Preservation and migration of browsing sessions in mobile environments

Eros Lever, Daniele Rossetti

June 2012

***Abstract*—Nowadays the web browsing activity on a mobile device may be influenced by several factors which may create unwanted delays or unpredicted behaviors. Connection drops are the most common obstacles for browsing and they may occur for different reasons, e.g. external circumstances or OS internal decisions as well. In our work we aim to provide a solution that can allow the user activities to be protected in this kind of situations and for this purpose we are introducing a browser session preservation system that allows a user to take a snapshot of an active web session state on a mobile browser. The system responds to user requests and automatically reacts as well to external events which may cause the corruption of the current session. The infrastructure allows the user to retrieve the snapshot at a later time to recover the same active web session, and possibly migrating it to a different device. The design of the system is based on a browser-side plug-in that can capture the browser session state.**

I. Problem Definition

Our work lives in a context of a smart and self-aware mobile operating system (i.e. MorphoneOS, Android-based OS[**?**]), that focuses its attention on improving several aspects of a mobile device and adapts itself to the surrounding environment, in order to provide a better user experience by smartly facing situations than can affect the device behaviour.

The main problem we addressed was the fact that the system can not guarantee by itself the preservation of the navigation session if a connection drop occurs, causing the user to partially or completely lose her activity.
The connection drop may occur with external events as well as with internal ones, and it can be predictable or unpredictable, here follows a list of example of these possible scenarios.

1) *Unpredictable External*
   Trivially, just think of a situation when the user is browsing a web page, compiling a form and submitting it when unexpectedly the data connection goes down, due to physical limitations such as distance from the base-station or interferences, causing the web transaction to be aborted and forcing the user to start again her operation.
2) *Predictable External*
   When the user enters a Faraday cage such as an elevator or an underground parking loft, the connection will likely drop, but this event could be predicted by a smart user, which can decide to delay her entrance if she is performing a transaction.
3) *Unpredictable Internal*
   In these category lie situations such as hardware failure inside the device or system crash; this kind of issues is practically impossible to recover from, without implying a restart from a known working situation.
4) *Predictable* Internal
   When the system itself, due to some internal choices, causes the connection to go down or to switch from a data network to another (e.g. switching from 3G connection to EDGE connection) to reduce the power consumption. As a matter of facts, rather than trying to improve the technology beyond the battery autonomy, which could be a way too tough job, the operating system may decide to pay more attention on using its available resources more efficiently. One way to handle it is to dynamically switching from a faster but more power wasteful data network (e.g. 3G or Wireless) to a slower but more power efficient one (e.g. EDGE) in some particular situations, such as when running out of power. Although it seems a quite simple mechanism, the system must guarantee that this power-saving switching behavior is completely transparent to the user, so that she can enjoy the benefits of a more power-conscious device without having her activity compromised. Unfortunately switching network will cause the connection to be stalled for a small timeframe and, as pointed out in the first case, it will likely corrupts the user web transaction.

In addition to these problematic situations, we also aimed to provide a solution for *another kind of scenario*, that is whenever the user may decide to move her activity to a different device and continue browsing without any additional effort. Unfortunately existing solutions do not allow to completely migrate a browsing session in such a transparent way that the user can resume her task like nothing changed.

Once stated this, our *goal* is to make sure that the OS become aware of this kind of situations and smartly react to them, in order to guarantee a better user experience to support the defined scenarios. We aim to achieve this goal by allowing the user to restore and preserve her browsing session as connection faults occur and to design a smart infrastructure to support session migration such that no effort are required to the user to complete the task.

II. Analysis

In this section we analyze definition, possible solutions and our implementation of the given problem.

A. Browser State Definition

The first step was to identify the set of elements that describe the state of the browser for the current session, keeping in mind that a session may be recovered also in a device completely different from the original one. We identified mainly two elements:

1) *Cookies*
   As well-known, HTTP is a stateless protocol and exploits cookies mechanism to keep the state of the current user's navigation. They basically contains pieces of data represented by strings of characters limited in size and duration that are used to track the same use from a page to the following ones. Therefore cookies were surely components to keep in consideration when saving a session state, especially when we would need to migrate the session and continue browsing on a different device[**?**].

