The Virtual Splitter: Refactoring Web Applications for the Multiscreen Environment

Mira Sarkis, Cyril Concolato, Jean-Claude Dufourd

The Virtual Splitter: Refactoring Web Applications for the Multiscreen Environment

Mira Sarkis, Cyril Concolato, Jean-Claude Dufourd
Telecom ParisTech; Institut Mines-Telecom; CNRS LTCI
{sarkis, concolato, dufourd}@telecom-paristech.fr

ABSTRACT

Creating web applications for the multiscreen environment is still a challenge. One approach is to transform existing single-screen applications but this has not been done yet automatically or generically. This paper proposes a refactoring system. It consists of a generic and extensible mapping phase that automatically analyzes the application content based on a semantic or a visual criterion determined by the author or the user, and prepares it for the splitting process. The system then splits the application and as a result delivers two instrumented applications ready for distribution across devices. During runtime, the system uses a mirroring phase to maintain the functionality of the distributed application and to support a dynamic splitting process. Developed as a Chrome extension, our approach is validated on several web applications, including a YouTube page and a video application from Mozilla.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; D.2.11 [**Software Engineering**]: [Software Architecture]

Keywords

Application Distribution, Authoring, Multiscreen, Web Application

1. INTRODUCTION

Recently, intense research activity has been focused on multi-screen scenarios [4][6] inside home environment where cooperation between heterogeneous devices is leveraged. In such a cooperative environment, a "Multi-Screen Application" (MSA) is an application distributed across multiple connected devices, each having a screen, and designed to offer a convivial experience. Examples of MSA are: using a tablet to display additional information synchronized with a TV program, or using the interaction capabilities of smartphones (e.g., touch screen) jointly with the large screen and processing power of a PC or a TV to display media elements.

Multi-screen applications impose multiple challenges to the application developers. First, they have to design an application that leverages the multi-screen environment and copes with the diversity of devices. Then, they have to determine how the application content will be distributed across devices based on their specific capabilities, to manage the distributed content and to maintain the synchronization and consistency of the distributed content. The use of web technologies helps reducing the complexity of these tasks and increasing the possibility of deploying a ubiquitous application. In this paper, the term application refers to a web application. Many applications were created before the development of the multi-screen concept. Many of them are actually made of different components that could benefit from being distributed. However, few of them were designed in a modular manner that facilitates code distribution.

With such a motivation and focusing on challenges related to authoring MSAs, this paper aims to meet one principal objective: *To propose a new approach that reuses existing single-screen applications and refactors them for the multi-screen environment.* In contrast to existing works, our approach is based on mapping the application content onto available devices by automatically analyzing the content following the author or the user choices, on splitting the application into two sub-applications and synchronizing them while maintaining the overall application functionality and supporting a dynamic splitting process. The ability for the user to guide this splitting process is novel and opens up many usage scenarios.

This paper is organized as follows. Section 2 compares our approach with related work. Section 3 describes the virtual splitter architecture, including the mapping, annotation, splitting and mirroring phases. An implementation of the solution, an evaluation and a survey of its limitations are given in Section 4. Finally perspectives and conclusions are drawn in Section 5.

2. STATE OF THE ART

In prior works, a 'WebSplitter' [5] was proposed to split XML-based applications, based on a metadata file. This file is unique for each application and determines which application portions, i.e., which XML elements can be seen on each user device. The splitter requires a middleware proxy that splits the application content into partial views and a client-side component that receives data pushed by the server. The XML splitter architecture is centralized and requires a man-

DocEng '15, September 08 - 11, 2015, Lausanne, Switzerland
Copyright 2014 ACM 978-1-4503-2949-1/14/09 ...$15.00.
http://dx.doi.org/10.1145/2644866.2644893.

