* may contain the data for the title child element of the head, and *hChild2* may contain the data for a style element of the head. The contents of these children head elements are separately transmitted so that later Ajax-Snippet can properly and easily update document contents on different types of browsers such as Firefox and Internet Explorer. Similarly, the name-value lists and innerHTML values extracted from other top-level children (e.g., body or frameset) of the cloned document are carried in the CDATA sections of their respective elements. We use the escape encoding function and CDATA section to ensure that the response data can be precisely contained in an "application/xml" message and correctly transmitted over the Internet.

The generation of this XML format response content combines both the structural advantages of using DOM and the performance and simplicity advantages of using innerHTML. This implementation ensures that the response content can be efficiently generated on a host browser; more importantly, it guarantees the same webpage content can be accurately and efficiently rendered on a participant browser. The innerHTML property is well supported by all popular browsers and has been included into the HTML 5 DOM specification. Note that the whole response content generation procedure is executed only once for each new document content, and the generated XML format response content is reusable for multiple participant browsers. Also note that RCB-Agent does not replicate HTTP cookies or the *referer* request header to a participant browser. We can extend RCB-Agent to have these capabilities, but in our experiments we did not observe the necessity to do so because a participant browser can download supplementary objects of a webpage from a website (in the non-cache mode) or RCB-Agent (in the cache mode) for both HTTP and HTTPS sessions.

4.2 Ajax-Snippet

Ajax-Snippet is implemented as a set of JavaScript functions. It is embedded in the head element of the initial HTML page and sent to a participant browser as a part of RCB-Agent's response to a new connection request. Ajax-Snippet uses the XMLHttpRequest object [31] to asynchronously exchange data with RCB-Agent.

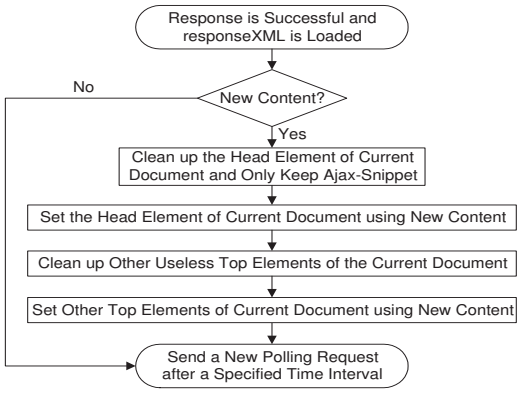

Figure 5: Response processing procedure of Ajax-Snippet.

4.2.1 Ajax Request Sending

Sending Ajax requests is relatively simple for Ajax-Snippet. The first Ajax request is sent after the initial HTML page is loaded on a participant browser. Each following Ajax request is triggered after the response to the previous one is received. A new XMLHttpRequest object is created to send each Ajax request. An "onreadystatechange" event handler is registered for an XMLHttpRequest object to asynchronously process its *readystatechange* events. The XMLHttpRequest object uses the "POST" method so that action information of a co-browsing participant can be directly piggybacked in an Ajax polling request. Before a request is sent out, its *Content-Length* request header needs to be correctly set.

4.2.2 Ajax Response Processing

It is more challenging for Ajax-Snippet to properly process Ajax responses and smoothly update webpage content on a participant browser. Figure 5 illustrates the high-level response processing procedure of Ajax-Snippet. This procedure is implemented in the "onreadystatechange" event handler. It is triggered when a response is successful (HTTP status code sent by RCB-Agent is 200) and the data transfer has been completed (readyState is "DONE" and responseXML is loaded) for an XMLHttpRequest. If RCB-Agent indicates "no new content" with an empty response content, Ajax-Snippet simply uses the JavaScript *setTimeout* function to send a new polling request after a specified time interval; otherwise, Ajax-Snippet will update the current webpage document in which it resides, using the new content contained in the responseXML document.

A new content could be either a brand new webpage or an update to the existing webpage. To make the content update process smooth and simple on a participant browser, Ajax-Snippet follows a specific four-step procedure. First, Ajax-Snippet cleans up other content

in the head element of the current document, but it always keeps itself as a "<script>" child element within the head element of any current document. Next, Ajax-Snippet extracts the attribute name-value lists and innerHTML values from the *docHead* element of the new content (shown in Figure 4) and appends them to the head element of the current document. Ajax-Snippet detects browser capability and executes this step differently to best accommodate different browser types. For example, since the innerHTML property of the head element is writable in Firefox, Ajax-Snippet will directly set the new value for it. In contrast, the innerHTML property is read-only for the head element (and its style child element) in Internet Explore, so Ajax-Snippet will construct each child element of the head element using DOM methods (e.g., createElement and appendChild).