2) *Current Visited Page*
   Trivially, we needed to send the user back to the page she was visiting. Since we wanted to guarantee that after the session recovery, the user would find herself exactly in the same state as before the restoring, keeping track of the URL visited may not be sufficient. We will discuss this topic later, since we took several ideas in considerations before finding the most appropriate for the aspired behavior.

B. Possible Solutions

After defining the elements we would need to gather from the browser to preserve the session state, we needed to find the best approach to interact with the browser to actually obtain them.

1) A first hypothesis was to build up a separate application, make it running in background and request the wanted information to the browser when needed. This approach may seems to have one important advantage that is compatibility, meaning the fact of being totally independent from the browser application used on the device. Unfortunately, due to the very different implementation of many browsers, it is impossible to build up an application that can effectively interact with any browser. It would have been necessary to design different interfaces for every browser we would have wanted to support. On the other hand, this was not the real drawback of this approach, since the number of available browsers for Android OS is not so significant and we could have focussed just on the two or three most popular. The reason why we discarded this solution was for its true limit, namely the fact that, mainly for security reason, a browser would not simply allow any external application to retrieve its private components, e.g. most of the browsers encrypt their cookies in order to prevent the stealing of sensible datas.
2) On second thoughts, our choice moved to a different solution. We realized that the best way to effectively interact with the browser core was to build up a component that could extend its functionality, still being inside the context of the browser itself, that is to say a browser add-on. This choice had all the features required to retrieve all the components we needed to have a consistent state of a session, since it allowed us to interact directly with the browser and all of its elements just from inside its core[**?**]. Although this solution might seem to have lack of portability and compatibility, since every browser needs different implementations for its extensions, we thought that accurately choosing one of the most common and supported browsers we could reach out to a wide target of devices as well.

Chosen Platform

Our target browser has been the popular Mozilla Firefox Mobile (codename: Fennec / current version: 14.0)[**?**]. The development of this application is significantly active and it is one of the fews, also among the most commonly used, that supports extensions (e.g. Google Chrome and Opera do not currently support add-ons in their current mobile versions), providing one of the most complete and advanced SDK for add-ons development. We also preferred Firefox since, according to us, Mozilla is a more trusted source than others small open-source projects and can guarantee a more stable and reliable solution. The official version available on Google Play (Android official application store) supports just ARMv7 processors and requires at least Android 2.1. Anyways we have been able to find a parallel development project[**?**] which supports ARMv6 as well, while Mozilla is currently working to officially support this family of processors as well. Thanks to that, we could support almost all the devices running AndroidOS.

C. Implementation

We just want to outline in this section the most interesting and problematic aspects of the development process.
As already pointed out, we needed to get the cookies from the browser and somehow preserve the current state of the last visited page, exploiting the fact of being completely able to interact with the browser thanks to the add-on design.

1) Saving a snapshot of a page: When we talk about the current state of the page, we refers to the fact that pages are not static but they may contain some data input by the user and not yet committed (e.g. a form being filled) and it also may have been generated dynamically by previous HTTP (POST/GET/) synchronous or asynchronous (AJAX) request.

- A very basic and simple idea was to just save the URL, but since, again , we want to make the session restoring transparent to the user as much as possible, it could not be sufficient : we would not had the same state if the

Figure 1.

page were built dynamically or just if some data have been input by the user. This is because nowadays pages are not always static documents but they may vary from time to time or even on a per access basis and it can be also altered with different kinds of interaction, such as user input and active code (Javascript or Flash).

- A more appropriate solution could have been saving the URL and additionally the input data, by parsing the HTML, and eventually facing some dynamic generation of the page by tracking all the HTTP requests made by the browser since the first page visited by the user. This solution would undoubtedly been too computationally heavy for a mobile processor and we did not want to add more stress in terms of power consumption or system's slowing down to the device.

- Alternatively, we moved on a more lightweight solution, which consists in saving directly the whole HTML page to a local file as a real snapshot of the current DOM, so that all the dynamically generated elements and inputs would have been preserved. Afterwards, restoring the page would have just meant reload it locally.