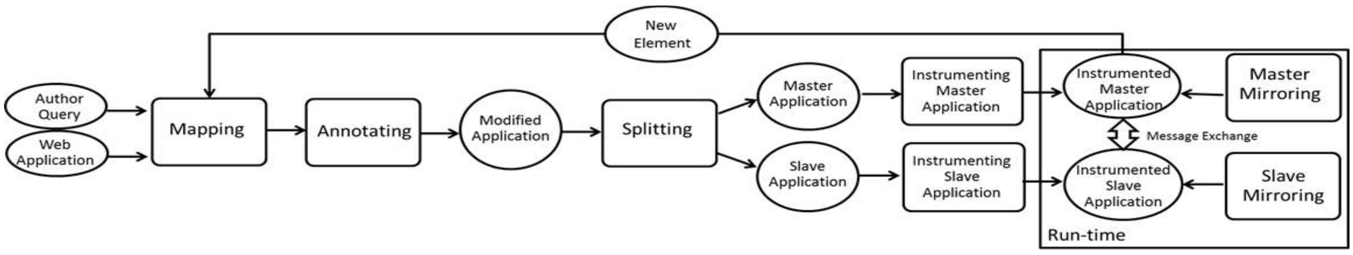


Figure 1: Virtual Splitter Architecture

ual mapping for each XML element of the application. In his research, Cheng [3] proposed a virtual browser capable of separating the application logic from its rendering. The logic is kept within a virtual web page. Automatically the virtual browser splits the main DOM tree into multiple DOM trees and maps these trees to corresponding devices as denoted in a hint file that is specific to each application and manually created by the developer. Cross-device operations are executed in a centralized manner depending ultimately on the browser. Bassbouss et al.[2] outlined how to enable traditional applications to become multi-screen-ready. The application is developed as a single-screen application and requires a multi-screen enabled browser. Based on metadata information provided manually by the developer, specific elements are assigned to a remote device while always being shown on the main device.

In contrast to [3] and [5] our system has a decentralized architecture. Similar to [2] it delivers master-slave applications. The common part for the three previous works is that each application is analyzed and mapped separately and manually by the author (via a hint file or metadata). This means there is no generic analysis method that can be applied to a set of applications. On the contrary, we propose an extensible system that is capable of automatically analyzing the application and mapping its elements, thus simplifying the author's task and involving the end user in mapping her application. The author only has to determine the analysis criterion.

3. THE VIRTUAL SPLITTER

3.1 Overview

A web application consists mainly of HTML, CSS and JavaScript (JS) resources, that are tightly linked. Links exist between elements in the DOM tree (e.g., parent-child, siblings), between the DOM and JS when the JS accesses elements by specific attributes (e.g., id) or by document navigation, between the DOM and the CSS via selectors, etc. In such a context, splitting an application will break some links and cause a failure in the application look and functionality.

In addition, an application presents two dynamic aspects that make the splitting approach more complicated. On one side, splitting a web application requires support for its dynamism since elements are continuously modified, created, moved or removed during runtime. Supporting this dynamism during run-time is essential to ensure the coherence between elements in each of the distributed application. On the other side, automatic partitioning of the application script is a hard task since JS is a flexible and dynamic language characterized by high-order functions, closures, 'eval' function which dynamically evaluates a string expression, etc. In this paper, we take care of the links and dynamicity of the application by focusing on splitting only the HTML document, maintaining links and providing a dynamic splitting phase during run-time while keeping the JS code as a whole running on one device. The following subsections present a detailed description of the virtual splitter architecture as illustrated in Figure 1.

3.2 HTML Elements Mapping

The mapping phase is the first phase in our system. Its purpose is to determine which of the application elements map to the devices involved in the multi-screen experience. In any multiscreen scenario, at least two devices are cooperating. The literature refers to the smartTV as the first screen and assigns the expression 'companion screen' [1] or 'second screen' [8] to a device providing a means of interaction with the smartTV services. In this work, the 'principal device' is responsible of processing the main application logic, while the 'secondary device' receives processing results only if it is concerned. As depicted in Figure 1, the mapping phase takes as input a query from the application author, when the application is pre-processed offline; or from the user, when the whole process of splitting the application is done at run-time. In our approach, we have envisaged several possible mapping techniques: based on the analysis of HTML elements, their associated semantics and roles, discussed in Section 3.2.1; or based on the visual rendering of elements, discussed in Section 3.2.2. These techniques could be also combined. For instance, mapping only interactive elements placed in a certain region of the screen to the secondary device. This phase is extensible and other analysis techniques could be used here. The query can therefore be either a simple query indicating which mapping technique should be used along with its specific parameters (e.g., element category or position); or a combined query using boolean logic. The output of the mapping phase is two lists of elements, one for each device.