After properly updating the content of the head element in the above two steps, Ajax-Snippet will then check the new content and clean up other useless top-level elements of the current document. For example, if the current document uses a body top-level element while the new content contains a new webpage with a frameset top-level element, Ajax-Snippet will remove the body node of the current document. Finally, Ajax-Snippet sets other attribute name-value lists and innerHTML values of the current document based on the data extracted from the new content, following their order in the XML format.

The above procedure ensures that the webpage content on a participant browser can be accurately and smoothly synchronized to that on the host browser. Meanwhile, Ajax-Snippet always resides in the current webpage on a participant browser to maintain the communication with the host browser. After updating the current document with the new content, Ajax-Snippet sends a new polling request to RCB-Agent, in the same way as it does for the "no new content" case.

It is also worth mentioning that any dynamic DOM changes on a host browser are synchronized to a participant browser. Since Ajax-Snippet updates the content mainly using innerHTML, the code between a pair of "<script>" and "</script>" tags will not be executed automatically in both Firefox and Internet Explore. However, event handlers previously rewritten by RCB-Agent can be triggered. The executions of these event handlers on a participant browser will not directly update any URL or change the DOM; they just ask Ajax-Snippet to send action information back to the host browser.

5 Evaluations

In this section, we present the performance evaluation and usability study of our RCB framework.

5.1 Performance Evaluation

To quantify the performance of our RCB framework, we conducted two sets of experiments: one in a LAN environment and the other in a WAN environment.

5.1.1 Experimental Methods

The homepages of 20 sample websites (shown in Table 1) were used for co-browsing experiments. These websites were chosen from the top 50 sites listed by Alexa.com, with a few diversity-related criteria (such as geographical location and content category) taken into consideration.

We introduce six metrics to evaluate the real-time performance of the RCB framework: **M1**, the time used by a host browser to load the HTML document of a homepage from a Web server; **M2**, the time used by a participant browser to load the content of the same HTML document from the host browser; **M3**, the time used by the participant browser to download the supplementary Web objects (of the HTML document) in the non-cache mode; **M4**, the time used by the participant browser to download the supplementary Web objects (of the HTML document) in the cache mode; **M5**, the time used by the host browser to generate the response content for an HTML document; **M6**, the time used by the participant browser to update its current document based on the new content of an HTML document.

Intuitively, the metric M1 measures the download speed of an HTML document while the metric M2 measures the synchronization speed of the HTML document. We use M3 and M4 to determine whether using the cache mode is beneficial to a co-browsing participant. The metrics M5 and M6 quantify the speed of RCB-Agent in response content generation (i.e., the procedure illustrated in Figure 3) and the speed of Ajax-Snippet in response processing (i.e., the procedure illustrated in Figure 5), respectively. User browsing action information (such as form co-filling data) can be carried in a small-sized request or response and efficiently transmitted, so we do not present the detailed results.

In each experimental environment, we used one host browser and one participant browser. The polling time interval of Ajax-Snippet was set to one second, which we believe is small enough because users' average think time on a webpage is about ten seconds [16]. We co-browsed all the 20 sample sites in the cache mode for the first round and then in the non-cache mode for the second round. Both browsers were directly connected to the Internet without using any proxies. Before each round of co-browsing, the caches of both browsers were cleaned up. This procedure was repeated five times and we present the average results.

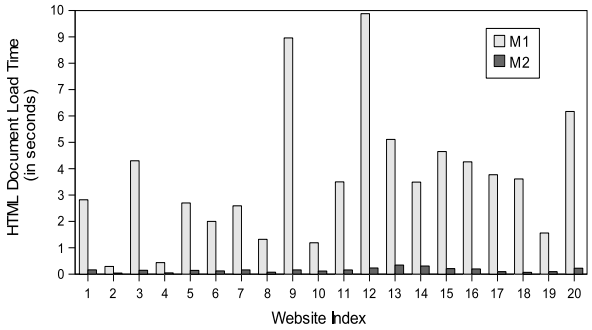

Figure 6: HTML document load time in the LAN environment.