2) Problems of a local copy: Having a local copy of the last visited page solved the problems reported before, but on the other hand it also introduced a non-trivial issue. Loading it locally would have meant that all the relatives links contained in the page (hrefs,styles,scripts) would had just simply stopped working, since the base URL would have become localhost rather than the remotehost.

- *Replace the URLs*
  At first, we thought to solve this issue by replacing all the occurrences of relative URLs by their correct absolute value:

  /some_page.html $\Rightarrow$ *www.somehost.com/some_page.html*

  We exploited an open-source code from a web-sanitizer javascript library and adapted to our needs to parse the whole document and fix the (static) problematic URLs . Making some performance (ms/page) tests we got these quite unsatisfying results, in terms of time required to complete the task. In these tests we tracked the required time of saving the session adopting the proposed methodology; the test ran on known web pages allowing us to determine the amount of URLs correctly replaced. Since also the size of the page influences the requested time, we chose to have pages with three different weights.

  Although we could get good results in terms of required storage space, the URLs replacements was not so satisfying. With more complex URLs our system could not manage to fix all of them and in addition increasing the page size led to a significant performance loss.

Table I
PERFORMANCES

Website	RequiredTime	Size	URLsCorrectlyReplaced
Twitter	800ms	50kb	7/7
Facebook	930ms	200kb	10/12
Amazon	1900ms	350kb	21/39

- *Rebase the URLs*
  The fact that it is practically impossible to recognize and alter all the URLs, and that it required a high amount of computation, pushed us to find some different solution that could led us to more acceptable performances. We then spotted and took advantage of one of the brand new HTML 5 features which is the <base> tag[**?**] that actually rebase (i.e. set the base path) almost all the relative paths contained in the page. The only links that are not affected by the <base> tag are the ones referred in the CSS sections of the page and <link> tags that have a protocol relative "HREF" attribute. Addressing only those elements of the DOM we could get a really quick fix by URL replacing, without affecting the system performance. As expected, appending just the *base* tag and accomplishing a fast parse-and-replace on a limited number of code lines, gave us more satisfying results.

Table II
PERFORMANCES

Website	RequiredTime	Size	URLsCorrectlyReplaced
Twitter	300ms	50kb	7/7
Facebook	330ms	200kb	12/12
Amazon	900ms	350kb	39/39

  Since we knew in advance which kind of URLs needed to be replaced, we could avoid to parse all the document, just address them in the DOM tree and get more brilliant results.

- *Input data*[**?**]
  Another problem we encountered was the fact that Firefox (even in the Desktop edition) does not dump by default the content of user edited form fields. There are basically two methods to cope with this issue, that are tracking all the user clicks and typed keys, or taking the content of the form the exact moment the page is being saved. Since the first way introduces unneeded power consumption and may interfere with user privacy, having she thinking that something is tracking all her input, we decided to adopt the second methodology.
  When saving the session, the content of the input forms is made effective transcribing its value with the *setAttibute* function in Javascript.

3) Effectively responds to connection loss: We thought of different possible solutions that could help preventing such a situation, namely a first one having a timer that regularly saves the session and a second one that detects connection

drops and handles them saving the session before it gets lost.

- We first decided to use a timer, but experimentally we saw this solution not being truly effective. The reasons around this impreciseness were the trade off between saving interval and power efficiency, more in the detail: to avoid losing last performed actions during the session, it has to be saved frequently, obviously causing unwanted power consumption. Although there are a few improvement that can be took in account, such as avoiding saving multiple times the very same session and other more complicated as altering only the differences from the previous state when saving the same web page repeatedly, these proposed enhancements still do not really provide a valid solution since they requires the session to be saved frequently maybe adding costs in computing and tracing differences.

- Therefore we decided to take advantage of Firefox Mobile built-in notifications and observers system, mainly the "network:offline-about-to-go-offline"[**?**] that detects connection drops and notifies our extension before altering the active page. This way we are able to save the session only when it is truly necessary, avoiding wastes in power consumption.