3.2.1 Semantic Mapping

As a possible mapping criterion, we describe here a semantic based approach. It is fully automatic, selected by either the author or the user. In the context of multi-screen applications, we analyzed the HTML5 elements defined in the standard to determine their roles and how they could be classified for the purpose of application splitting. We identified four relevant classes: **interactive** elements (i.e., a, area, button, datalist, form, input, keygen, textarea, nav, optgroup, option, output, select), **multimedia** elements (i.e., video, audio, source, track), **non-interactive, non-multimedia visual** elements (i.e., caption, dialog, figcaption, h1 to h6, hgroup, img, kbd, label, legend, object, p, progress) and **other** elements. As part of the query parameters, the author indicates one or more class of elements to be moved to the secondary device depending on the characteristics of the available devices. For instance, if a smartTV is present, the system or the user may decide to move only multimedia elements on that device. As another example, if a touch screen is present, the interactive elements may be moved onto that device. This mapping technique produces the lists of elements as follows. First, for each element of the application, the algorithm compares it to the tag names present in the classes indicated in the author query. If the element falls within the indicated classes, it is added to the secondary device list. If the element is a composite element (i.e., div, table, iframe) the algorithm iterates over the its children first. If there is no match and if the element does not belong to the 'other' class, it is added to the primary device list. Then, the algorithm detects any change in the element basic role by checking its attributes, mainly declarative event listeners (e.g., 'onclick') since they are responsible of making elements interactive. A non-interactive element which role is only displaying content i.e., image, becomes interactive if it has an event listener that lets users interact with it. In addition, we exploit the semantic links that are created between HTML elements (e.g., 'for' attribute) by keeping these elements together in the same list.

3.2.2 Screen Region Mapping

We also investigated a region-based approach as a mapping criterion. It is fully automatic but this time based on the application visual rendering. It is selected during runtime by the end-user in her browser. Once the user selects a screen region, the system detects elements within that rectangular region to produce the secondary device list. All other elements are placed in the primary device list.

3.3 Annotating Elements

As depicted in Figure 1, the second phase is the annotation. It prepares the application for the splitting phase and takes as input the lists of elements produced by the mapping phase. The annotation algorithm starts by processing each DOM leaf element and sets the value of the 'data-device' attribute to 'device2' for elements in the secondary device list and 'device1' for elements in the primary device list. It should be noted that the previous mapping phase may have populated those lists with only some DOM elements, and not all elements in the tree, for instance only interactive elements or only the elements located in a given region. Thus, each remaining leaf element (resp. parent element) in the DOM tree is annotated with the value of its siblings (resp. its children), if the annotation is the same across siblings (resp. children), or with the value 'dev1&dev2' if they differ, meaning that the element will be present on both devices. As a result, the application is totally annotated: each element contains metadata information, in a 'data-device' attribute, reflecting its target device(s).

3.4 Splitting Application Content

After the annotation phase, during run-time, the splitting phase relies on the element metadata information to form two separated applications: a master and a slave application as Figure 1 shows. From the original application, elements annotated with 'device1' or 'dev1&dev2' values are kept visible on the screen of the primary device. Elements annotated with 'device2' value are hidden on the primary device. This forms the master application. These hidden elements serves as a shortcut whenever the application main logic requires reading or modifying elements of the remote application on the secondary device, thus the term 'virtual splitter'. Elements annotated with 'device2' and 'dev1&dev2' values are extracted from the original application and imported to the new slave application running on the secondary device. On the master application, in addition to the retained original application logic, JS code is added in an instrumentation phase and aims at making the master application capable of working synchronously with its slave (see Section 3.5). This code also supports the application dynamism and ensures a dynamic mapping and splitting at run-time. On the slave application, the JS code makes the application capable of collecting user interactions, redirecting them to the master, receiving and integrating changes made to its DOM tree.