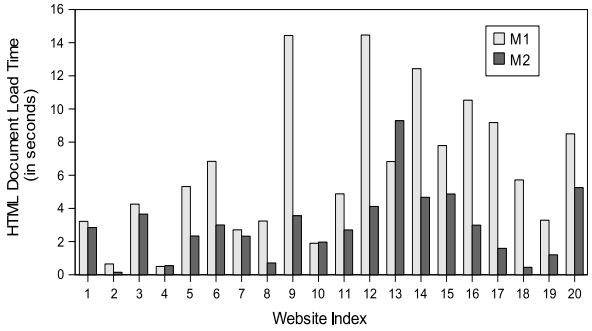

Figure 7: HTML document load time in the WAN environment.

5.1.2 Experimental Results

The first set of experiments were conducted in a 100Mbps Ethernet LAN environment, where the host and participant PCs resided in the same campus network. The second set of experiments were performed in a WAN environment, where the host and participant PCs resided in two geographically separated homes. Both homes used slow speed Internet access services with 1.5Mbps download speed and 384Kbps upload speed.

Figure 6 shows the comparison between metrics M1 and M2 in the LAN environment, and Figure 7 presents the same comparison in the WAN environment. In the LAN environment, for all the 20 sample sites, the values of M2 are less than 0.4 seconds, which are much smaller than those of M1. In other words, the HTML document content synchronization delay experienced by the participant browser is much smaller than the time it has to spend to directly download the HTML document from a remote Web server. This result is expected since the host PC and participant PC were in the same LAN. In the WAN environment, the values of M2 become larger than those in the LAN environment. This is mainly because the upload link speed at the host PC side was slow (only 384Kbps). However, we can see that most values of M2 (17 out of 20 sample sites) are still smaller than those of M1, indicating an acceptable content synchronization speed.

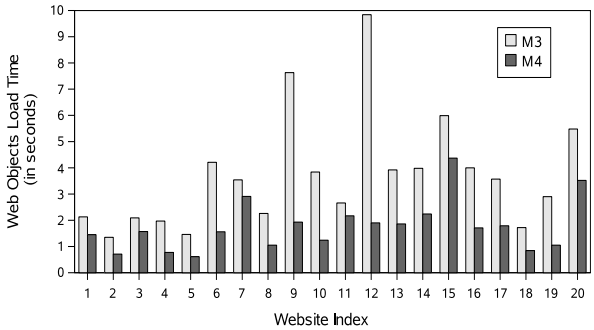

Figure 8: Cache mode performance gain in the LAN environment.

Index	Site Name	Page Size (KB)	M5 *non-cache* (second)	M5 *cache* (second)	M6 (second)
1	yahoo.com	130.3	0.066	0.098	0.135
2	google.com	6.8	0.015	0.020	0.045
3	youtube.com	69.2	0.107	0.172	0.126
4	live.com	20.9	0.019	0.037	0.057
5	msn.com	49.6	0.079	0.145	0.119
6	myspace.com	53.2	0.085	0.097	0.126
7	wikipedia.org	51.7	0.113	0.138	0.171
8	facebook.com	23.2	0.029	0.036	0.067
9	yahoo.co.jp	101.4	0.111	0.156	0.154
10	ebay.com	50.5	0.049	0.098	0.100
11	aol.com	71.3	0.099	0.189	0.142
12	mail.ru	83.8	0.176	0.346	0.268
13	amazon.com	228.5	0.371	0.687	0.318
14	cnn.com	109.4	0.298	0.599	0.280
15	espn.go.com	110.9	0.175	0.376	0.194
16	free.fr	70.0	0.211	0.279	0.222
17	adobe.com	37.3	0.050	0.085	0.086
18	apple.com	10.0	0.029	0.056	0.118
19	about.com	35.8	0.056	0.100	0.081
20	nytimes.com	120.0	0.221	0.382	0.196

Table 1: Homepage size and processing time of 20 sites.

Figure 8 illustrates the comparison between metrics M3 and M4 in the LAN environment. We can see that the values of M4 are less than those of M3 for all the 20 sample sites. It means that for the participant browser, downloading the supplementary Web objects from the host browser is faster than retrieving them from the remote Web server. This result is expected as well since the co-browsing PCs were in the same LAN. Therefore, we suggest to turn on the cache mode in LAN environments so that co-browsing participants can take advantage of the performance gain provided by cache. In the WAN environment, co-browsing participants can still benefit from the cache at the host side although the performance gain is not as significant as that in the LAN environment. We omit the details to save space.