4) Session packing: We distinguished between HTML document and cookies using two different files, one each. Cookies are saved encoded in JSON format, which is handy to use in Javascript, as they are retrieved from the browser APIs (we will discuss the security limits of this approach in IV-C). The obtained HTML is then combined with the cookies, and these two items are saved in a ZIP file to shrink the size and avoid wastes in term of memory footprint on the device and transfer duration when sent over the network to another device. The resulted ZIP file is then stored in a local directory sessions on the SD card, making it accessible by third-party applications, such as SwishApp[**?**], which allows to transfer datas between near devices. The interaction between our extension and the SwishApp is detailed in section III-C.

5) Session restoring: Being able to restore the session means being able to bring back the user to the very same web page she was interacting with, allowing her to continue her navigation using the cookies that were saved. Our extension provides a restore functionality triggerable by the user, that once invoked (through a resource within the extension, accessed as a *chrome://* registered path) it will unpack the session from the ZIP file provided, update the cookies with their saved value, and then show the HTML document.

III. Test Case

In this section we describe three scenarios used by our tests to point out different aspects of session handling.

A. Simulating connection loss

To test the success of our system with respect to unexpected connection losses we design a very simple Android background application that randomly turn off the Wireless connection. Thanks to the reliable system of Android and Firefox in detecting the network going down, we could manage to handle successfully every connection faults, correctly saving and restoring the navigation session.

B. Forced Sessions Destruction

To test if our system actually managed to successfully and completely recover from scratch a session we added a debug functionality to our add-on that actually destroys all the cookies saved by the browser. A simple test consisted in logging in a website, fill in some form, saving the session and then destroying all the cookies (meaning that the website can not validate the logged user). After restoring the user was still logged in with the input data in the right place, proving that the session was correctly recovered.

C. Session Migration with SwishApp

To test the adaptativity of our system with respect to the session migration we exploited the Swish application to move a session from a device to another. In order to support it we had to build an helper application that could manage the session file as soon as the second device receive it, since the SwishApp just opens the received file with the default handler. More precisely the helper application registers itself as the default handler for .session file (custom extension we defined for our session files), just moves the received file to the sessions directory and opens Firefox with a special URL, calling back the add-ons functions that start the restoring process.Once the session is moved, the system managed to successfully recover the session, as already proved with III-B test case.

IV. Limits and Possible Improvements

Even if our work provides a working solution, there can be few limitation that could be taken into account for possible future works.

A. Client-side issues

Our current solution does not take care of the running state of client-side scripts, such as Javascript, meaning that the values of the variables will be lost during the session saving/restoring process. As a practical example, if a page contains a counter, dynamically updated by Javascript, the saving system can not access the internal state of the script so, after restoring, the counter will be reset to the initial value. Similarly, Flash scripts state would not be recovered, meaning that for example watching a video on Youtube and then restoring the activity would not take the user the right timing of the video. Another problem is that switching the context of the script from remote to local may lead to some security issues, since when executed in remote pages Flash objects may have different privileges with respect to when they are executed inside a local page.

B. Multimedia streams

When dealing with streams, such as audio or video media, it could be useful to restart from a specific position. To detect this specific position is really an hard task to accomplish, since the methodology may vary from case to case. Moreover technologically speaking, it is not always possible to ask a server for only a part of a stream. These limits prevented us to cope with multimedia streams.

C. Security

Migrating cookies to another device may lead to security issues, for example migrating an authenticated session to a device under control of another person, can potentially introduce an abuse of the cookies performing operations as if the receiving person were the other party. This is intrinsic to cookies and impossible to prevent without modifying the authentication check server-side, adding controls such as IP address or User-Agent in use.

D. Mobile-vs-Desktop pages layout

When migrating a session from a device to another, it may be the case in which one should see the mobile version of a webpage and the other the desktop one. When dealing with bare links, it is normally just matter of add a “m.” in front of the remote hostname, but since this is not a standard we can not assume this methodology to be always valid. Furthermore, using two different hostnames may lead to invalidate the HTTP cookies, preventing the user the ability to continue her browsing session.