3.5 Mirroring Application Contents

The virtual splitting phase described in Section 3.4 duplicates some content between the master and slave applications. The role of the mirroring phase is to ensure that the slave application has a DOM tree that is an accurate mirror of the hidden DOM tree in the master application. It is performed as follows: On the 'primary device', any dynamic change affecting elements of the 'secondary device' (e.g., node modification, removal or creation) is mirrored to that device through change messages. Upon receiving a message, the application running on the 'secondary device' updates its DOM tree and integrates this change. On the 'secondary device', any user interaction (e.g., clicks, data inputs) is captured and propagated to the 'principal device' where the interaction handler is processed.

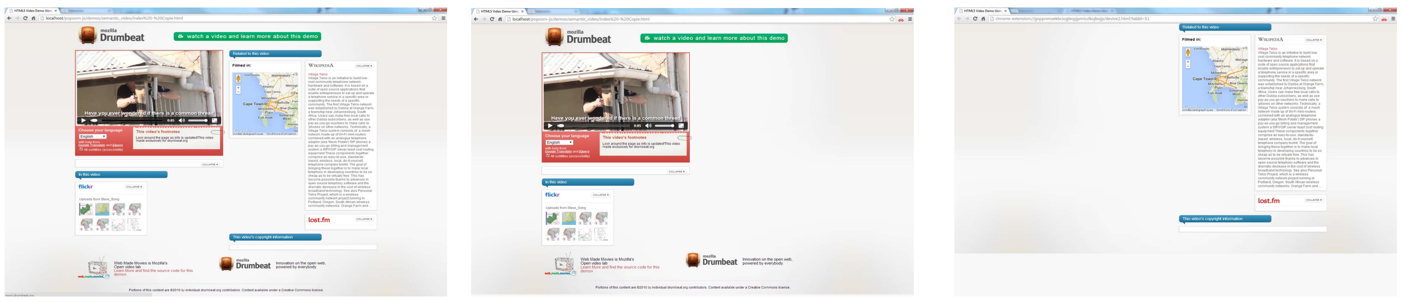
4. IMPLEMENTATION AND RESULTS

4.1 Implementation

We decided to implement the virtual splitter as a Google Chrome extension, first to enable on-the-fly instrumentation of the application without having to change the application itself, and second for the better debugging environment compared to the situation on devices. Master and slave applications are rendered as tabs in the browser and communication between them is done using the postMessage API that is similar to the communication mecanism in COLTRAM[4].

As discussed in Section 3.5, to detect relevant changes in the DOM we use the Mutation Summary library[1] which is based on the working draft of the Mutation Observer API [2]. A Mutation-Summary object is configured to watch changes made to elements with 'device2' annotation. If any change happens to these elements, their descendants or attributes, the extension sends a message to the 'secondary device'. The message contains a list of changes. Each change object consists of the type of change, the concerned node, its position in the DOM tree (i.e., parent and previous sibling node), the concerned attribute(s) and the new value of a text node. However, the Mutation Summary library suffers from some limitations. For instance, it cannot detect changes made to HTML elements using JS functions especially if they are not reflected on the DOM tree. To overcome this limitation, as well as supporting dynamism, we use the Monkey Patching

[1]see `http://code.google.com/p/mutation-summary`
[2]see `http://www.w3.org/TR/domcore`



(a) Main Application (b) Master Application (c) Slave Application

Figure 2: Splitting The Semantic Video Application Based On The Screen-Region Criterion

technique [7], to extend in JS some native browser functions with custom code, in particular: the 'createElement' function is extended to detect the creation of new elements and to trigger dynamically the mapping, annotation and splitting of these elements; the 'setAttribute' function to update the Mutation Summary configuration, to enable the mirroring of newly created attributes; and the 'addEventListener' function to overcome the limitation of the Mutation Summary library, and to replace an event handler triggered on the slave application to a call to the master application.