Table 1 lists the homepage size of the sample sites and the processing time in terms of the M5 metric for both the non-cache mode and cache mode, and the M6 metric. Based on the results in the table, we have the following observations. First, the larger the HTML document size is, the more processing time is needed. Second, RCB-Agent can efficiently generate the response content for an HTML document. Most pages (16 out of 20 for M5

non-cache, and 14 out of 20 for M5 cache) can be processed in less than 0.2 seconds. Since a generated new content can be reused by multiple co-browsing participants, this processing time on the host browser is reasonably small. Third, RCB-Agent needs more processing time in the cache mode than in the non-cache mode, i.e., the values of M5 cache are greater than those of M5 non-cache. This is because extra cache lookup time is spent in the cache mode. However, this small cost is outweighed by the benefits of using the cache-mode for co-browsing participants, especially in LAN environments as shown above. Finally, Ajax-Snippet can efficiently update webpage content on a participant browser. As indicated by the values of the M6 metric, this processing time is less than one-third of a second for all the 20 webpages.

5.2 Usability Evaluation

To measure whether our RCB framework is helpful and easy to use, we conducted a usability study based on two real co-browsing scenarios: (1) coordinating a meeting spot via Google Maps, and (2) online co-shopping at Amazon.com. In the remainder of this section, we first introduce these two scenarios and explain why we chose them. We then present and analyze the usability study.

5.2.1 Coordinating a Meeting Spot via Google Maps

Suppose Alice is going to visit New York City. She plans to meet her local cousin Bob at the Cartier jewelry store on the Fifth Avenue in Manhattan to buy a watch. Bob wants to use Google Maps to show Alice the direction to the store. Since the neighborhood around the Fifth Avenue in Manhattan is extremely crowded, Bob uses our RCB tool to give Alice accurate directions to the exact meeting spot.

Bob hosts a co-browsing session and Alice joins the session. Bob then searches the store address using Google Maps. He may zoom in and out of the map, drag the map, and show different views of the map. Whatever content Bob is seeing on his browser is instantly and accurately synchronized to Alice's browser. Figure 9 shows one snapshot of the destination map shown on Alice's browser. Bob may even use the street-view Flash of Google Maps to show Alice panoramic street-level views of the meeting spot. Note that our current implementation does not support the synchronization of users' actions on a Flash, so Alice and Bob can only individually operate on a Flash. During the session, they may use an instant message tool (e.g., MSN Messenger) or telephone as the supplementary communication channel to mediate actions. Eventually Alice and Bob come to the agreement that they will meet outside the four red roof show-windows of Cartier on the Fifth Avenue side.

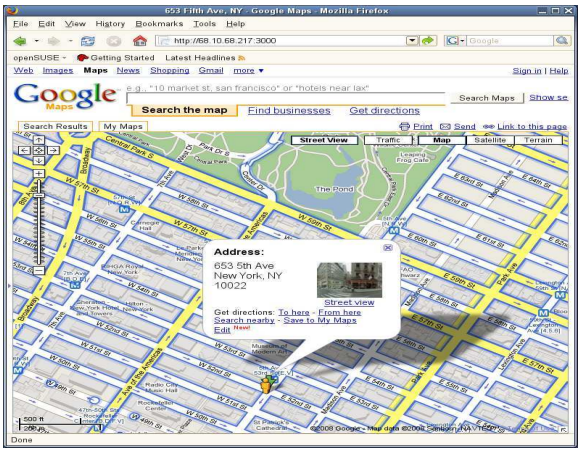

Figure 9: Snapshot of the destination map shown on Alice's browser.

This scenario exemplifies that our RCB framework can efficiently support rich Web contents and communication intensive webpages. Google Maps actually also uses Ajax to asynchronously retrieve small images (usually in the size of 256 by 256 pixels) and smoothly update the map content grid by grid. With our RCB tool, one user's view is further synchronized accurately and smoothly to another user's browser, achieving real-time collaborative browsing. In general, the URL in the address bar remains the same even if the webpage content has been updated by Ajax and many other DHTML (Dynamic HTML) techniques. Therefore, without RCB, the map content changes caused by Bob's browsing actions such as zooming and panning cannot be synchronized to Alice by simply sharing URLs.

5.2.2 Online Co-shopping at Amazon.com

Bob is going to buy a present for his cousin Alice. Bob hosts a co-browsing session and Alice joins the session. They co-browse a number of webpages at Amazon.com to select a newly-released MacBook Air laptop favored by Alice. Both Alice and Bob can type in, search and click on a webpage. Bob's browsing requests will be directly sent to Amazon.com, but Alice's action information such as searching or clicking is first sent back to the RCB-Agent on Bob's browser and then sent out to Amazon.com. After they made the decision, Bob adds the selected laptop to the shopping cart and uses his account to start the checkout procedure. Bob can ask Alice to co-fill some forms (e.g., the shipping address form) using her information, and he finishes the rest of the checkout procedure. Figure 10 shows the snapshot of the form filling window on Bob's browser, on which the form data is sent back from Alice's browser.

The online shopping scenario verifies that our RCB tool can: (1) correctly synchronize webpages with very

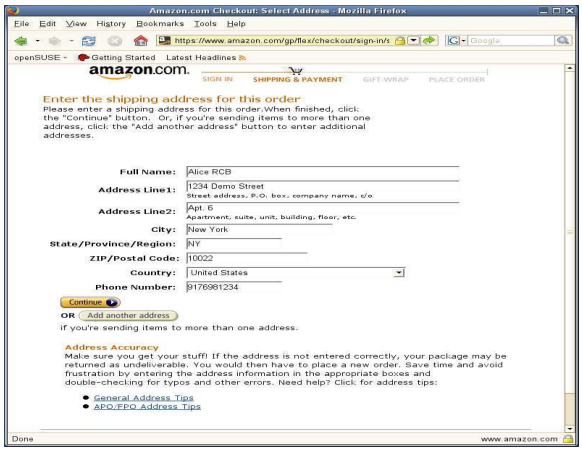

Figure 10: Snapshot of the form filling window on Bob's browser.

complicated layout and dynamically-generated content, (2) allow anyone in a co-browsing session to initiate browsing actions and navigate to new pages, (3) support co-browsing features such as form co-filling and mouse clicking, and (4) support session-protected webpages.

5.2.3 Usability Study

The main objective of the usability study is to measure whether our RCB tool is helpful and easy to use.

(1) Test subjects: A total of 20 adults, 11 females and 9 males, participated as users in our study. These test subjects were undergraduate and graduate students who were randomly recruited from nine degree programs at our university. Eighteen test subjects were between ages of 18 and 30, and two were over 30 years old. Nineteen test subjects were using the Internet daily, and one was using it weekly. We did not screen test subjects based on experience using Firefox because they simply had to perform tasks (such as entering URLs and interacting with webpages) that are common to different browsers. We also did not screen test subjects based on experience using Google Maps or shopping at Amazon.com.

(2) Procedure and Tasks: We combined the two scenarios (Google Maps and Amazon.com) introduced above into a single co-browsing session. Each session consists of 20 tasks as listed in Table 2. Ten tasks were performed by Bob and ten tasks were performed by Alice, and Alice and Bob represent two role-players regardless of their actual genders. The 20 test subjects were randomly grouped into 10 pairs. We asked each pair of test subjects to complete two sessions. In the first session, we randomly asked one test subject to act as Alice and the other test subject to act as Bob. After the two test subjects finished the 20 tasks in a session, they switched their roles to perform the 20 tasks in the second session.

The two test subjects in a pair were asked to use two computers located at different locations either in our department or in the library of university. We pre-installed RCB-Agent to the Firefox browser on Bob's computer so that we can keep the focus of the study on using the RCB tool itself. Before a pair of test subjects started performing the tasks, we explained the main functionality of RCB and how to use it. We also gave them an instruction sheet that describes the two scenarios and lists the tasks to be completed by a role-player.

Task#	Brief Task Description
T1-B	Bob starts a RCB co-browsing session using a Firefox browser.
T1-A	Alice types the URL told by Bob in a Firefox browser to join the session.
T2-B	Bob searches the location "653 5th Ave, New York" using Google Maps.
T2-A	Alice tells Bob that the map of the location is automatically shown on her browser.
T3-B	Bob zooms in and out of the map, drags up/down/left/right the map.
T3-A	Alice tells Bob that the map is automatically updated on her browser.
T4-B	Bob clicks to the street-view of the searched location.
T4-A	Alice tells Bob that the street-view is also automatically shown on her browser.
T5-B	Bob tells Alice to meet outside the four red roof show-windows of Cartier shown in the street-view.
T5-A	Alice finds the four red roof show-windows of Cartier and agrees with the meeting spot.
T6-B	Bob continues to visit the homepage of Amazon.com website.
T6-A	Alice tells Bob that the homepage of Amazon.com is automatically shown on her browser.
T7-B	Bob searches and clicks to find a MacBook Air laptop at the Amazon.com website.
T7-A	Alice tells Bob that the pages are automatically updated on her browser.
T8-B	Bob asks Alice to search and click on the pages shown on her browser to choose a different MacBook Air laptop.
T8-A	Alice chooses a different MacBook Air laptop and tells Bob that this laptop is her final choice.
T9-B	Bob adds the selected laptop to the shopping cart and starts the checkout procedure.
T9-A	Alice fills the shipping address form shown on her browser.
T10-B	Bob finishes the rest of the checkout procedure.
T10-A	Alice leaves the co-browsing session.