4.2 Results and Discussion

We tested our system on different applications from simple static pages to dynamic applications, among them: a semantic video application[3], relying on the Popcorn and JQuery libraries and showing various information (e.g., map, text, images) synchronized with a video. We first used the region-based mapping. On the video application, we separated the video and Flick'r images from the additional information as Figure 2 shows. This experiment verified the performance of the mirroring phase by maintaining a reliable mirror and the synchronization between both master and slave applications. In addition, no compatibility issues were reported between our instrumented code and the JS libraries. We then used the semantic mapping to split a YouTube page and to separate the interactive class of elements from the other classes (i.e., non-interactive and multimedia). As a result, the video runs on the master with all the comments of users while all buttons, anchors, guide container that proposes additional videos to watch later are moved to the secondary device.

Based on this, we identified a few areas for future improvement. This includes mainly the re-organization of application layout based on devices screen characteristics, the solving of some problems related to the use of relative URLs, the handing of HTML elements such as Canvas that have no inner DOM representation but are controlled by JS code. We will also conduct more systematic testing and validation.

5. PERSPECTIVES AND CONCLUSIONS

In the multi-screen context, this paper proposed a system to transform existing applications from single-screen to multi-screen applications based on author or user choices. The system consists of an automatic and extensible mapping phase that analyzes the application semantically, visually or a combination of these two, an annotation and a virtual splitting phase that result in master-slave applications. A mirroring phase ensures the correct functionality and synchronization between both parts and a dynamic splitting process. We validated our system on two existing applications: YouTube and semantic-video, and we verified the correct content mapping, synchronization and application functionality. As a future step, we aim at implementing our system in the COLTRAM multi-screen platform and extending this system with a context driven splitting technique that collects information concerning involved devices and creates dynamically adaptive mapping criteria.

[3]see `http://popcornjs.org/demo/semantic-video`

6. REFERENCES

[1] S. Basapur, H. Mandalia, S. Chaysinh, Y. Lee, N. Venkitaraman, and C. Matcalf. FANFEEDS: Evaluation of socially generated information feed on second screen as a tv show companion. In *EuroiTV '12 Proceedings of the 10th European Conference on Interactive TV and Video*, pages 87–96, Berlin, 2012.

[2] L. Bassbouss, M. Tritschler, S. Steglich, K. Tanaka, and Y. Miyazaki. Towards a multi-screen application model for the web. In *IEEE 37th Annual Computer Software and Applications Conference Workshops*, pages 528–533, Japan, July 2013.

[3] B. Cheng. Virtual browser for enabling multi-device web applications. In *Proceedings of the Workshop on Multi-device App Middleware*, Montreal, Quebec, December 2012.

[4] J.C. Dufourd, M. Tritschler, L. Bassbouss, R. Bouazizi, and S. Steglich. An open platform for multiscreen services. In *In the 11th European Interactive TV conference EuroITV*, Como, Italy, 2013.

[5] R. Han, V. Perret, and M. Naghshineh. Websplitter: A unifed xml framework for multi-device collaborative web browsing. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, pages 221–230, Philadelphia, USA, December 2000.

[6] J. Jang, H. Nam, and Y. Kim. Mobile device-controlled live streaming traffic transfer for multi-screen services. In *IEEE International Conference on Information Networking*, pages 415 – 420, Bali, February 2012.

[7] B.S. Lerner, H. Venter, and D. Grossman. Supporting dynamic, third-party code customizations in javascript using aspects. *SIGPLAN Not.*, 45(10):361–376, 2010.

[8] E. Tsekleves, L. Cruickshank, A. Hill, K. Kondo, and R. Whitham. Interacting with digital media at home via a second screen. In *9th IEEE International Symposium on Multimedia*, pages 201–206, December 2007.