Table 2: The 20 tasks used in a co-browsing session. Alice and Bob are two role-players. Bob performs ten tasks from T1-B to T10-B, and Alice performs ten tasks from T1-A to T10-A. Bob and Alice use a voice supplementary communication channel to mediate actions.

(3) Data Collection: We collected data in two ways: through observation and through two questionnaires. During each co-browsing session, two experimenters sat with each test subject to observe the progress of the tasks. After completing two co-browsing sessions, each test subject was asked to answer a five-point Likert-scale (Strongly disagree, Disagree, Neither agree nor disagree, Agree, Strongly Agree) [27] questionnaire. The 16 questions in this questionnaire are listed in Table 3. In addition to this close-ended questionnaire, each test subject was also asked to answer an open-ended questionnaire to solicit additional feedback. After finishing the two questionnaires and before leaving, each test subject was given a $5 gift card as compensation for the participation.

(4) Results and Analysis: Through observation, we found that the 10 pairs of test subjects successfully completed all their co-browsing sessions. Each pair of test subjects took an average of 10.8 minutes to complete

Perceived Usefulness
Q1-P: It is helpful to use RCB to coordinate a meeting spot via Google Maps.
Q1-N: It is useless to use RCB to coordinate a meeting spot via Google Maps.
Q2-P: It is helpful to use RCB to perform online co-shopping at Amazon.com.
Q2-N: It is useless to use RCB to perform online co-shopping at Amazon.com.
Ease-of-use as a co-browsing host
Q3-P: It is easy to use RCB to host the Google Maps scenario.
Q3-N: It is hard to use RCB to host the Google Maps scenario.
Q4-P: It is easy to use RCB to host the online co-shopping scenario.
Q4-N: It is hard to use RCB to host the online co-shopping scenario.
Ease-of-use as a co-browsing participant
Q5-P: It is easy to participate in the RCB Google Maps scenario.
Q5-N: It is hard to participate in the RCB Google Maps scenario.
Q6-P: It is easy to participate in the RCB online co-shopping scenario.
Q6-N: It is hard to participate in the RCB online co-shopping scenario.
Potential Usage
Q7-P: It would be helpful to use RCB on other co-browsing activities.
Q7-N: It wouldn't be helpful to use RCB on other co-browsing activities.
Q8-P: I would like to use RCB in the future.
Q8-N: I wouldn't like to use RCB in the future.

Table 3: The 16 close-ended questions in four groups. Test subjects were not aware of the groupings. From Q1-P to Q8-P are eight positive Likert questions, and from Q1-N to Q8-N are eight correspondingly inverted negative Likert questions. These questions were presented to a test subject in random order to reduce response bias.

two sessions. Such a 100% success ratio may be attributable to two main reasons. One is that all the 20 test subjects were frequent Internet users and they might be familiar with online shopping and Web mapping service sites. The other reason is that RCB does not add any new user interface artifact and users simply use regular Web browsers, visit regular websites, and perform regular browsing activities.

A summary of the responses to the 16 close-ended questions is presented in Table 4. Since the data collected are ordinal and do not necessarily have interval scales, we used the median and mode to summarize the data and used the percentages of responses to express the variability of the results. Overall, the test subjects were very enthusiastic about this RCB tool. The median and mode responses are positive *Agree* for all the questions. In terms of the perceived usefulness (Q1-P, Q1-N, Q2-P, Q2-N), 52.5% of responses agree and 40.0% of responses strongly agree that it is helpful to use RCB in both the Google Maps scenario and the Amazon.com scenario.

In terms of the ease-of-use as a co-browsing host (Q3-P, Q3-N, Q4-P, Q4-N), 50.0% of responses agree and 40.0% of responses strongly agree that it is easy to use RCB to host the Google Maps scenario, and 62.5% of responses agree and 27.5% of responses strongly agree that it is easy to use RCB to host the online co-shopping scenario. In terms of the ease-of-use as a co-browsing participant (Q5-P, Q5-N, Q6-P, Q6-N), 62.5% of responses agree and 35.0% of responses strongly agree that it is easy to participate in the RCB Google Maps scenario, and 57.5% of responses agree and 35.0% of responses

	Strongly disagree	**Disagree**	**Neither agree nor disagree**	**Agree**	**Strongly Agree**	**Median**	**Mode**
Q1-P	0.0%	0.0%	7.5%	52.5%	40.0%	Agree	Agree
Q2-P	0.0%	0.0%	7.5%	52.5%	40.0%	Agree	Agree
Q3-P	5.0%	0.0%	5.0%	50.0%	40.0%	Agree	Agree
Q4-P	0.0%	2.5%	7.5%	62.5%	27.5%	Agree	Agree
Q5-P	0.0%	2.5%	0.0%	62.5%	35.0%	Agree	Agree
Q6-P	0.0%	5.0%	2.5%	57.5%	35.0%	Agree	Agree
Q7-P	0.0%	2.5%	5.0%	55.0%	37.5%	Agree	Agree
Q8-P	0.0%	0.0%	15.0%	55.0%	30.0%	Agree	Agree

Table 4: Summary of the responses to the 16 close-ended questions. To provide statistical coherence, we inverted the scores to the eight negative Likert questions (Q1-N to Q8-N) about the neutral mark (i.e., Strongly agree to Strongly disagree, Agree to Disagree, and vice versa) and then merged them with the scores to the corresponding positive Likert questions (Q1-P to Q8-P).

strongly agree that it is easy to participate in the RCB online co-shopping scenario. These two groups of results also indicate that participating a co-browsing session is slightly easier than hosting a session.

In terms of the potential usage (Q7-P, Q7-N, Q8-P, Q8-N), 55.0% of responses agree and 37.5% of responses strongly agree that it would be helpful to use RCB on other co-browsing activities, and 55.0% of responses agree and 30.0% of responses strongly agree that the test subject would like to use RCB in the future.

In our open-ended questionnaire, the test subjects were asked to write down whatever they think about the RCB tool. One test subject did not write anything, but nineteen test subjects wrote many positive comments such as "cool", "it helps cooperation", "useful", "simple operation", and "love it, fascinating and useful". Meanwhile, some test subjects also wrote a few suggestions and expectations to the RCB tool. For example, two test subjects suggested that indicators of the other person's connection and status may be needed. Four test subjects mentioned that it would be great if actions in the Google Maps street-view Flash could also be synchronized. Seven test subjects expressed that on some pages the wait time is a bit long, but it is not bad at all.

In summary, the results of the usability study clearly demonstrate that RCB is very helpful and easy to use. It is a simple and practical real-time collaborative browsing tool that people would like to use in their everyday browsing activities.

6 Conclusion

We have presented a simple and practical framework for Real-time Collaborative Browsing (RCB). Leveraging the power of Ajax techniques and the end-user extensibility of modern Web browsers, RCB enables real-time collaboration among Web users without the involvement of any third-party platforms, servers, or proxies.

A co-browsing host only needs to install an RCB-Agent browser extension, and co-browsing participants just use their regular JavaScript-enabled Web browsers. We detailed the design and the Firefox version implementation of the RCB framework. We measured the real-time performance of RCB through extensive experiments, and we validated its high quality co-browsing capabilities using a formal usability study. The evaluation results demonstrate that our RCB framework is simple, practical, helpful and easy to use.

In our future work, we plan to explore co-browsing in mobile computing environments. We have recently ported our RCB-Agent implementation to the Fennec Web browser, which is the mobile version of Firefox. Our preliminary experiments on a Nokia N810 Internet tablet show that RCB-Agent can also efficiently support co-browsing using mobile devices. Currently we are applying our RCB techniques to enable a few interesting mobile applications. We also plan to implement RCB-Agent on other Web browsers. Enabling direct interactions between Web end-users can create many interesting interactive Internet applications. We believe that further exploring this end-user direct interaction capability and its applications is an important future research direction.

7 Acknowledgments

We thank the anonymous reviewers and our shepherd Niels Provos for their insightful comments and valuable suggestions. We also thank Professor Peter M. Vishton of the Department of Psychology at the College of William and Mary for his generous help in usability study. This work was partially supported by NSF grants CNS-0627339 and CNS-0627340.

References

[1] ANEIROS, M., AND ESTIVILL-CASTRO, V. Usability of Real-Time Unconstrained WWW-Co-Browsing for Educational Settings. In *Proc. of the IEEE/WIC/ACM International Conference on Web Intelligence* (2005), pp. 105–111.

[2] APPELT, W. WWW Based Collaboration with the BSCW System. In *Proc. of the 26th Conference on Current Trends in Theory and Practice of Informatics* (1999), pp. 66–78.

[3] ATTERER, R., SCHMIDT, A., AND WNUK, M. A Proxy-Based Infrastructure for Web Application Sharing and Remote Collaboration on Web Pages. In *Proc. of the IFIP TC13 Interantional Conference on Human-Computer Interaction* (2007), pp. 74–87.

[4] CABRI, G., LEONARDI, L., AND ZAMBONELLI, F. A proxy-based framework to support synchronous cooperation on the Web. *Softw. Pract. Exper. 29*, 14 (1999), 1241–1263.

[5] CHANG, M. L. CoBrowse Firefox Add-ons. https://addons.mozilla.org/en-US/firefox/addon/1469.

[6] COLES, A., DELIOT, E., MELAMED, T., AND LANSARD, K. A framework for coordinated multi-modal browsing with multiple clients. In *Proc. of the WWW* (2003), pp. 718–726.

[7] ESENTHER, A. W. Instant Co-Browsing: Lightweight Real-time Collaborative Web Browsing. In *Proc. of the WWW* (2002), pp. 107–114.

[8] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, 1999.

[9] GRAHAM, T. C. N. GroupScape: Integrating Synchronous Groupware and the World Wide Web. In *Proc. of the IFIP TC13 Interantional Conference on Human-Computer Interaction* (1997), pp. 547–554.

[10] GRAHAM, T. C. N., URNES, T., AND NEJABI, R. Efficient distributed implementation of semi-replicated synchronous groupware. In *Proc. of the ACM UIST* (1996), pp. 1–10.

[11] GREENBERG, S., AND ROSEMAN, M. GroupWeb: a WWW browser as real time groupware. In *Proc. of the ACM CHI Companion* (1996), pp. 271–272.

[12] HAN, R., PERRET, V., AND NAGHSHINEH, M. WebSplitter: a unified XML framework for multi-device collaborative Web browsing. In *Proc. of the ACM CSCW* (2000), pp. 221–230.

[13] ICHIMURA, S., AND MATSUSHITA, Y. Lightweight Desktop-Sharing System for Web Browsers. In *Proc. of the 3rd International Conference on Information Technology and Applications* (2005), pp. 136–141.

[14] JACOBS, S., GEBHARDT, M., KETHERS, S., AND RZASA, W. Filling HTML forms simultaneously: CoWeb architecture and functionality. *Comput. Netw. ISDN Syst. 28*, 7-11 (1996), 1385–1395.

[15] KOBAYASHI, M., SHINOZAKI, M., SAKAIRI, T., TOUMA, M., DAIJAVAD, S., AND WOLF, C. Collaborative customer services using synchronous Web browser sharing. In *Proc. of the ACM CSCW* (1998), pp. 99–109.

[16] MAH, B. A. An Empirical Model of HTTP Network Traffic. In *Proc. of the INFOCOM* (1997), pp. 592–600.

[17] RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. Virtual network computing. *IEEE Internet Computing 2*, 1 (1998), 33–38.

[18] ROSEMAN, M., AND GREENBERG, S. Building real-time groupware with GroupKit, a groupware toolkit. *ACM Trans. Comput.-Hum. Interact. 3*, 1 (1996), 66–106.

[19] SAKAIRI, T., SHINOZAKI, M., AND KOBAYASHI, M. CollaborationFramework: A Toolkit for Sharing Existing Single-User Applications without Modification. In *Proc. of the Asian Pacific Computer and Human Interaction* (1998), pp. 183–188.

[20] Ajax (programming). http://en.wikipedia.org/wiki/Ajax_(programming).

[21] Browser Statistics. http://www.w3schools.com/browsers/browsers_stats.asp.

[22] Cobrowse & Chat for Rich Ajax Applications - Backbase. http://www.backbase.com/products/ajax-applications/cobrowse.

[23] Document Object Model (DOM). http://www.w3.org/DOM.

[24] Firefox Extensions. http://developer.mozilla.org.

[25] http://point-at-infinity.org.

[26] Internet Explorer Browser Extensions. http://msdn.microsoft.com/en-us/library/aa753587(VS.85).aspx.

[27] Likert scale. http://en.wikipedia.org/wiki/Likert_scale.

[28] PageShare. https://www.pageshare.com/web/products/index.html.

[29] Port forwarding. http://en.wikipedia.org/wiki/Port_forwarding.

[30] SamePlace. http://sameplace.cc/wiki/shared-web-applications.

[31] XMLHttpRequest. http://www.w3.org/TR/XMLHttpRequest.

[32] XMPP Standards Foundation. http://www.xmpp.org.

[33] XPCOM. http://www.xulplanet.com/references/xpcomref.